Subtyping Recursive Types

Roberto M. Amadio[1]
CNRS-CRIN, Nancy

Luca Cardelli
DEC, Systems Research Center

Abstract

We investigate the interactions of subtyping and recursive types, in a simply typed λ-calculus. The two fundamental questions here are whether two (recursive) types are in the subtype relation, and whether a term has a type.

To address the first question, we relate various definitions of type equivalence and subtyping that are induced by a model, an ordering on infinite trees, an algorithm, and a set of type rules. We show soundness and completeness between the rules, the algorithm, and the tree semantics. We also prove soundness and a restricted form of completeness for the model.

To address the second question, we show that to every pair of types in the subtype relation we can associate a term whose denotation is the uniquely determined coercion map between the two types. Moreover, we derive an algorithm that, when given a term with implicit coercions, can infer its least type whenever possible.

Appears in: Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, pp. 104-118, ACM Press, 1991.
Also to appear in TOPLAS.

SRC Research Report 62, August 14, 1990. Revised January 1, 1993.
© Digital Equipment Corporation 1990,1993.
This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individuals contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

---

[1]This author's work has been supported in part by Digital Equipment Corporation and in part by the Stanford-CNR Collaboration Project.

A preliminary version of this paper appears in the Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, ACM Press, 1991.

Contents

1. Introduction
- **1.1 Types**
- **1.2 Subtypes**
- **1.3 Equality of Recursive Types**
- **1.4 Subtyping of Recursive Types**
- **1.5 Algorithm outline**
- **1.6 Formal development**

2. A Simply Typed λ-calculus with Recursive Types
- **2.1 Types**
- **2.2 Terms**
- **2.3 Equations**

3. Tree Ordering
- **3.1 Subtyping Non-recursive Types**
- **3.2 Folding and Unfolding**
- **3.3 Tree Expansion**
- **3.4 Finite Approximations**

4. An Algorithm
- **4.1 Canonical Forms**
- **4.2 Computational Rules**
- **4.3 Soundness and Completeness of the Algorithm**
- **4.4 An Implementation**

5. Typing Rules
- **5.1 Type Equivalence Rules**
- **5.2 Completeness of Equivalence Rules**
- **5.3 Subtyping Rules**
- **5.4 Completeness of Subtyping Rules**

6. A Per Model
- **6.1 Realizability Structure**
- **6.2 Complete Uniform Pers**
- **6.3 Completeness of an F-interpretation**

7. Coercions
- **7.1 Definability**
- **7.2 Inference**

8. Conclusion

9. Acknowledgments

References

1. Introduction

Subtyping is an inclusion relation between types that is present to some degree in many programming languages. Subtyping is especially important in object-oriented languages, where it is crucial for understanding the much more complex notions of inheritance and subclassing.

Recursive types are also present in most languages. These types are supposed to *unfold* recursively to match other types. Moreover, unfolding must preserve typing soundness and not cause the compiler to diverge.

In this paper we investigate the interaction of *unrestricted* recursive types with subtyping. This interaction is present in some modern languages based on *structural type matching* (where type equality or subtyping is determined by some abstract type structure, and not by how types are syntactically presented). In the past, recursive types have often been restricted by other language features; for example by explicit unfolding in ML, and by *name matching* in Modula-2. Algol68 was the first language to rely on a structural type equality algorithm for recursive types. Thereafter name matching became popular, largely because it is easier to implement but also because it prevents accidental matches based on type structure.

Name-matching determines type equality by relying, at least partially, on the names assigned to types in a given program, instead of on their structure. With name matching, recursive analysis can stop at occurrences of type names. Unfortunately there is no general definition of name matching; each language, and sometimes each compiler, implements it slightly differently. Types with the same meaning (in the eye of the programmer) may or may not be equated in different runs of the compiler, depending on irrelevant textual perturbations that affect the name matching rules.

The inconsistency of name-matching rules becomes a problem in distributed environments, where type definitions and data may migrate outside the compiler or program run in which they are created. Types and data should have a meaning independent of particular runs, hence languages such as Modula-3 [22] and other experimental languages such as Amber [10] and Quest [9, 12] concerned with data persistence and data migration, have again adopted structural matching. Since these languages also rely on subtyping, structural subtyping becomes an issue. Because of various language design issues, Modula-3 restricts itself to structural equivalence plus a limited form of structural subtyping; in this paper we deal with the unrestricted combination of recursion and subtyping, which forms the basis of Amber and Quest.

With this motivation, we investigate type systems with recursive types and subtyping, and the related problems of structural matching and structural subtyping. Structural matching techniques are well known, and have strong connections with well-understood theoretical concepts. Structural subtyping is a much newer subject. We provide the first complete theory of recursive subtypes that leads naturally to an effective type theory and to typechecking algorithms. In practice it is easy to adapt algorithms for structural typing to structural subtyping (although to our knowledge, this was first done in Amber), but formalizing the type rules and the proofs of correctness of the algorithms is more challenging. We show that both our algorithm and our type rules are complete with respect to a natural notion of subtyping.

In the rest of the introduction we provide the basic intuitions about recursive subtypes, and we illustrate the main problems along with several non-solutions. Section 2 formalizes the syntax of a basic calculus with recursive types and section 3 introduces a subtyping relation based on a tree ordering. Section 4 describes a subtyping algorithm, and section 5 describes the corresponding type rules. A partial equivalence relation model is given in section 6. Finally, section 7 relates subtyping to type coercions.

1.1 Types

A *type*, as normally intended in programming languages, is a collection of values sharing a common structure or shape. Examples of *basic* types are: Unit, the trivial type containing a single element, and Int, the collection of integer numbers. Examples of *structured* types are: Int→Int, the functions from integers to integers; Int×Int, the pairs of two integers; and Unit+Int, the *disjoint union* of Unit and Int consisting of either a unit value marked "left" or an integer marked "right" (given two arbitrary but distinct marks).

A *recursive type* is a type that satisfies a *recursive type equation*. Common examples are:

$$\text{Tree} = \text{Int} + (\text{Tree} \times \text{Tree})$$

the collection of binary trees with integer leaves, and:

$$\text{List} = \text{Unit} + (\text{Int} \times \text{List})$$

the collection of lists of integers. Note that these are not definitions of Tree and List; they are equational properties that any definition of Tree and List must satisfy.

There are also useful examples of recursion involving function spaces, typical of the object-oriented style of programming:

$$\text{Cell} = (\text{Unit} \to \text{Int}) \times (\text{Int} \to \text{Cell}) \times (\text{Cell} \to \text{Cell})$$

A Cell is interpreted as the collection of integer-containing memory cells, implemented as triples of functions *read*: Unit→Int, *write*: Int→Cell, and *add*: Cell→Cell. In each of these functions the current cell is implicit, so for example *add* needs only to receive another cell in order to perform a binary addition.

Recursive types can hence be described by equations, and we shall see that in fact they can be unambiguously *defined* by equations. To see this, we need some formal way of reasoning about the solutions of type equations. These formal tools become particularly useful if we start examining problematic equations such as $t = t$, $s = s \times s$, $r = r \to r$, etc., for which it is not clear whether there are solutions or whether the solutions are unique.

It is appealing to set up sufficient conditions so that type equations have *canonical* solutions. Then, if we have an equation such as $t = \text{Unit} + (\text{Int} \times t)$, we can talk about *the* solution of the equation. Such a canonical solution can then be indicated by a term such as $\mu t.\text{Unit} + (\text{Int} \times t)$; *the* type t that is equal to $\text{Unit} + (\text{Int} \times t)$. Here $\mu t.\alpha$ is a new type construction just introduced for denoting canonical solutions.

To say that $L \triangleq \mu t.\text{Unit} + (\text{Int} \times t)$ (where $\triangleq$ means *equal by definition*) is the solution of the List equation, implies that L must satisfy the equation; that is, $L = \text{Unit} + (\text{Int} \times L)$ must be

provable. This requirement suggests the most important rule for the $\mu t.\alpha$ construction, which amounts to a one-step unfolding of the recursion:

$$\mu t.\alpha \;=\; [\mu t.\alpha/t]\alpha$$

meaning that $\mu t.\alpha$ is equal to α where we replace t by $\mu t.\alpha$ itself. In our example we have:

$$L \;=\; \mu t.\mathrm{Unit}+(\mathrm{Int}\times t) \;=\; [L/t](\mathrm{Unit}+(\mathrm{Int}\times t)) \;=\; \mathrm{Unit}+(\mathrm{Int}\times L)$$

which is the equation we expected to hold.

Having discussed recursive types, we now need to determine when a value belongs to a recursive type. The rule above for $\mu t.\alpha$ allows us to expand recursive types arbitrarily far, for a finite number of expansions. Hence, we can postulate that a finite value belongs to a recursive type if it belongs to one of its finite expansions according to the ordinary typing rules. That is, we push the troublesome μ's far enough until we no longer need to consider them.

However, if the values are not finite, for example if they are defined recursively, we may not be able to push the μ's out of the way. In that case, we need to provide adequate notions of finite *approximations* of values and types, and postulate that a value belongs to a type when every approximation of the value belongs to some approximation of the type. An approximation α^n of a type expression α is an appropriate truncation of α at depth n, hence it is different from an unfolding. This will be made precise in later sections.

1.2 Subtypes

If types are collections of values, *subtypes* should be subcollections. For example, we can introduce two new basic types $\bot$ (*bottom*), the collection containing only the divergent computation, and $\top$ (*top*), the collection of all values. Then $\bot$ should be a subtype of every type, and every type should be a subtype of $\top$. We write these relations as $\bot \le \alpha$ and $\alpha \le \top$.

Function spaces $\alpha \to \beta$ have a subtyping rule that is *antimonotonic* in the first argument. That is,

$$\alpha \to \beta \le \alpha' \to \beta' \quad \text{if} \quad \alpha' \le \alpha \quad \text{and} \quad \beta \le \beta'$$

For example, if Nat ≤ Int, and f: Int→Cell stores an integer into a cell, then f is also willing to store a natural number into a cell, that is f: Nat→Cell. Hence, it is sound to have Int→Cell ≤ Nat→Cell, but not the opposite. This antimonotonic rule is familiar in object-oriented programming, where it is one of the main considerations for the correct typechecking of methods.

Adequate subtyping rules can be found for all the other type constructions we may have. For example, for products we have $\alpha\times\beta \le \alpha'\times\beta'$ if $\alpha\le\alpha'$ and $\beta\le\beta'$. Similarly, for disjoint unions we have $\alpha+\beta \le \alpha'+\beta'$ if $\alpha\le\alpha'$ and $\beta\le\beta'$.

What is, then, subtyping for recursive types? The intuition we adopt is that two recursive types α and β are in the subtype relation if their *infinite* unfoldings also are in this relation, in some appropriate sense. We might at first just consider finite unfoldings α^+ of a type α, and require that "$\alpha \le \beta$ if for every α^+ of α there is a β^+ of β with $\alpha^+ \le \beta^+$". However, we shall see shortly that this condition is not strong enough. Hence, we insist on inclusion of infinite

unfoldings. This is made precise by the notion, mentioned above, of finite approximations α^n of a type α, and by defining "$\alpha \le \beta$ if, for every n, $\alpha^n \le \beta^n$ ".

Unfortunately, the formal subtyping rules for recursive types and the related algorithms cannot rely on approximations, since "$\alpha^n \le \beta^n$ for every n" involves testing an infinite number of conditions. The subtyping rules should rely instead on "finitary" rules, and it is therefore not so obvious how to invent a collection of rules that achieve the desired effect. For example, a first idea might be simply to say that:

$$\text{if} \quad \alpha \le \beta \quad \text{then} \quad \mu t.\alpha \le \mu t.\beta \qquad (1)$$

where t may occur free in α and β. By this we can show that, for example, $\mu t.\ \top \to t \le \mu t.\ \bot \to t$, just from the assumption that $t \le t$. Unfortunately we also have:

$$(1) \qquad \text{implies} \qquad \alpha \triangleq \mu t.\ t \to \bot \ \le \ \mu t.\ t \to \top \triangleq \beta$$

and this is quite wrong. By unfolding both α and β twice we get:

$$(1) \qquad \text{implies} \qquad (\alpha \to \bot) \to \bot \ \le \ (\beta \to \top) \to \top$$

and these are not subtypes: the first $\bot$ on the left and the first $\top$ on the right are in the wrong inclusion relation ($\top \le \bot$), being in antimonotonic position.

The problem with rule (1) comes from the *negative* occurrences (on the left of an odd number of $\to$'s) of the recursion variable. In fact rule (1) is sound for types that are monotonic in the recursion variable.

A correct (and finitary) rule for inclusion of recursive types is instead the following:

$$(s \le t \ \Rightarrow \ \alpha \le \beta) \ \Rightarrow \ \mu s.\alpha \le \mu t.\beta \qquad (2)$$

where s occurs only in α, and t occurs only in β. That is, if by assuming the inclusion of the recursive variables we can verify the inclusion of the bodies, then we can deduce the inclusion of the recursive types. (It is interesting to check how subtyping now fails on the example above.)

Going back to the List example, if we have Nat≤Int and:

$$\text{NatList} \triangleq \mu s.\ \text{Unit} + (\text{Nat} \times s)$$
$$\text{IntList} \triangleq \mu t.\ \text{Unit} + (\text{Int} \times t)$$

then we can safely deduce NatList≤IntList from rule (2) since s≤t implies Unit+(Nat×s) ≤ Unit+(Int×t).

On the other hand, the Cell example does not work as smoothly.

$$\text{NatCell} \triangleq \mu s.\ (\text{Unit} \to \text{Nat}) \times (\text{Nat} \to s) \times (s \to s)$$
$$\text{IntCell} \triangleq \mu t.\ (\text{Unit} \to \text{Int}) \times (\text{Int} \to t) \times (t \to t)$$

Here we cannot conclude NatCell≤IntCell from rule (2), because of antimonotonicity: both the inclusion of the second component (*write*) and the inclusion of the third (*add*) fail. This is however not a deficiency of rule (2); such a conclusion would be unsound. For example, a NatCell might have a *write* function of type Nat→NatCell that fails on negative numbers. If such a cell were considered as an IntCell, it would be possible to pass a negative integer to this *write* and cause it to fail. These issues are related to the typechecking of object types in object-oriented languages, and are discussed at length in [15] and [8].

1.3 Equality of Recursive Types

We need now to consider strong notions of equality of recursive types. This is necessary because the rule (2) above is weak in some areas; for example, we cannot deduce directly from it that:

$$\mu t.t\to t \le \mu s.s\to s$$

because this would require assuming both $s\le t$ and $t\le s$. The combination of rule (2) and equality rules will finally give us all the power we need.

To check whether two recursive types $\mu s.\alpha'$ and $\mu t.\beta'$ are equivalent, we could assume $s=t$, and attempt to prove $\alpha'=\beta'$ under this assumption. This would work for $\mu t.\ t\to t$ and $\mu s.\ s\to s$. But now consider the types:

$$\alpha \triangleq \mu s.\mathrm{Int}\to s \qquad \beta \triangleq \mu t.\mathrm{Int}\to\mathrm{Int}\to t$$

They both expand infinitely into $\mathrm{Int}\to\mathrm{Int}\to\mathrm{Int}\to\mathrm{Int}\to...$, and they also have the same set of values (for example, recursive terms like $\mu f.\ \lambda x{:}\mathrm{Int}.\ f$). However, the assumption $s=t$ does not show $\mathrm{Int}\to s = \mathrm{Int}\to\mathrm{Int}\to t$; we get stuck on the question whether $s = \mathrm{Int}\to t$.

Another attempt might involve expanding the μ's, but unfortunately we cannot expand them out of existence. By unfolding alone we can get only:

$$\alpha = \mu s.\mathrm{Int}\to s = \mathrm{Int}\to(\mu s.\mathrm{Int}\to s) = \mathrm{Int}\to\mathrm{Int}\to(\mu s.\mathrm{Int}\to s) = \mathrm{Int}\to\mathrm{Int}\to\alpha$$
$$\beta = \mu t.\mathrm{Int}\to\mathrm{Int}\to t = \mathrm{Int}\to\mathrm{Int}\to(\mu t.\mathrm{Int}\to\mathrm{Int}\to t) = \mathrm{Int}\to\mathrm{Int}\to\beta$$

which after a few unfoldings leaves us with the original problem of determining whether $\alpha=\beta$. This is what we meant earlier by the insufficiency of "$\alpha \le \beta$ if for every expansion α^+ of α there is a β^+ of β with $\alpha^+ \le \beta^+$".

In fact, we seem to have made some progress here; we have come back to the original question $\alpha=\beta$ only after analyzing the entire structure of α and β. It seems that we should then be able to conclude that $\alpha=\beta$, because a complete analysis of α and β has found no contradiction. This kind of reasoning is possible but it has to be carefully justified, and in general we need to determine the conditions under which this stronger notion of equality does not lead to a circular argument.

Note that in the process above we have found a single context $C[X] \triangleq \mathrm{Int}\to\mathrm{Int}\to X$ such that $\alpha = C[\alpha]$ and $\beta = C[\beta]$; that is, both α and β are fixpoints of C. We shall be able to show that all the non-trivial (formally, *contractive*) type contexts C[X] have unique fixpoints over infinite trees, and therefore if they have two fixpoints these must be equal. Hence, the necessary rule for determining type equality can be formulated as follows:

$$\alpha = C[\alpha] \ \wedge\ \beta = C[\beta] \ \wedge\ C \text{ contractive} \quad\Rightarrow\quad \alpha = \beta \qquad (3)$$

It remains to be shown how to generate contractive contexts that allow us to equate any two types that have equal infinite expansions. This can be done via an algorithm, and in fact a natural one. We will show that this algorithm is sound (it will not equate types with different infinite expansions) and complete (it will equate all types that have equal infinite expansions). Such proofs of correctness of algorithms are among our major goals here, but first we need to carefully develop a formal framework.

1.4 Subtyping of Recursive Types

The problem of equating recursive types such as α and β above can be related to well-known solvable problems, such as the equivalence of finite-state automata. However, the similar problem for subtyping has no well-known parallel. Take, for example:

$$\gamma \triangleq \mu s.\mathrm{Int}\to s \qquad \delta \triangleq \mu t.\mathrm{Nat}\to\mathrm{Nat}\to t$$

Again, looking at the infinite expansions we obtain $\gamma = \mathrm{Int}\to\mathrm{Int}\to...$, and $\delta = \mathrm{Nat}\to\mathrm{Nat}\to...$, from which we would like to deduce $\gamma\le\delta$ by antimonotonicity. But what are the exact rules? Attempts to unfold γ and δ fall into the same difficulties as before.

The strategy here is to reduce the subtyping problem to an equality problem, which we solve by rule (3), plus rule (2). That is, we first show that $\delta' \triangleq \mu t.\mathrm{Nat}\to t = \mu t.\mathrm{Nat}\to\mathrm{Nat}\to t \equiv \delta$. After that, we can use rule (2) to show $\gamma\le\delta'$, and hence $\gamma\le\delta$.

Initially, this strategy suggests a two-step algorithm that first synchronizes the recursions in some appropriate way, and then uses rule (2) without additional folding/unfolding. Instead, we present an algorithm that tests subtyping of recursive types directly; the correspondence between the algorithm and the rules is then less obvious.

The example above involves two distinct recursive types for which the rule (2) alone is not sufficient to determine subtyping. This example may seem artificial, however this situation can easily happen in practice. As a slightly more plausible example, suppose we define the type of lists of alternating integers and naturals:

$$\mathrm{IntNatList} \triangleq \mu t.\mathrm{Unit}+\mathrm{Int}\times(\mathrm{Unit}+\mathrm{Nat}\times t)$$

This definition could arise more naturally from a mutual recursion construct in some programming language, for example:

$$\begin{aligned}&\text{Let Rec } \mathrm{IntNatList} = \mathrm{Unit}+\mathrm{Int}\times\mathrm{NatIntList}\\&\text{and}\quad \mathrm{NatIntList} = \mathrm{Unit}+\mathrm{Nat}\times\mathrm{IntNatList}\end{aligned}$$

One would certainly expect $\mathrm{NatList} \le \mathrm{IntNatList}$ to hold. But,

$$\mathrm{NatList} \triangleq \mu s.\mathrm{Unit}+\mathrm{Nat}\times s$$

hence we have first to show that $\mathrm{NatList} = \mu s.\mathrm{Unit}+\mathrm{Nat}\times(\mathrm{Unit}+\mathrm{Nat}\times s)$, and only then can we apply rule (2) successfully.

1.5 Algorithm outline

We describe the algorithm informally and we show some sample runs. This is only an approximation of the algorithm analyzed in the formal part, but it should explain the main ideas. A more detailed description is given in section 4.4.

A recursive type of the form $\mu t.$ ...t... can be represented in memory as a cyclic linked structure such that every occurrence of t in the recursive body is represented by the address of the corresponding μt structure, i.e., by a *back-pointer*. Otherwise, all subexpressions of a type expression, including μ subexpressions, are uniquely determined by their address in memory. Every time the algorithm reaches a μ structure, possibly through a back-pointer, it has the option of analyzing the interior of the structure ("unfolding" the recursive type) or to compare its

address with other addresses as a termination condition. The algorithm for $\alpha \leq \beta$ operates on a pair of linked structures and a *trail*. A trail is a set of address pairs that records the pairs of addresses that have been jointly encountered when following a pair of paths in the two linked structures. To avoid diverging on cyclic structures, the algorithms registers a local successful termination when it reaches a pair of addresses that have already been *seen*, that is, a pair of addresses that are contained in the trail.

The algorithm to determine whether $\alpha \leq \beta$ starts with an empty trail and proceeds through the following steps in sequence. We only consider basic types, function types, and recursive types.

[1] Succeed if the pair of addresses of α and β (in this order) is contained in the trail.
(In this situation, we have completely explored a pair of cyclic paths and found no subtyping failures; hence we declare success for these paths.)

[2] Succeed if α and β are type constants that are equal or in the subtype relation.
(This is the base case for the given collection of basic types and basic inclusions.)

[3] When α is $\alpha' \to \alpha''$ and β is $\beta' \to \beta''$, recur on $\beta' \leq \alpha'$ and on $\alpha'' \leq \beta''$. Succeed if both recursions succeed.
(This is the case for function types; note the swapping of inclusion on the domains because of antimonotonicity of $\to$; no such swapping would occur for data type constructors such as $\times$ and +).

[4.1] When α is $\mu t.\alpha'$ and β is $\mu s.\beta'$, add the pair of addresses of α and β (in this order) to the trail, and recur on $\alpha' \leq \beta'$. Succeed if the recursion succeeds.
(The presence of μ's signals potential cyclic paths, hence we store the current pair of addresses in the trail so that case [1] can prevent looping. We use an ordered pair of addresses because inclusion is, obviously, not symmetric; this detail differs from the standard trail algorithms for type equivalence. The next two cases are similar.)

[4.2] When α is $\mu t.\alpha'$, add the pair of addresses of α and β to the trail, and recur on $\alpha' \leq \beta$. Succeed if the recursion succeeds.

[4.3] When β is $\mu s.\beta'$, add the pair of addresses of α and β to the trail, and recur on $\alpha \leq \beta'$. Succeed if the recursion succeeds.

[5] Otherwise, fail. (This means we have found a pair of incomparable type expressions, such as a function type and a base type.)

A faithful description of a run of this algorithm would involve assigning arbitrary addresses to subexpressions of type expressions; this would only obscure the exposition. Instead, we display the type expressions and we leave their addresses implicit: the reader is urged to keep this in mind.

The diagrams below represent execution trees. The starting goal is at the bottom, the branching represents recursive calls, and the leaves represent termination conditions. The trail is shown in curly brackets; its elements are written as $t \leq s$, and represent pairs of addresses of type expressions. We indicate in square bracket the step of the algorithm used in each line to obtain the line above it.

The first sample run involves two types with matching μ structures; their inclusion is non-trivial because of antimonotonicity.

$\{t \le s\}\ t \le s$ [1] $\quad\quad$ $\{t \le s\}\ \bot \le t$ [2]

$\{t \le s\}\ s \to \bot \le t \to t$ [3] $\quad\quad$ $\{t \le s\}\ \bot \le \top$ [2]

$\{t \le s\}\ (t \to t) \to \bot \le (s \to \bot) \to \top$ [3]

$\{\}\ \mu t.((t \to t) \to \bot) \le \mu s.((s \to \bot) \to \top)$ [4.1]

The second sample run involves two types with mismatching μ structures. This mismatch introduces the need to examine a cyclic path more that once. For this, we use a *loopback* step, which corresponds to following a cyclic structure back to its original entry point (an artificial loopback step is needed only because, as we said, we keep the address information implicit). In the algorithm above, a loopback situation corresponds to a failure of step [1] followed by some dereferencing of back-pointers that leads to step [4].

$\{t \le s,\ t \le \bot \to s\}\ \bot \le \top$ [2] $\quad\quad$ $\{t \le s,\ t \le \bot \to s\}\ t \le s$ [1]

$\{t \le s,\ t \le \bot \to s\}\ \top \to t \le \bot \to s$ [3]

$\{t \le s\}\ \mu t.(\top \to t) \le \bot \to s$ [4.2]

$\{t \le s\}\ \bot \le \top$ [2] $\quad\quad$ $\{t \le s\}\ t \le \bot \to s$ [*loopback*]

$\{t \le s\}\ \top \to t \le \bot \to (\bot \to s)$ [3]

$\{\}\ \mu t.(\top \to t) \le \mu s.(\bot \to (\bot \to s))$ [4.1]

Hence, in this run we go around the μt loop twice in order to go around the μs loop once.

For other interesting examples, check how $\mu t.(t \to t) \le \mu s.(s \to s)$ succeeds, and how $\mu t.(t \to \bot) \le \mu s.(s \to \top)$ fails.

One of the main aims of this paper is to show that the algorithm above is consistent with, and in fact equivalent to, the rules (2) and (3) of sections 1.2 and 1.3. For this we need to place both the rules and the algorithm in a more formal framework.

1.6 Formal development

Having explained most of the problems and the unsatisfactory solutions arising from subtyping recursive types, we can now proceed to the formal treatment.

So far we have discussed rules for the subtyping of recursive types which are motivated by some operational intuition. In the following we will broaden our perspective and consider various notions of type equivalence, $\alpha = \beta$, and subtyping, $\alpha \le \beta$. These are induced by:

a) An *ordering on infinite trees*:	$\alpha =_T \beta$,	$\alpha \le_T \beta$	(Section 3)
b) An *algorithm*:	$\alpha =_A \beta$,	$\alpha \le_A \beta$	(Section 4)
c) A collection of *typing rules*:	$\alpha =_R \beta$,	$\alpha \le_R \beta$	(Section 5)
d) A collection of *per models*:	$\alpha =_M \beta$,	$\alpha \le_M \beta$	(Section 6)

The mathematical content of the paper consists mainly in analyzing the relationships between these notions. For a simply typed lambda calculus with recursive types (described in Section 2) we show, among other properties:

$$\alpha =_T \beta \iff \alpha =_A \beta \iff \alpha =_R \beta \implies \alpha =_M \beta$$
$$\alpha \le_T \beta \iff \alpha \le_A \beta \iff \alpha \le_R \beta \implies \alpha \le_M \beta$$

Moreover, we prove a restricted form of completeness with respect to the model (6.3), we show the definability in the calculus of certain maps that interpret coercions (7.1), and we give an

algorithm for computing the minimal type of a term with respect to $\leq_T$ (7.2). All these results support the relevance of the theory for the subtyping of recursive types sketched in this introduction.

2. A Simply Typed λ-calculus with Recursive Types

We consider a simply typed λ-calculus with recursive types and two ground types $\bot$ (bottom) and $\top$ (top); the latter play the roles of least and greatest elements in the subtype relation. Although this calculus is very simple, it already embodies the most interesting problems for which we can provide solutions sufficiently general to extend to other domains. In the conclusions we comment on which techniques can be applied to more complex calculi.[2]

2.1 Types

In an informal BNF notation, types are defined as follows:

t,s, ... type variables and type constants, indifferently
$\alpha ::= t \mid \bot \mid \top \mid \alpha\rightarrow\beta \mid \mu t.\alpha$

Types are identified up to renaming of bound variables. We use parentheses to determine precedence; in their absence, $\rightarrow$ associates to the right, and the scoping of μ extends to the right as far as possible. For simplicity we omit the other type constructors considered in the introduction.

2.2 Terms

Terms are denoted with M, N, ... ; the following rules establish when a term M has type α (written $M:\alpha$).

(assmp)	$x^{\alpha} : \alpha$
($\rightarrow$I)	$M : \beta \Rightarrow (\lambda x^{\alpha}.M) : \alpha\rightarrow\beta$
($\rightarrow$E)	$M : \alpha\rightarrow\beta,\ N : \alpha \Rightarrow (MN) : \beta$
(fold)	$M : [\mu t.\alpha/t]\alpha \Rightarrow (\mathrm{fold}_{\mu t.\alpha}\, M) : \mu t.\alpha$
(unfold)	$M : \mu t.\alpha \Rightarrow (\mathrm{unfold}_{\mu t.\alpha}\, M) : [\mu t.\alpha/t]\alpha$

Hence terms are either typed variables, typed λ-abstractions, applications, or fold and unfold coercions. The latter should be subscripted with the intended recursive type, to facilitate type inference, but these subscripts are sometime omitted. The fold/unfold coercions are technical devices to explicitly contract or expand the recursive type of a term; that is, such contractions and expansions do not happen automatically.

2.3 Equations

Here are some fundamental equations for the calculus. In particular, notice that the constants "fold" and "unfold" establish an isomorphism between a recursive type and its unfolding.

[2]Conventions: $\triangleq$ stands for equality by definition; $\equiv$for abbreviation or syntactic identification; $\vdash$ precedes a judgment provable in a certain formal system; $\supset$ is the linguistic implication; $\Rightarrow$ is the metalinguistic implication; [U/x]V denotes the substitution of U for x in V.

(β)	$(\lambda x^{\alpha}.M)N = [N/x^{\alpha}]M$	
(μ)	$\mathrm{fold}(\mathrm{unfold}\ x) = x$	$\mathrm{unfold}(\mathrm{fold}\ x) = x$

In section 6 we will consider a model in which many more types and terms are equated, for example the following will be valid equations:

(fold-unfold)	$[\mu t.\alpha/t]\alpha = \mu t.\alpha$	
(η)	$\lambda x^{\alpha}.Mx^{\alpha} = M$	if $x^{\alpha} \notin FV(M)$
($\bot$)	$x^{\bot} = y^{\bot}$	
($\top$)	$x^{\top} = y^{\top}$	

3. Tree Ordering

There is a well-established theory of subtyping for the non-recursive types. Basic motivations can be found, for example, in [11]. The notion of non-recursive type is merely syntactic; it means that the type does not contain μ's. The purpose of this section is to extend this theory to the recursive types, by defining a notion of approximation on infinite trees.

3.1 Subtyping Non-recursive Types

We have the following simple rules. There is a least type $\bot$ and a greatest type $\top$; the operator $\rightarrow$ is antimonotonic in the first argument and monotonic in the second. The relation $\leq$ is reflexive by virtue of (var) and ($\rightarrow$) below.

($\bot$)	$\bot \leq \alpha$
($\top$)	$\alpha \leq \top$
(var)	$t \leq t$
($\rightarrow$)	$\alpha' \leq \alpha,\ \beta \leq \beta' \ \Rightarrow\ \alpha \rightarrow \beta \leq \alpha' \rightarrow \beta'$

It is fairly easy to prove that the relation $\preccurlyeq$, defined as $\alpha \preccurlyeq \beta$ iff $\alpha \leq \beta$ is derivable in the system above, is a partial order on the collection of non-recursive types. In particular, one has to show that the transitivity rule:

(trans) $\quad \alpha \preccurlyeq \beta,\ \beta \preccurlyeq \gamma \ \Rightarrow\ \alpha \preccurlyeq \gamma$

is derived. This can be proven by defining a collection of rewriting rules on proofs that have the property that, when applied to a proof using transitivity, produce a (trans)-free proof of the same judgment. More abstractly one can look at the rules as the clauses of an inductive definition of a binary relation $\preccurlyeq$ and show that such a relation is transitive (see 3.4.4).

3.2 Folding and Unfolding

Should the types $[\mu t.\alpha/t]\alpha$ and $\mu t.\alpha$ be considered as equivalent? In general they are provably isomorphic in the calculus via fold and unfold. However, in most languages fold and unfold are implicit, and most implementations do not generate run-time code for them. So it seems reasonable to require that $[\mu t.\alpha/t]\alpha \leq \mu t.\alpha$ and $\mu t.\alpha \leq [\mu t.\alpha/t]\alpha$, thereby making unfolding transparent.

In fact, we will exhibit a model of the calculus in which $\mu t.\alpha$ and $[\mu t.\alpha/t]\alpha$ are equated because recursive domain equations are solved up to equality. However, a theory of type equivalence based only on the congruence closure of:

(fold-unfold) $\quad [\mu t.\alpha/t]\alpha \le \mu t.\alpha \quad \mu t.\alpha \le [\mu t.\alpha/t]\alpha$

turns out to be too weak; for example, the types $\mu t.s\to s\to t$ and $\mu t.s\to t$ are not equivalent.

Once we assume the transparency of unfolding, it seems natural to consider types with the same infinite expansions as equivalent. Infinite expansion can be rephrased as an *approximation property* such that the semantics of a type is completely determined by the semantics of its finite syntactic approximations. In fact, this is a very desirable property in the semantics of programming languages (see, for example, the approximation theorem in [29]).

3.3 Tree Expansions

As we have seen, simple unfolding does not induce a sufficiently strong notion of type equivalence. A stronger condition of approximation seems required to deal with infinite expansions. Let us first explain how to associate a finitely branching, labeled, regular tree with any recursive type.

Paths in a tree are represented by finite sequences of natural numbers $\pi,\sigma\in\omega^*$, with $\pi\sigma$ for concatenation and *nil* as the empty sequence.

Nodes in a tree are labeled by a ranked alphabet $L = \{\bot^0, \top^0, \to^2\} \cup \{t^0 \mid t \text{ is a type variable}\}$, where the superscripts indicate arity.

A tree $A\in\omega^*\rightharpoonup L$ is a partial function from (paths) ω^* into (node labels) L, whose domain is non-empty and prefix-closed, and such that each node has a number of children equal to the rank of the associated label.

Formally, let $A(\pi)\downarrow$ indicate that π is in the domain of A (and $A(\pi)\uparrow$ indicate the opposite). Then the collection Tree(L) of finitely-branching labeled trees over L, is given by the partial maps:

A: $\omega^*\rightharpoonup L$ such that:
$A(nil)\downarrow$
$A(\pi\sigma)\downarrow \;\Rightarrow\; A(\pi)\downarrow$
$A(\pi) = p^i \;\Rightarrow\; \forall 0\le j<i.\ A(\pi j)\downarrow$

We can now define a function T: Type $\to$ Tree(L) from recursive types (as defined in 2.1) to Tree(L). Let $h\mu$ be the function that counts the number of μ's in the head position of a type. We define $T(\alpha)(\pi)$ by induction on $(|\pi|, h\mu(\alpha))$:

$T(\bot)(nil) \triangleq \bot \qquad T(\top)(nil) \triangleq \top \qquad T(t)(nil) \triangleq t$

$T(\alpha\to\beta)(nil) \triangleq \to \qquad T(\alpha\to\beta)(0\pi) \triangleq T(\alpha)(\pi) \qquad T(\alpha\to\beta)(1\pi) \triangleq T(\beta)(\pi)$

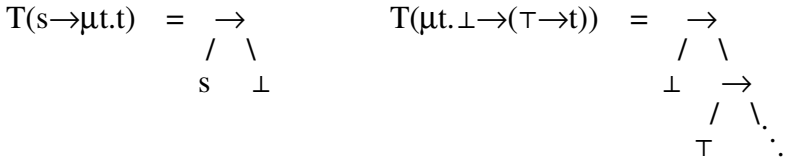
$T(\mu t.\alpha)(nil) \triangleq \bot$ if α has the shape $\mu t_1....\mu t_n.t$ $(t_i\ne t, i\in 1..n, n\ge 0)$

$T(\mu t.\alpha)(\pi) \triangleq T([\mu t.\alpha/t]\alpha)(\pi)$ if α does not have the shape above

$T(\alpha)(\pi) \triangleq \uparrow$ in all other cases

Note that the $\alpha\to\beta$ case reduces $|\pi|$, and second $\mu t.\alpha$ case preserves $|\pi|$ while reducing $h\mu(\alpha)$; this entails that the definition is well-founded.

Here are some simple examples; the tree on the right repeats itself after the "..." :



Finally, define the collection of finite trees, $\mathrm{Tree}_{fin}(L)$, as follows:

$$\mathrm{Tree}_{fin}(L) \triangleq \{A \in \mathrm{Tree}(L) \mid \exists k.\ \forall \pi \in \omega^*.\ |\pi| > k \Rightarrow A(\pi)\uparrow\}$$

Remarks

3.3.1 T induces a bijection between $\mathrm{Tree}_{fin}(L)$ and non-recursive types. We denote its inverse with T^{-1}.

3.3.2 Tree(L) is a complete metric space with respect to the usual metric on trees [4]. In fact it is the completion of the space of finite trees $\mathrm{Tree}_{fin}(L)$. We recall:

- A metric space is *complete* iff every Cauchy sequence converges.
- A map f:M→M over a metric space M with distance d is *contractive* iff there is a real number q<1 such that $\forall a,b \in M$: $d(f(a),f(b)) \le q \cdot d(a,b)$.
- Banach's fixpoint theorem asserts that a contractive map over a complete metric space has a unique fixpoint.
- The distance d(A,B) on Tree(L) is defined as either 0 if A=B; or else $2^{-c(A,B)}$, where c(A,B) is either ∞ if A=B, or else it is the length of a shortest path that distinguishes A from B.

3.3.3 For every α, Tα is a regular tree, that is, a tree with a finite number of different subtrees. Every tree is completely specified by the language of its occurrences, where if p∈L and A∈ Tree(L) then the occurrences are $\mathrm{Occ}(p,A) \triangleq \{\pi \in \omega^* \mid A(\pi) = p\}$. In particular, every regular tree A has an associated set $\{\pi p \mid \pi \in \mathrm{Occ}(p,A), p \in L\}$ which is a regular language [16].

From this it follows that given types α, β, the problem of deciding if Tα = Tβ is reducible to the problem of the equivalence of deterministic finite-state automata.

3.3.4 Going back to the example in 3.2, observe that T(μt.s→s→t) = T(μt.s→t).

3.4 Finite Approximations

Finite trees are in one-one correspondence with the non-recursive types, therefore they have a partial order as defined in 3.1. The problem we are going to consider now is how to extend this partial order on finite trees to Tree(L).

Hence, we introduce the notion of finite approximation of a tree. It is crucial to keep in mind the antimonotonic behavior of the → in its first argument.

We define a family of functions:

$$\{\ |_k : \mathrm{Tree}(L) \to \mathrm{Tree}_{fin}(L)\}_{k \in \omega}$$

Given A∈Tree(L) its *cut at the k-th level* is defined as follows:

$$A_{|k}(\pi) \triangleq \begin{cases} \uparrow & \text{if } |\pi| > k \\ A(\pi) & \text{if } |\pi| < k, \text{ or } |\pi| = k \text{ and } A(\pi)\uparrow \\ \bot & \text{if } |\pi| = k,\ A(\pi)\downarrow, \text{ and } \pi \text{ is positive in A} \\ \top & \text{if } |\pi| = k,\ A(\pi)\downarrow, \text{ and } \pi \text{ is negative in A} \end{cases}$$

where we say that π is positive (negative) in A if along the path π from the root we select the left sibling of a node labeled $\rightarrow$ an even (odd) number of times.

We can extend this definition to types:

$\alpha_{|k} \triangleq T^{-1}((T\alpha)_{|k})$ (a non-recursive type)

Convention

The bijection T,T^{-1} between $Tree_{fin}(L)$ and non-recursive types is from now on often omitted. That is, given any finite tree $A \in Tree_{fin}(L)$, we ambiguously identify it with the corresponding non-recursive type. Similarly, for $A \in Tree(L)$, we denote with $A_{|k}$ both its cut and the corresponding non-recursive type.

We are now ready to introduce a notion of tree ordering.

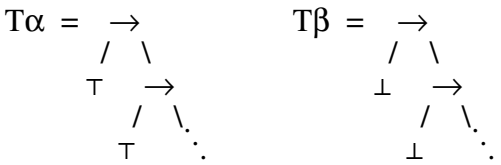
3.4.1 Definition *(tree ordering)*

For $A,B \in Tree_{fin}(L)$: $A \leq_{fin} B \Leftrightarrow T^{-1}A \leq T^{-1}B$ (as finite types; see 3.1)
For $A,B \in Tree(L)$: $A \leq_{\infty} B \Leftrightarrow \forall k.\ (A_{|k} \leq_{fin} B_{|k})$
For $\alpha,\beta \in Type$: $\alpha \leq_T \beta \Leftrightarrow T\alpha \leq_{\infty} T\beta$

Remarks

3.4.2 $\leq_{\infty}$ is a partial order on Tree(L).

3.4.3 $\alpha \leq_T \beta$ is a preorder on recursive types, and is such that for all k $\alpha_{|k} \leq_T \alpha$. We can now show, for example, $\alpha \triangleq \mu t.\top \rightarrow t \leq_T \mu t.\bot \rightarrow (\bot \rightarrow t) \triangleq \beta$; consider the tree expansions:



Observe that $\top$ and $\bot$ always occur in negative position so from $\bot \leq \top$ we can conclude $\forall k.\ \alpha_{|k} \leq \beta_{|k}$ and this gives us the statement.

3.4.4 One can think of other tree orderings; for example, consider the following inductive definition that gives an ordering $\leq_{Ind}$ on Tree(L).

$\leq_{Ind}$ is the least *reflexive* relation such that, $\forall A,B,A',B' \in Tree(F)$:

$\bot \leq_{Ind} A$; $A \leq_{Ind} \top$; $A' \leq_{Ind} A$, $B \leq_{Ind} B' \Rightarrow$ $\overset{\rightarrow}{/\ \backslash}$ (A B) $\leq_{Ind}$ $\overset{\rightarrow}{/\ \backslash}$ (A' B')

Equivalently, $\leq_{Ind} = \bigcup_{n<\omega} \leq^n$ where:

$$\leq^0 \quad = \quad \{(\bot,A), (A,\top) \mid A \in Tree(L)\} \cup Id_{Tree(L)}$$

$$\leq^{n+1} \quad = \quad \leq^n \cup \{(\underset{A}{/} \overset{\rightarrow}{\underset{B}{\backslash}} , \underset{A'}{/} \overset{\rightarrow}{\underset{B'}{\backslash}}) \mid A' \leq^n A, B \leq^n B'\}$$

It is not difficult to prove by induction on n that $\leq_{Ind}$ is a partial order on Tree(L), it conservatively extends the ordering on $Tree_{fin}(L)$ and it is contained in $\leq_\infty$. Moreover, such containment is strict as shown by the example in 3.4.3. In fact, $\leq_{Ind}$ lacks the crucial approximation property possessed by $\leq_\infty$.

4. An Algorithm

In this section we show that the tree ordering we have defined on types (3.4.1) can be decided by a rather natural modification of the algorithm that tests directly (that is, without reduction to a minimal form) the tree equivalence of two types.

4.1 Canonical Forms

The first step towards formalizing the algorithm is to introduce canonical forms for types and systems of equations.

Canonical forms of types allow us to ignore the trivial type equivalences due to redundant uses of μ binders. For example, the recursive type $(\mu t.\mu s.t \to s) \to ((\mu t.t) \to (\mu t.\top))$ can be simplified to the canonical form $(\mu v.v \to v) \to (\bot \to \top)$ without changing the denoted tree. In a canonical form, the body of each μ is an $\to$ type, and each μ variable is used in its μ body. Note, however, that different canonical forms may generate the same tree, for example $\mu t.s \to t$, $s \to \mu t.s \to t$, and $\mu t.s \to s \to t$.

Implementations of the subtyping algorithm manipulate cyclic linked data structures in computer memory. We represent these data structures abstractly as special sets of equations. Informally, each equation relates a memory address, represented by a variable, to a node of the data structure, represented by a type constant or a type constructor applied to variables. For example, here is a simple type with a corresponding equational representation and a possible memory representation:

Type	Equations	A memory representation			
	(v_0 is the *root*)	Addr.	Node	Child1	Child2
$\mu t.\bot \to t$	$v_0 = v_1 \to v_0$	0:	$\to$	1	0
	$v_1 = \bot$	1:	$\bot$	-	-

Sets of equations in this stylized form are called canonical. In this section we show that a canonical set of equations, along with a root variable, determines a unique tree which is called the *solution* of the equations. Moreover, we give effective ways of going from a type to a canonical set of equations, and vice versa, while preserving the represented tree.

Proviso

In order to have a simple correspondence between recursive types and systems of regular equations, we assume that all variables, both bound and free, in the types $\alpha_1, ..., \alpha_n$ under consideration are distinct. When a type is unfolded, the necessary renaming of bound variables must be performed. For example, $(\mu t.t\to s)\to(\mu s.t\to s)$ should be rewritten as $(\mu v.v\to s)\to(\mu r.t\to r)$.

4.1.1 Recursive Types in Canonical Form

Henceforth, Tp denotes the collection of non-recursive types, and μTp denotes the collection of *recursive types in canonical form*, defined as follows:

$$\alpha \ ::= \ \bot \mid \top \mid t \mid \alpha\to\beta \mid \mu t.\alpha\to\beta$$

where in the case $\mu t.\alpha\to\beta$, t must occur free in $\alpha\to\beta$. Hence the body of a μ in canonical form must immediately start with an →; in particular, it cannot be another μ. The introduction of μTp simplifies the case analysis in the following proofs.

4.1.2 Proposition *(existence of canonical forms)*

For every type α there is a type β in canonical form such that $T\alpha=T\beta$.

Proof

The crucial observation is that $T\mu t.\mu s.\gamma[t,s] = T\mu v.\gamma[v,v]$. See also 5.1.3 for a proof of this fact that uses the rules for type equivalence. □

4.1.3 Regular System of Equations in Canonical Form

Systems of regular equations are a well-known tool for representing regular trees (see for example [16], [17]).

For our purposes a *regular system of equations in canonical form* is an element of Tenv, that is, a finite association of distinct type variables (members of Tvar) with types in a specific form:

$$\begin{aligned}&Tenv \triangleq\\ &\quad \{\ \varepsilon \in Tvar \rightharpoonup Tp \mid Dom(\varepsilon) \text{ is finite and } \forall t\in Dom(\varepsilon) \text{ we have that}\\ &\quad\ \varepsilon(t) \text{ is one of } \bot, \top, t_1, t_2\to t_3, \text{ where } t_1\notin Dom(\varepsilon) \text{ and } t_2, t_3\in Dom(\varepsilon)\ \}\end{aligned}$$

A pair $(\alpha, \varepsilon) \in Tp \times Tenv$ represents the following *system of regular equations* (not necessarily in canonical form because α may be complex):

$$\begin{aligned}t_\alpha &= \alpha \qquad (t_\alpha \text{ a fresh variable})\\ t &= \varepsilon(t) \qquad \text{for each } t\in Dom(\varepsilon)\end{aligned}$$

It is important to observe that, by the definition of Tenv, this system defines a *contractive functional* $(G_0,..., G_n)$ over $Tree(L)^{n+1}$ (see remark 3.3.2) where $n = |Dom(\varepsilon)|$, $Dom(\varepsilon) = \{t_1, ..., t_n\}$ and:

$$\begin{aligned}G_0(A_0, ..., A_n) &\triangleq [A_0/t_\alpha, A_1/t_1, ..., A_n/t_n]T\alpha\\ G_i(A_0, ..., A_n) &\triangleq [A_0/t_\alpha, A_1/t_1, ..., A_n/s_n]T\varepsilon(t_i) \quad (1\le i\le n)\end{aligned}$$

The predicate $Reach(\alpha,\varepsilon)$ denotes the variables reachable from the free variables in α by applying the equations in ε. Formally:

$\mathrm{Reach}(t,\varepsilon)^0 \triangleq \{t\}$

$\mathrm{Reach}(t,\varepsilon)^{n+1} \triangleq$ if $t \notin \mathrm{Dom}(\varepsilon)$ then $\{t\}$

if $t \in \mathrm{Dom}(\varepsilon)$ then

if $\varepsilon(t)=\bot$ or $\varepsilon(t)=\top$ then $\emptyset$

if $\varepsilon(t)=s$ then $\{s\}$

if $\varepsilon(t)=t_1 \to t_2$ then $\mathrm{Reach}(t_1,\varepsilon)^n \cup \mathrm{Reach}(t_2,\varepsilon)^n$

$\mathrm{Reach}(t,\varepsilon) \triangleq \bigcup_{n\in\omega} \mathrm{Reach}(t,\varepsilon)^n$

$\mathrm{Reach}(\alpha,\varepsilon) \triangleq \bigcup_{t\in \mathrm{FV}(\alpha)} \mathrm{Reach}(t,\varepsilon)$

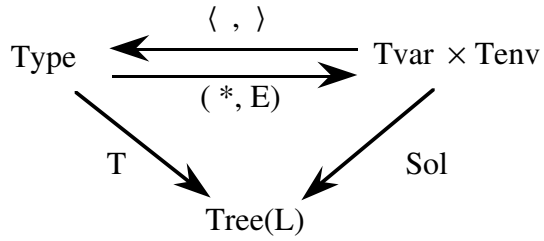
4.1.4 Definition *(solution of a system)*

We denote with *Sol(α, ε)* the first component B_0 of the solution $(B_0,..., B_n)$ in $\mathrm{Tree}(L)^{n+1}$ of the system associated with (α, ε). The solution is given by Banach's unique fixpoint theorem (see remark 3.3.2).

Remark

Given a system of regular equations in canonical form, it is possible to minimize the number of variables by a procedure that is analogous to the one for minimizing the number of states in a deterministic finite-state automaton. This immediately provides an algorithm for deciding the equality of the trees represented by two regular systems of equations in canonical form.

In the rest of this section we describe maps between types and regular systems of equations in canonical form, as summarized by the following diagram, where all the paths leading from a node to Tree(L) commute.



4.1.5 Proposition *(From recursive types to regular systems)*

There is a pair of maps $*\in \mathrm{Type}\to\mathrm{Tvar}$, $E\in \mathrm{Type}\to\mathrm{Tenv}$ such that:

$\forall\alpha\in \mathrm{Type}.\ T\alpha = \mathrm{Sol}(\alpha^*, E\alpha)$

Proof

It is enough to prove the result for every term in $\mu\mathrm{Tp}$. Then the lemma follows by 4.1.2. We now define (*, E) by induction on the structure of $\gamma\in\mu\mathrm{Tp}$.

Cases $\gamma\equiv t$, $\gamma\equiv\bot$, and $\gamma\equiv\top$. Take $\gamma^*\triangleq s$ and $E\gamma\triangleq\{s=\gamma\}$, for any s not appearing in the original type α.

Case $\gamma \equiv \alpha \to \beta$.

We denote with $p[t_1...t_k]$ for $p \in L \setminus \{t_1...t_k\}$ a type in Tp of the form $p(u_1..u_{\#p})$, where #p is the arity of p, and $\{u_1..u_{\#p}\} \subseteq \{t_1...t_k\}$.

Assume, by induction hypothesis, that $E\alpha = \{t_{i+1} = p_i[t_2...t_{n+1}] \mid i \in 1..n\}$, $\alpha^* = t_2$, $E\beta = \{t_{n+1+j} = q_j[t_{n+2}...t_{n+m+1}] \mid j \in 1..m\}$, and $\beta^* = t_{n+2}$. (We require here that t_1, $t_2...t_{n+1}$, and $t_{n+2}...t_{n+m+1}$ are disjoint variables not appearing in the original α; otherwise a consistent renaming must be performed.) Then $E\gamma$ is the following system and $\gamma^* \triangleq t_1$:

$t_1 = r_1[t_1...t_{n+m+1}] \equiv t_2 \to t_{n+2}$,
$t_2 = r_2[t_1...t_{n+m+1}] \equiv p_1[t_2...t_{n+1}]$ ...
$t_{n+1} = r_{n+1}[t_1...t_{n+m+1}] \equiv p_n[t_2...t_{n+1}]$
$t_{n+2} = r_{n+2}[t_1...t_{n+m+1}] \equiv q_1[t_{n+2}...t_{n+m+1}]$ ...
$t_{n+m+1} = r_{n+m+1}[t_1...t_{n+m+1}] \equiv q_m[t_{n+2}...t_{n+m+1}]$

The property $Sol(\gamma^*, E\gamma) = Sol(t_1, E\gamma) = Sol(t_2 \to t_{n+2}, E\gamma) = Sol(t_2, E\alpha) \to Sol(t_{n+2}, E\beta) = T\gamma$ follows easily from the induction hypothesis.

Case $\gamma \equiv \mu t.\alpha \to \beta$.

Let $\gamma' = [\gamma/t]\alpha \to [\gamma/t]\beta$ (of course $T\gamma' = T\gamma$). As in the previous case, assume $E\alpha = \{t_{i+1} = p_i[t_2...t_{n+1}] \mid i \in 1..n\}$, $\alpha^* = t_2$, $E\beta = \{t_{n+1+j} = q_j[t_{n+2}...t_{n+m+1}] \mid j \in 1..m\}$ and $\beta^* = t_{n+2}$. Then $E\gamma$ is the following system and $\gamma^* \triangleq t_1$:

$t_1 = r_1[t_1...t_{n+m+1}] \equiv t_2 \to t_{n+2}$
$t_2 = r_2[t_1...t_{n+m+1}] \equiv t_2 \to t_{n+2} | p_1[t_2...t_{n+1}]$ ...
$t_{n+1} = r_{n+1}[t_1...t_{n+m+1}] \equiv t_2 \to t_{n+2} | p_n[t_2...t_{n+1}]$
$t_{n+2} = r_{n+2}[t_1...t_{n+m+1}] \equiv t_2 \to t_{n+2} | q_1[t_{n+2}...t_{n+m+1}]$ ...
$t_{n+m+1} = r_{n+m+1}[t_1...t_{n+m+1}] \equiv t_2 \to t_{n+2} | q_m[t_{n+2}...t_{n+m+1}]$

By $t_2 \to t_{n+2} | p_i[t_2...t_{n+1}]$ we denote $t_2 \to t_{n+2}$ if $p_i \equiv t$, and $p_i[t_2...t_{n+1}]$ otherwise. Analogously, $t_2 \to t_{n+2} | q_j[t_{n+2}...t_{n+m+1}]$ denotes $t_2 \to t_{n+2}$ if $q_j \equiv t$, and $q_j[t_{n+2}...t_{n+m+1}]$ otherwise. Next proceed by induction on $(|\pi|, \gamma)$ to prove $T\gamma(\pi) = Sol(\gamma^*, E\gamma)(\pi)$. The only difficulty arises for $\gamma \equiv \mu t.\alpha \to \beta$. In order to apply the induction hypothesis one needs a lemma. Following the notation above we show, for instance, $Sol(t_2, E\gamma) = Sol([\gamma/t]\alpha^*, E[\gamma/t]\alpha)$. See 4.1.7 for a proof of a related fact that gives the main insights while being slightly simpler. □

Here is an example of the procedure described in the proof above. Consider:

$\gamma \triangleq \mu t.t \to \bot$.

For the base cases t and $\bot$ we have (cunningly choosing the names t_2 and t_3):

$t^* = t_2$; $Et = \{t_2 = t\}$
$\bot^* = t_3$; $E\bot = \{t_3 = \bot\}$

From the μ case of the proof we obtain:

$\gamma^* = t_1$; $E\gamma = \{t_1 = t_2 \to t_3,\ t_2 = t_2 \to t_3,\ t_3 = \bot\}$

Note the first two equations of the system $E\gamma$; the redundancy facilitates the uniform treatment of the μ case.

4.1.6 Definition *(From regular systems to recursive types)*

We define a function $\langle - , - \rangle : Tp \times Tenv \to Type$ by induction on $(|Dom(\varepsilon)|, \alpha)$:

$\langle \bot,\varepsilon\rangle \quad \triangleq \quad \bot$
$\langle \top,\varepsilon\rangle \quad \triangleq \quad \top$
$\langle \alpha\to\beta,\varepsilon\rangle \quad \triangleq \quad \langle \alpha,\varepsilon\rangle \to \langle \beta,\varepsilon\rangle$
$\langle t,\varepsilon\rangle \quad \triangleq \quad t \qquad$ if $t\notin Dom(\varepsilon)$
$\langle t,\varepsilon\rangle \quad \triangleq \quad \mu t.\langle \varepsilon(t),\varepsilon\backslash t\rangle \qquad$ if $t\in Dom(\varepsilon)$

where $\varepsilon\backslash t$ is like ε except that it is undefined on t.

Continuing the example above, we have:

$\langle \gamma^*, E\gamma\rangle = \langle t_1, \{t_1 = t_2\to t_3,\ t_2 = t_2\to t_3,\ t_3 = \bot\}\rangle$
$= \mu t_1.\langle t_2\to t_3, \{t_2 = t_2\to t_3,\ t_3 = \bot\}\rangle$
$= \mu t_1.\langle t_2, \{t_2 = t_2\to t_3,\ t_3 = \bot\}\rangle \to \langle t_3, \{t_2 = t_2\to t_3,\ t_3 = \bot\}\rangle$
$= \mu t_1.(\mu t_2.\langle t_2\to t_3, \{t_3 = \bot\}\rangle) \to (\mu t_3.\langle \bot, \{t_2 = t_2\to t_3\}\rangle)$
$= \mu t_1.(\mu t_2.\langle t_2, \{t_3 = \bot\}\rangle \to \langle t_3, \{t_3 = \bot\}\rangle) \to (\mu t_3.\bot)$
$= \mu t_1.(\mu t_2.t_2\to(\mu t_3.\langle \bot, \{\ \}\rangle)) \to (\mu t_3.\bot)$
$= \mu t_1.(\mu t_2.t_2\to(\mu t_3.\bot)) \to (\mu t_3.\bot)$

The last line is equivalent to the original type $\gamma = \mu t.t\to\bot$, as established in general by the following proposition.

4.1.7 Proposition *(More on commuting translations)*

(1) For any system of equations, the first component of the solution coincides with the tree expansion of the associated recursive type:

$\forall(\alpha, \varepsilon) \in Tp \times Tenv.\ \ Sol(\alpha, \varepsilon) = T\langle \alpha,\varepsilon\rangle$ (we abbreviate $T(\langle \alpha,\varepsilon\rangle)$ as $T\langle \alpha,\varepsilon\rangle$)

(2) The map $\langle\ ,\ \rangle$ satisfies the conditions:

1. $\langle \bot,\varepsilon\rangle = \bot$
2. $\langle \top,\varepsilon\rangle = \top$
3. $\langle t,\varepsilon\rangle = t \qquad$ if $t\notin Dom(\varepsilon)$
4. $T\langle t,\varepsilon\rangle = T\langle \varepsilon(t),\varepsilon\rangle \qquad$ if $t\in Dom(\varepsilon)$
5. $T\langle \alpha\to\beta,\varepsilon\rangle = T(\langle \alpha,\varepsilon\rangle \to \langle \beta,\varepsilon\rangle)$
6. $T\langle \alpha^*, E\alpha\rangle = T\alpha$

Proof

(1) Show by induction on $(|\pi|, \alpha)$ that $T\langle \alpha,\varepsilon\rangle(\pi) = Sol(\alpha, \varepsilon)(\pi)$.
The interesting case arises when $\alpha\equiv t$, $t\in Dom(\varepsilon)$, $\varepsilon(t) = t_1\to t_2$.
Then, $Sol(t, \varepsilon) = Sol(\varepsilon(t), \varepsilon) = Sol(t_1\to t_2,\varepsilon)$;
and, $T\langle t,\varepsilon\rangle = T\mu t.\langle \varepsilon(t), \varepsilon\backslash t\rangle = T[\langle t,\varepsilon\rangle/t]\langle t_1\to t_2, \varepsilon\backslash t\rangle$.
In order to apply the induction hypothesis and complete this case
one needs to prove $T\langle t_i, \varepsilon\rangle = T([\langle t,\varepsilon\rangle/t]\langle t_i, \varepsilon\backslash t\rangle)$ (i=1,2).
To obtain the latter, we show the following lemma:

For any canonical system ε, and type variables, t, t',
we have $T\langle t', \varepsilon\rangle = T([\langle t,\varepsilon\rangle/t]\langle t', \varepsilon\backslash t\rangle)$.

We proceed by induction on the depth of the path π, and by case analysis, to show:

$T\langle t', \varepsilon\rangle(\pi) = T([\langle t,\varepsilon\rangle/t]\langle t', \varepsilon\backslash t\rangle)(\pi)$.

Case $t\equiv t'$: $[\langle t,\varepsilon\rangle/t]\langle t', \varepsilon\backslash t\rangle = [\langle t,\varepsilon\rangle/t]\ t' = \langle t,\varepsilon\rangle = \langle t',\varepsilon\rangle$.

Case $t \neq t'$:
Subcase $t \in Dom(\varepsilon)$, $t' \in Dom(\varepsilon)$:

Say: $\varepsilon(t) = t_1 \rightarrow t_2$, $\varepsilon(t') = t'_1 \rightarrow t'_2$.

Then: $\alpha \equiv \langle t', \varepsilon \rangle = \mu t'. \langle t'_1, \varepsilon \backslash t' \rangle \rightarrow \langle t'_2, \varepsilon \backslash t' \rangle$.

and $T\alpha =$

$$\begin{array}{ccc} & \rightarrow & \\ & / \quad \backslash & \\ T([\langle t', \varepsilon \rangle / t'] \langle t'_1, \varepsilon \backslash t' \rangle & & T([\langle t', \varepsilon \rangle / t'] \langle t'_2, \varepsilon \backslash t' \rangle \end{array}$$

Also: $\beta \equiv [\langle t, \varepsilon \rangle / t] \langle t', \varepsilon \backslash t \rangle = [\langle t, \varepsilon \rangle / t] \mu t'. \langle t'_1, \varepsilon \backslash t \backslash t' \rangle \rightarrow \langle t'_2, \varepsilon \backslash t \backslash t' \rangle =$

$= [\langle t, \varepsilon \rangle / t] ([\langle t', \varepsilon \backslash t \rangle / t'](\langle t'_1, \varepsilon \backslash t \backslash t' \rangle \rightarrow \langle t'_2, \varepsilon \backslash t \backslash t' \rangle))$.

So: $T\beta =$

$$\begin{array}{ccc} & \rightarrow & \\ & / \quad \backslash & \\ T[\langle t, \varepsilon \rangle / t] ([\langle t', \varepsilon \backslash t \rangle / t'](\langle t'_1, \varepsilon \backslash t \backslash t' \rangle)) & & T[\langle t, \varepsilon \rangle / t] ([\langle t', \varepsilon \backslash t \rangle / t'](\langle t'_2, \varepsilon \backslash t \backslash t' \rangle)) \end{array}$$

If $n\pi$ is the current path, we can apply the inductive hypothesis on the shorter path π w.r.t.:

(i) the variables t', t'_i, $(i=1,2)$ and the system ε to show:
$T\langle t'_i, \varepsilon \rangle(\pi) = T([\langle t', \varepsilon \rangle / t'] \langle t'_i, \varepsilon \backslash t' \rangle(\pi)$.

(ii) the variables t', t'_i, $(i=1,2)$ and the system $\varepsilon \backslash t$ to show:
$T\langle t'_i, \varepsilon \backslash t \rangle(\pi) = T[\langle t', \varepsilon \backslash t \rangle / t'](\langle t'_i, \varepsilon \backslash t \backslash t' \rangle)(\pi)$.

(iii) the variables t, t'_i, $(i=1,2)$ and the system ε to show:
$T[\langle t, \varepsilon \rangle / t] \langle t'_i, \varepsilon \backslash t \rangle(\pi) = T\langle t'_i, \varepsilon \rangle(\pi)$.

Finally we use the substitutivity of the T operation, $T[\gamma/t]\delta = [T\gamma/t]T\delta$, to conclude $T\alpha = T\beta$.

Subcase $t \in Dom(\varepsilon)$, $t' \notin Dom(\varepsilon)$:

Say: $\varepsilon(t) = t_1 \rightarrow t_2$.

Then: $T\langle t', \varepsilon \rangle = t'$

$T([\langle t, \varepsilon \rangle / t] \langle t', \varepsilon \backslash t \rangle) = T([\langle t, \varepsilon \rangle / t]\ t' = t'$.

Subcase $t \notin Dom(\varepsilon)$, $t' \in Dom(\varepsilon)$:

Say: $\varepsilon(t') = t'_1 \rightarrow t'_2$.

Then: $T([\langle t, \varepsilon \rangle / t] \langle t', \varepsilon \backslash t \rangle) = T([t/t] \langle t', \varepsilon \backslash t \rangle) = T\langle t', \varepsilon \backslash t \rangle = T\langle t', \varepsilon \rangle$.

Subcase $t \notin Dom(\varepsilon)$, $t' \notin Dom(\varepsilon)$:

Then: $T([\langle t, \varepsilon \rangle / t] \langle t', \varepsilon \backslash t \rangle) = T\ [t/t]\ t' = Tt' = T\langle t', \varepsilon \rangle$

(2) Conditions 1, 2, 3, 5 follow by definition.
Condition 4 follows from $Sol(t, \varepsilon) = Sol(\varepsilon(t), \varepsilon)$ and part (1).
Condition 6 follows from prop. 4.1.5 and part (1): $T\alpha = Sol(\alpha^*, E\alpha) = T\langle \alpha^*, E\alpha \rangle$. □

4.2 Computational Rules

The subtyping algorithm described in this section is based on the canonical sets of equations described in the previous section (again, these equations can be interpreted as linked data structures in memory). The algorithm involves a single set of equations ε, with two distinct roots α and β representing the types to be compared. It also involves a *trail* Σ of the form $\{t_1 \leq s_1, ..., t_n \leq s_n\}$, which records inclusions of variables discovered as the algorithm progresses. An invocation of the algorithm with parameters Σ, ε, α and β, is written as the *judgment* $\Sigma, \varepsilon \supset \alpha \leq \beta$.

The algorithm is not expressed as an ordinary procedure, but as a collection of rules that resembles a Prolog program. The typical rule is written as a logical implication of judgments:

$$\Sigma_1,\varepsilon_1 \supset \alpha_1 \le \beta_1,\ \Sigma_2,\varepsilon_2 \supset \alpha_2 \le \beta_2 \ \Rightarrow\ \Sigma,\varepsilon \supset \alpha \le \beta$$

Operationally, this means that in order to determine whether $\Sigma,\varepsilon \supset \alpha \le \beta$ holds, we must invoke the "subroutines" $\Sigma_1,\varepsilon_1 \supset \alpha_1 \le \beta_1$ and $\Sigma_2,\varepsilon_2 \supset \alpha_2 \le \beta_2$ and check whether they hold. In general, given a logical deduction in this system of rules, the algorithm execution can be recovered by reading the rules backwards from the conclusion to the assumptions.

In the following, t,s,r,u denote arbitrary variables; a,b denote variables not in the domain of ε; Σ is a finite set of subtyping assumptions on pairs of type variables; and $\alpha,\beta \in Tp$.

The algorithm can then be written as follows:

$(assmp_A)$	$\Sigma,\varepsilon \supset t \le s$	if $t \le s \in \Sigma$
$(\bot_A)$	$\Sigma,\varepsilon \supset \bot \le \beta$	
$(\top_A)$	$\Sigma,\varepsilon \supset \alpha \le \top$	
(var_A)	$\Sigma,\varepsilon \supset a \le a$	
$(\to_A)$	$\Sigma,\varepsilon \supset \alpha' \le \alpha,\ \ \Sigma,\varepsilon \supset \beta \le \beta' \ \Rightarrow\ \Sigma,\varepsilon \supset \alpha \to \beta \le \alpha' \to \beta'$	
(μ_A)	$\Sigma \cup \{t \le s\},\varepsilon \supset \varepsilon(t) \le \varepsilon(s) \ \Rightarrow\ \Sigma,\varepsilon \supset t \le s$	if $t,s \in Dom(\varepsilon)$

The initial judgment $\Sigma,\varepsilon \supset \alpha \le \beta$ that starts an execution of the algorithm must obey a special condition expressing some reasonable assumptions. This condition says that the initial type structures α,β are simple root variables denoting disjoint structures, and that Σ has not yet come into play. For $\Sigma = \{t_1 \le s_1,\ldots, t_n \le s_n\}$, define:

$$Vars(\Sigma) \ \triangleq\ \{t_1,s_1,\ldots,t_n,s_n\}$$
$$\Sigma \# \varepsilon \ \Leftrightarrow\ Vars(\Sigma) \cap Dom(\varepsilon) = \emptyset$$

Then, a judgment $\Sigma,\varepsilon \supset \alpha \le \beta$ satisfies the *initiality condition* (or equivalently, is an *initial goal*) iff $\alpha \equiv t$, $\beta \equiv s$, ε can be decomposed in $\varepsilon_1 \cup \varepsilon_2$ so that $t \in Dom(\varepsilon_1)$ and $s \in Dom(\varepsilon_2)$, $Dom(\varepsilon_1) \cap Dom(\varepsilon_2) = \emptyset$, and $\Sigma \# \varepsilon$.

By the way canonical systems are constructed, and by the fact of starting with an initial goal, the expansions of variables according to ε, as in (μ_A), is always synchronized. That is, in a call to $\Sigma,\varepsilon \supset \alpha \le \beta$ during the execution of the algorithm we never have a situation where α is a variable in $Dom(\varepsilon)$ and β is not, or vice versa; hence (μ_A) covers all the cases that may arise. If one desires to treat more general systems of equations, then it may be necessary to introduce other μ-rules that take into account situations in which just an ε-expansion on the left (or the right) is needed. In these cases we would have rules like:

$(assmp'_A)$	$\Sigma,\varepsilon \supset \alpha \le \beta$	if $\alpha \le \beta \in \Sigma$	
(μ_{lA})	$\Sigma \cup \{t \le \alpha' \to \beta'\},\varepsilon \supset \varepsilon(t) \le \alpha' \to \beta' \ \Rightarrow\ \Sigma,\varepsilon \supset t \le \alpha' \to \beta'$		if $t \in Dom(\varepsilon)$
(μ_{rA})	$\Sigma \cup \{\alpha' \to \beta' \le s\},\varepsilon \supset \alpha' \to \beta' \le \varepsilon(s) \ \Rightarrow\ \Sigma,\varepsilon \supset \alpha' \to \beta' \le s$		if $s \in Dom(\varepsilon)$.

Note also that there are two conceptually distinct uses of the rule $(assmp_A)$ in the algorithm: one for the initial assumptions contained in Σ, which represent known inclusions on type constants, and one for the assumptions inserted during the computation, which come from the unfolding of μ's.

4.2.1 Generating the Execution Tree

Given a goal $\Sigma,\varepsilon \supset t \leq s$, the algorithm consists in applying the inference rules backwards, generating subgoals in the cases ($\rightarrow_A$) and (μ_A). This process is completely determined once we establish that ($assmp_A$) has priority over the other rules and ($\bot_A$) has priority over ($\top_A$).

A tree of goals built this way is called an *execution tree*. If no rules are applicable to a certain subgoal, that branch of the execution tree is abandoned, and execution is resumed at the next subgoal, until all subgoals are exhausted.

4.2.2 Termination

The execution tree is always *finite*. Observe that if $t \leq s$ is the assumption that we add to Σ, then t and s are type variables in $Dom(\varepsilon)$. Also observe that the ($\rightarrow$) rule shrinks the size of the current goal by replacing it with subexpressions of the goal, and that each application of a μ-rule enlarges Σ.

The bound on the depth of the execution tree for $\alpha \leq_A \beta$ is of the order of the product of the sizes of the two systems $E\alpha$, $E\beta$.

4.2.3 Algorithm Ordering

An execution tree *succeeds* if all the leaves correspond to an application of one of the rules ($assmp_A$), ($\bot_A$), ($\top_A$), and (var_A). Dually, it *fails* if at least one leaf is an unfulfilled goal (no rule can be applied).

We write $\vdash_A \Sigma,\varepsilon \supset t \leq s$ iff $\Sigma,\varepsilon \supset t \leq s$ is an initial goal (4.2) and the corresponding execution tree succeeds.

Given recursive types α,β we write:

$$\alpha \leq_A \beta \quad \Leftrightarrow \quad \vdash_A \emptyset, E\alpha \cup E\beta \supset \alpha^* \leq \beta^*$$

For testing type equality, we can define:

$$\alpha =_A \beta \quad \Leftrightarrow \quad \alpha \leq_A \beta \ \wedge \ \beta \leq_A \alpha$$

Alternatively, we could directly define a (more efficient) type equality algorithm, along the same lines as the subtyping algorithm.

4.3 Soundness and Completeness of the Algorithm

We now show that the subtyping algorithm described in the previous section is sound and complete with respect to the infinite-tree interpretation of types. That is, the algorithm precisely embodies our intuition of recursive types as infinite trees.

First we prove soundness and completeness for non-recursive types. Soundness is then derived by observing that a successful execution of the algorithm on some input must also be successful on all the finite approximations of the input. Completeness is proven by examining a failing execution tree, and concluding that the trees corresponding to the input must have been different to start with.

4.3.1 Lemma *(Derived structural computational rules)*

Given the definition of Tenv in 4.1.3, the algorithm in 4.2, and the ordering in 4.2.3, we have:

Σ-weaken

If $\vdash_A \Sigma,\varepsilon \supset t \leq s$ and $\Sigma \cup \Sigma' \# \varepsilon$ then $\vdash_A \Sigma \cup \Sigma',\varepsilon \supset t \leq s$.

Σ-strengthen

If $\vdash_A \Sigma \cup \Sigma',\varepsilon \supset t \leq s$ and $Reach(t \rightarrow s,\varepsilon) \cap Vars(\Sigma') = \emptyset$ then $\vdash_A \Sigma,\varepsilon \supset t \leq s$.

ε-weaken

If $\vdash_A \Sigma,\varepsilon \supset t \leq s$, $Reach(t \rightarrow s,\varepsilon) \cap Dom(\varepsilon') = \emptyset$, $\Sigma \# \varepsilon \cup \varepsilon'$, and $Dom(\varepsilon) \cap Dom(\varepsilon') = \emptyset$ then $\vdash_A \Sigma,\varepsilon \cup \varepsilon' \supset t \leq s$.

ε-strengthen

If $\vdash_A \Sigma,\varepsilon \cup \varepsilon' \supset t \leq s$ and $Reach(t \rightarrow s,\varepsilon) \cap Dom(\varepsilon') = \emptyset$ then $\vdash_A \Sigma,\varepsilon \supset t \leq s$.

4.3.2 Proposition *(Completeness of $\leq_A$ for non-recursive types)*

Given $\alpha,\beta \in Tp$ non-recursive types then $\alpha \leq_T \beta \Rightarrow \alpha \leq_A \beta$.

Proof

Let $\varepsilon \triangleq E\alpha \cup E\beta$. We show $\alpha \leq_T \beta \Rightarrow \forall \Sigma.\ \Sigma \# \varepsilon \Rightarrow \vdash_A \Sigma,\varepsilon \supset \alpha^* \leq \beta^*$ by induction on the structure of α and β.

Case $\alpha \equiv \bot$. Then $\varepsilon = \{\alpha^* = \bot\} \cup E\beta$. Take any Σ s.t. $\Sigma \# \varepsilon$:

$\Rightarrow \Sigma \cup \{\alpha^* \leq \beta^*\},\varepsilon \supset \varepsilon(\alpha^*) \leq \varepsilon(\beta^*)$ by $(\bot_A)$ since $\varepsilon(\alpha^*) = \bot$

$\Rightarrow \Sigma,\varepsilon \supset \alpha^* \leq \beta^*$ by (μ_A) since $\alpha^*,\beta^* \notin Vars(\Sigma)$

Cases $\alpha \equiv \top$, $\alpha \equiv a$. Similar.

Case $\alpha \equiv \alpha' \rightarrow \alpha''$. Since $\alpha \leq_T \beta$, we have either:

Case $\beta \equiv \top$, similar to the case $\alpha \equiv \bot$.

Case $\beta \equiv \beta' \rightarrow \beta''$, with $\beta' \leq_T \alpha'$ and $\alpha'' \leq_T \beta''$.

Then $\varepsilon = \{\alpha^* = \alpha'^* \rightarrow \alpha''^*\} \cup \{\beta^* = \beta'^* \rightarrow \beta''^*\} \cup \varepsilon' \cup \varepsilon''$ where $\varepsilon' \triangleq E\alpha' \cup E\beta'$ and $\varepsilon'' \triangleq E\alpha'' \cup E\beta''$.

By induction hypothesis $\forall \Sigma'.\Sigma' \# \varepsilon' \Rightarrow \vdash_A \Sigma',\varepsilon' \supset \beta'^* \leq \alpha'^*$ and $\forall \Sigma''.\ \Sigma'' \# \varepsilon'' \Rightarrow \vdash_A \Sigma'',\varepsilon'' \supset \alpha''^* \leq \beta''^*$.

Take any Σ such that $\Sigma \# \varepsilon$ then:

$\vdash_A \Sigma,\varepsilon' \supset \beta'^* \leq \alpha'^*$ and $\vdash_A \Sigma,\varepsilon'' \supset \alpha''^* \leq \beta''^*$.

By ε-weaken (note $\Sigma \# \varepsilon \Rightarrow \Sigma \# \varepsilon' \cup \varepsilon''$):

$\vdash_A \Sigma,\varepsilon' \cup \varepsilon'' \supset \beta'^* \leq \alpha'^*$ and $\vdash_A \Sigma,\varepsilon' \cup \varepsilon'' \supset \alpha''^* \leq \beta''^*$.

By Σ-weaken:

$\vdash_A \Sigma \cup \{\alpha^* \leq \beta^*\},\varepsilon' \cup \varepsilon'' \supset \beta'^* \leq \alpha'^*$ and $\vdash_A \Sigma \cup \{\alpha^* \leq \beta^*\},\varepsilon' \cup \varepsilon'' \supset \alpha''^* \leq \beta''^*$.

Hence, by applying $(\rightarrow_A)$ and (μ_A) we can conclude $\vdash_A \Sigma,\varepsilon \supset \alpha^* \leq \beta^*$. □

4.3.3 Proposition *(Soundness of $\leq_A$ for non-recursive types)*

Given $\alpha,\beta \in Tp$ non-recursive types then $\alpha \leq_A \beta \Rightarrow \alpha \leq_T \beta$.

Proof

We show $\vdash_A \emptyset,\varepsilon \supset \alpha^* \leq \beta^* \Rightarrow \alpha \leq_T \beta$, where $\varepsilon \triangleq E\alpha \cup E\beta$, by induction on the structure of α and β.

Case $\alpha\equiv\bot$. Then $\bot\leq_A\beta$ by $(\bot_A)$ ((assmp_A) does not apply), and also $\bot\leq_T\beta$.

Cases $\alpha\equiv\top$, $\alpha\equiv a$. Similar.

Case $\alpha\equiv\alpha'\to\alpha''$. Assume $\vdash_A\emptyset,\varepsilon\supset\alpha^*\leq\beta^*$; then the first step is either:

Case $(\top_A)$. Similar to the case $\alpha\equiv\bot$.

Case (μ_A). Then the second step is $(\to_A)$, that is also $\beta\equiv\beta'\to\beta''$
with $\{\alpha^*\leq\beta^*\},\varepsilon\supset\beta'^*\leq\alpha'^*$ and $\{\alpha^*\leq\beta^*\},\varepsilon\supset\alpha''^*\leq\beta''^*$,
where $\varepsilon=\{\alpha^*=\alpha'^*\to\alpha''^*\}\cup\{\beta^*=\beta'^*\to\beta''^*\}\cup\varepsilon'\cup\varepsilon''$ and
$\varepsilon'\triangleq E\alpha'\cup E\beta'$ and $\varepsilon''\triangleq E\alpha''\cup E\beta''$.
Since α,β contain no μ, $\text{Reach}(\beta'^*\to\alpha'^*,\varepsilon)\cap(\text{Dom}(\varepsilon'')\cup\{\alpha^*,\beta^*\})=\emptyset$.
By a simple analysis we have: $\vdash_A\{\alpha^*\leq\beta^*\},\varepsilon'\cup\varepsilon''\supset\beta'^*\leq\alpha'^*$.
By ε-strengthen $\vdash_A\{\alpha^*\leq\beta^*\},\varepsilon'\supset\beta'^*\leq\alpha'^*$. Similarly, $\vdash_A\{\alpha^*\leq\beta^*\},\varepsilon''\supset\alpha''^*\leq\beta''^*$.
Now, by Σ-strengthen $\vdash_A\emptyset,\varepsilon'\supset\beta'^*\leq\alpha'^*$ and $\vdash_A\emptyset,\varepsilon''\supset\alpha''^*\leq\beta''^*$.
By induction hypothesis $\beta'\leq_T\alpha'$ and $\alpha''\leq_T\beta''$; hence $\alpha'\to\alpha''\leq_T\beta'\to\beta''$. □

4.3.4 Lemma *(Uniformity of $\leq_A$)*
Let $\alpha,\beta\in\text{Type}$. If $\alpha\leq_A\beta$ then $\forall k.\alpha_{|k}\leq_A\beta_{|k}$.

Proof (sketch)

Given any k, from the execution tree of $\alpha\leq_A\beta$ it is possible to extract a successful execution tree for $\alpha_{|k}\leq_A\beta_{|k}$. The point is that the use of the (assmp_A) rule can be arbitrarily delayed by repeating a certain pattern of computation.

For example, consider $\mu t.\top\to t\leq_A\mu s.\bot\to s$, which gives raise to:

$$\varepsilon\triangleq\varepsilon_1\cup\varepsilon_2,\ \varepsilon_1\triangleq\{t_1=t_2\to t_1,\ t_2=\top\},\ \varepsilon_2\triangleq\{s_1=s_2\to s_1,\ s_2=\bot\}.$$

The execution tree of the initial goal $\emptyset,\varepsilon\supset t_1\leq s_1$ is:

$$\begin{array}{l}
\qquad\qquad\qquad

This is now the execution tree of a different initial goal, which might have originated from the problem $\top\to(\mu t.\top\to t) \leq_A \bot\to\mu s.(\bot\to s)$, which is equivalent to the original problem.

In a similar way, this execution tree can be further transformed into one for $\top\to(\top\to\bot) \leq_A \bot\to(\bot\to\bot)$ by replacing the (assmp_A) leaf with a $(\bot_A)$ leaf. By repeating this process we can obtain an execution tree for $\alpha_{|k} \leq_A \beta_{|k}$, for an arbitrarily large k. □

4.3.5 Proposition *(Soundness of $\leq_A$)*

Let $\alpha,\beta\in$ Type; if $\alpha\leq_A\beta$ then $\alpha\leq_T\beta$.

Proof

From 4.3.4 we have: $\alpha\leq_A\beta \Rightarrow \forall k.\alpha_{|k} \leq_A \beta_{|k}$.
From 4.3.3 and the definition of $\leq_T$ we have: $\forall k.\alpha_{|k} \leq_{fin} \beta_{|k}$ and $\alpha\leq_T\beta$. □

4.3.6 Lemma *(Faithfulness of $\leq_A$ w.r.t. paths)*

Let $\mathrm{lead}(\alpha,\varepsilon) \triangleq \mathrm{Sol}(\alpha,\varepsilon)(\mathrm{nil})$ be the first label of α in ε (that is, skipping initial variables in α,ε).

Let $\Sigma,\varepsilon \supset \alpha\leq\beta$ be the root of an execution tree, terminating with success or failure leaves, obtained from the rules in 4.2. Every node $\Sigma',\varepsilon \supset \alpha'\leq\beta'$ in the execution tree determines a path π from the root to itself, given by considering the occurrences of $(\to_A)$ and ignoring the other rules. Then:

1) Either α' and β' are both (bound) type variables, or neither is.

2) $T\alpha(\pi) = \mathrm{lead}(\alpha',\varepsilon)$ and $T\beta(\pi) = \mathrm{lead}(\beta',\varepsilon)$.

Proof

By induction on the depth of the execution tree. □

4.3.7 Proposition *(Completeness of $\leq_A$)*

Let $\alpha,\beta\in$ Type; if $\alpha\leq_T\beta$ then $\alpha\leq_A\beta$.

Proof

We show $\neg\,\alpha\leq_A\beta \Rightarrow \neg\, T\alpha\leq_\infty T\beta$.

By assumption, we have an execution tree for $\alpha\leq_A\beta$ which contains a failure node $\Sigma,\varepsilon \supset \alpha'\leq\beta'$, determining a path π as in Lemma 4.3.6. By 4.3.6.(2), $T\alpha(\pi) = \mathrm{lead}(\alpha',\varepsilon)$ and $T\beta(\pi) = \mathrm{lead}(\beta',\varepsilon)$. Hence we have a *common* path in $T\alpha$ and $T\beta$ corresponding to the failure node. The following table summarizes the possible cases for α',β' where the entry indicates either failure or the rule being applied by the algorithm; the n.a. (not applicable) cases come from 4.3.6.(1).

$\alpha'\backslash\beta'$	$\bot$	$\top$	s	b	$\beta'\to\beta''$
$\bot$	$\bot$	$\bot$	n.a.	$\bot$	$\bot$
$\top$	fail	$\top$	n.a.	fail	fail
t	n.a.	n.a.	assmp-μ	n.a.	n.a.
a	fail	$\top$	n.a.	var-fail	fail
$\alpha'\to\alpha''$	fail	$\top$	n.a.	fail	$\to$

Every "fail" in the algorithm corresponds to a situation where the two trees cannot be in the inclusion relation. □

4.4 An Implementation

In order to facilitate the proofs, the representation of data structures and algorithms given in 4.1 and 4.2 was rather abstract. In this section we show the beginning of a similar treatment for more concrete and traditional representations.

The computational rules in 4.2 can be converted into a straightforward and practical algorithm, based on the method of *trails* [28]. To reflect more closely actual implementations, we adopt the additional rules ($assmp'_A$), (μ_{lA}), and (μ_{rA}) described in 4.2. This results in the algorithm discussed informally in 1.5, but differs slightly from the one treated formally in 4.2 and 4.3 where the additional rules are not needed because the systems of equations are taken in canonical form.

A member α of μTp is represented as a directed cyclic graph l,S where the nodes in S are uniquely labeled (for example by memory addresses), and where l is the starting label. Each μ in α corresponds to a cycle in S.

More concretely, using an informal programming notation, S is a Store, where Store $\triangleq$ Label$\rightharpoonup$Node are the partial functions from labels to nodes (from memory addresses to memory locations). Then Graph $\triangleq$ Label$\times$Store, where Label $\triangleq$ Nat, and Node $\triangleq$ Bot+Top+Var(Tvar)+ Arrow(Label$\times$Label)+Rec(Label).

An *allocator* transforms a type into a graph structure:

Alloc: μTp$\times$Store$\times$(Tvar$\rightarrow$Label) $\rightarrow$ Graph

Let new(S) be a label l (for example the least one) such that l $\notin$ dom(S). We denote by S[l=Bot] a store that is just like S except that S(l)=Bot.

```
Alloc(⊥, S, e) ≜
    let l = new(S) in l,S[l=Bot]
Alloc(⊤, S, e) ≜
    let l = new(S) in l,S[l=Top]
Alloc(t, S, e) ≜
    if t∈dom(e) then e(t),S
    else let l=new(S) in l,S[l=Var(t)]
Alloc(α→β, S, e) ≜
    let l',S' = Alloc(α, S, e)
    and l",S" = Alloc(β, S', e)
    let l=new(S") in l,S"[l=Arrow(l',l")]
Alloc(μt.α, S, e) ≜
    let l = new(S)
    let l',S' = Alloc(α, S[l=Bot], e[t=l])
    in l,S'[l=Rec(l')]
```

The allocation of μt.α is done by reserving a new memory location l, then allocating the body α by binding every occurrence of t to l, and finally storing a Rec node containing the allocation of α back into l. The store S[l=Bot] is used in the recursion to prevent l from being returned again by new.

Given a path in ω^*, we can define a (partial) access function that returns the node corresponding to that path in a graph, but skipping over Rec nodes:

GT : Graph$\to$Tree(L) (Where Tree(L) = $\omega^* \rightharpoonup$ L, section 3.3)

GT(l,S)(nil) $\triangleq$
  if S(l) = Rec(l') then GT(l', S)(nil)
  if S(l) = Bot then $\bot$
  if S(l) = Top then $\top$
  if S(l) = Var(t) then t
  if S(l) = Arrow(l',l'') then $\to$
GT(l, S)(0.s) $\triangleq$
  if S(l) = Rec(l') then GT(l', S)(0.s)
  if S(l) = Arrow(l',l'') then GT(l',S)(s)
  else $\uparrow$
GT(l,S)(1.s) $\triangleq$
  if S(l) = Rec(l') then GT(l',S)(1.s)
  if S(l) = Arrow(l',l'') then GT(l'',S)(s)
  else $\uparrow$
GT(l,S)(n+2.s) $\triangleq \uparrow$

We now show that Alloc is correct, and that the initial state S is irrelevant.

4.4.1 Proposition

$\forall \alpha \in \mu Tp.\ \forall S,l',S'.$
  $Alloc(\alpha,S,[]) = l',S' \ \Rightarrow\ GT(l',S') = T\alpha$

Proof (sketch)

If $l \notin dom(S)$ then S[l=v] is a *single extension* of S. S' is an *extension* of S if it is S, or if it is the single extension of an extension of S.

We indicate by S^+ an arbitrary (finite) extension of S. Note that:

  If $l \in dom(S)$ then $S(l) = S^+(l)$.
  If $Alloc(\alpha, S, e) = l',S'$ then $l' \in dom(S')$ and S' is an extension of S.
  $\forall S^+.\ l \in dom(S) \Rightarrow GT(l,S) = GT(l,S^+)$.

To obtain the proposition, we need to prove a stronger statement:

$\forall \alpha \in \mu Tp.\ \forall n \geq 0.\ \forall S, m_1..m_n, \alpha_1..\alpha_n, l', S', \pi.$
  $Alloc(\alpha,S,[t_i=m_i]) = l',S' \ \wedge$
  $(\forall \pi' \text{ s.t. } |\pi'| \leq |\pi|.\ \forall S^+.\ GT(m_i,S^+)(\pi') = T\alpha_i(\pi') \text{ for all } i \in 1..n) \ \Rightarrow$
  $\forall S'^+.\ GT(l',S'^+)(\pi) = T([\alpha_i/t_i]\alpha)(\pi)$

The proof is then by induction on $|\pi|$; the hard case is $\pi = i.s$, $i \in \{0,1\}$, and $\alpha \equiv \mu t.\alpha' \to \alpha''$.

□

In the implementation of the algorithm, the assumption set Σ is represented as a *trail*, that is, a set of label pairs. This has the task of remembering the pairs of labels in the cyclic graphs that have been jointly visited.

From two types α and β we produce two graphs l^{α},S^{α}, $l^{\beta},S^{\alpha\beta}$ such that $S^{\alpha\beta}$ extends S^{α}.

Then $Alg(\emptyset,S^{\alpha\beta},l^{\alpha},l^{\beta})$ proceeds as follows, mimicking the rules in 4.2:

```
Alg(Tr, S, l, l') ≜
    if ⟨l,l'⟩ ∈ Tr then ok
    else if S(l) = Bot ok
    else if S(l') = Top then ok
    else if S(l) = Var(t) and S(l') = Var(t) then ok
    else if both S(l) =Arrow(l1,l2) and S(l') = Arrow(l1',l2') then
        Alg(Tr, S, l1', l1); Alg(Tr, S, l2, l2')
    else if S(l) = Rec(l1) and S(l') ≠ Rec(l1') then
        Alg(Tr∪⟨l,l'⟩, S, l1, l')
    else if S(l) ≠ Rec(l1) and S(l') = Rec(l1') then
        Alg(Tr∪⟨l,l'⟩, S, l, l1')
    else if S(l) = Rec(l1) and S(l') = Rec(l1') then
        Alg(Tr∪⟨l,l'⟩, S, l1, l1')
    else fail
```

An alternative approach is to avoid Rec nodes completely, and have the allocator construct direct loops in the graph. This leads to an algorithm where trails must be kept of every pair of nodes, instead of every pair of nodes of which one is a Rec node. This algorithm is closer to the formulation of the rules in 4.2, while the present algorithm, which in practice produces much shorter trails, uses the equivalent of the (μ_{lA}) and (μ_{rA}) rules described there.

4.4.2 Definition

$\alpha \leq_C \beta \quad \Leftrightarrow \quad \forall S,l',S',l'',S''.$

$\quad Alloc(\alpha,S,[])=l',S' \ \wedge \ Alloc(\beta,S',[])=l'',S'' \ \Rightarrow \ Alg(\emptyset,S'',l',l'') = ok$

From this point on it seems possible to mimic sections 4.3.4-4.3.7, modulo the use of the (μ_{lA}) and (μ_{rA}) rules, and show $\alpha \leq_C \beta$ iff $\alpha \leq_T \beta$, but we have not checked the cumbersome details.

5. Typing Rules

In this section we introduce a certain number of axioms and rules for type equality and subtyping. These are intended as natural rules for a language based on subtyping, and as a specification of a subtyping algorithm for such a language. In section 4 we have studied such a subtyping algorithm; here we see that the algorithm and the rules match each other perfectly, by relating them both to trees.

5.1 Type Equivalence Rules

We say that a type α is *contractive* in the type variable t if either t does not occur free in α, or α can be rewritten via unfolding as a type of the shape $\alpha_1 \to \alpha_2$. We write this fact as $\alpha \downarrow t$.

It is now easy to observe that the contractiveness of α in t is a sufficient (and necessary) condition to enforce the contractiveness of the following functional on the space Tree(L) (3.3):

$$G_{\alpha,t}(A) \triangleq [A/t]T\alpha \qquad A \in Tree(L)$$

([A/t]Tα denotes the substitution of the tree A for the occurrences of t in Tα.)

This remark suggests the following rule that is generalized to a larger calculus in [13]:

(contract) $\quad [\beta/t]\alpha = \beta,\ \ [\beta'/t]\alpha = \beta',\ \ \alpha \downarrow t \ \Rightarrow\ \beta = \beta'$

In words, if two types β and β' are fixpoints of the same functional $\alpha[t]$, then they are equal since contractive functionals have unique fixpoints. This rule was also inspired by a standard proof technique for bisimulation [23].

Moreover, it is convenient to identify $\mu t.t = \bot$.

In this section we consider the equivalence:

$$\vdash \alpha = \beta \qquad (\text{or } \alpha =_R \beta)$$

meaning that $\alpha = \beta$ can be derived in the congruence induced by the (contract) rule and the (fold-unfold) and (μ-$\bot$) axioms below. Here is the complete axiomatization:

(refl)	$\alpha = \alpha$
(symm)	$\alpha = \beta \ \Rightarrow\ \beta = \alpha$
(trans)	$\alpha = \beta,\ \ \beta = \gamma \ \Rightarrow\ \alpha = \gamma$
($\to$-congr)	$\alpha = \alpha',\ \ \beta = \beta' \ \Rightarrow\ \alpha \to \beta = \alpha' \to \beta'$
(μ-congr)	$\alpha = \beta \ \Rightarrow\ \mu t.\alpha = \mu t.\beta$
(μ-$\bot$)	$\mu t.t = \bot$
(fold-unfold)	$[\mu t.\alpha/t]\alpha = \mu t.\alpha$
(contract)	$[\beta/t]\alpha = \beta,\ \ [\beta'/t]\alpha = \beta',\ \ \alpha \downarrow t \ \Rightarrow\ \beta = \beta'$

5.1.1 Proposition *(Soundness of the equivalence rules w.r.t. the trees)*

$\alpha =_R \beta \ \Rightarrow\ T\alpha = T\beta$

Proof

Immediate by the previous considerations. □

5.1.2 Derived Rules

By means of (contract) and (fold-unfold) it is possible to prove new interesting equivalences, for example:

(1) $\quad \mu t.s \to t \ =\ \mu t.s \to (s \to t)$

(2) $\quad \mu t.\mu s.\alpha \ =\ \mu v.[v/t, v/s]\alpha \quad (\mu\text{-contraction})$

We make explicit a free variable by writing, for example, $\alpha[t]$.

Then we have:

(1) Consider $\gamma[r] = s\to(s\to r)$.
$\mu t.s\to t = s\to(\mu t.s\to t) = s\to(s\to(\mu t.s\to t)) = \gamma[\mu t.s\to t]$.
$\mu t.s\to(s\to t) = s\to(s\to(\mu t.s\to(s\to t))) = \gamma[\mu t.s\to(s\to t)]$.

(2) Let: $\alpha \equiv \mu t.\mu s.\gamma[t,s]$, $\alpha' \equiv \mu s.\gamma[\alpha,s] = \alpha$, $\beta \equiv \mu v.\gamma[v,v]$.
Consider $\gamma[w,w]$. Then: $\gamma[\alpha,\alpha] = \gamma[\alpha,\alpha'] = \alpha' = \alpha$ and $\gamma[\beta,\beta] = \beta$.

5.1.3 Reduction to Canonical Form

It easy to show that any recursive type is provably equivalent ($=_R$) to a type in canonical form. The strategy can be described as follows:

(a) Use unfold to get rid of all μ's that do not bind any variable.
(b) Use μ-contraction to reduce sequences of μ's to one μ.
(c) Use μ-$\bot$ to reduce to $\bot$ all subtypes of the shape $\mu t.t$.

5.2 Completeness of Equivalence Rules

By the strong connection between regular trees and recursive types we show that any time two recursive types α, β have the same tree expansion $T\alpha=T\beta$, then we can conclude $\vdash \alpha=\beta$.

First we show how to solve systems of type equations. Then we introduce the notion of *equational characterization* of a type; that is, how to characterize a type by a system of type equations. Finally we use equational characterizations to prove the completeness theorem.

In this section we use the following notation. If γ has free variables $\{u_1..u_p\} \subseteq \{t_1..t_n\}$, then we write $\gamma[\alpha_1...\alpha_n]$ for the substitution $[\alpha_1/t_1 \ ... \ \alpha_n/t_n]\gamma$. In particular, $\gamma[t_1...t_n]$ emphasizes a superset of the free variables of γ.

5.2.1 Lemma *(A system of equations has a solution, by iterated elimination)*

Every system of n equations in n variables:

$t_i=\gamma_i[t_1...t_n]$ $(i\in 1..n)$

has a solution in the congruence induced by the axiom (fold-unfold). That is, there are $\alpha_1...\alpha_n$ such that $\vdash \alpha_i=\gamma_i[\alpha_1...\alpha_n]$ $(i\in 1..n)$.

Proof

By induction on n.

Case n=1. Given the equation $t=\gamma[t]$ just take $\mu t.\gamma[t]$.

Case $n\geq 2$. Given the equations $t_i=\gamma_i[t_1...t_n]$ $(i\in 1..n)$ take $\alpha_n'[t_1...t_{n-1}] \equiv \mu t_n.\gamma_n[t_1...t_n]$.

Consider the system of n-1 equations: $t_i=\gamma_i[t_1...t_{n-1}\ \alpha_n'[t_1...t_{n-1}]]$ $(i\in 1..n-1)$

which by inductive hypothesis has solution $\alpha_1...\alpha_{n-1}$, that is :

$\alpha_i=\gamma_i[\alpha_1...\alpha_{n-1}\ \mu t_n.\gamma_n[\alpha_1...\alpha_{n-1}t_n]]$ $(i\in 1..n-1)$

Now take $\alpha_n \equiv \mu t_n.\gamma_n[\alpha_1...\alpha_{n-1}t_n]$ and check that $\alpha_1...\alpha_n$ is a solution for the original system. □

5.2.2 Lemma *(A system of contractive equations has a unique solution)*

Assume that, for $i\in 1..n$, we have two sets of types α_i, β_i, related by two systems of equations:

$\vdash \alpha_i=\gamma_i[\alpha_1...\alpha_n]$ $\vdash \beta_i=\gamma_i[\beta_1...\beta_n]$

such that $\gamma_i[t_1...t_n]\downarrow t_j$ for $i,j\in 1..n$. Then, for all i: $\vdash \alpha_i=\beta_i$.

Proof

By induction on n.

Case n=1. We have $\vdash \alpha=\gamma[\alpha]$ and $\vdash \beta=\gamma[\beta]$. Consider the context $\gamma[t]$, by (contract) we have $\vdash \alpha=\beta$.

Case n≥2. Consider the n-th equation. We have, by (fold-unfold),

$\vdash \mu t_n.\gamma_n[\alpha_1...\alpha_{n-1}\ t_n]=\gamma_n[\alpha_1...\alpha_{n-1}\ \mu t_n.\gamma_n[\alpha_1...\alpha_{n-1}\ t_n]]$

$\vdash \mu t_n.\gamma_n[\beta_1...\beta_{n-1}\ t_n]=\gamma_n[\beta_1...\beta_{n-1}\ \mu t_n.\gamma_n[\beta_1...\beta_{n-1}\ t_n]]$

Hence, by (contract) on t_n, $\vdash \alpha_n=\mu t_n.\gamma_n[\alpha_1...\alpha_{n-1}\ t_n]$, $\vdash \beta_n=\mu t_n.\gamma_n[\beta_1...\beta_{n-1}\ t_n]$.
Take $\gamma'[t_1...t_{n-1}] \equiv \mu t_n.\gamma_n[t_1...t_n]$. We can now construct a system of size n-1:

$t_i=\gamma_i[t_1...t_{n-1}\ \gamma'[t_1...t_{n-1}]]$ $\quad (i\in 1..n-1)$

and check that both $\alpha_1...\alpha_{n-1}$ and $\beta_1...\beta_{n-1}$ are solutions. Hence, by inductive hypothesis $\vdash \alpha_i=\beta_i$ for $i\in 1..n-1$. Moreover, by congruence we obtain $\vdash \alpha_n=\beta_n$. □

5.2.3 Definition

A *node context* $p[t_1...t_n]$ for $p\in L\setminus\{t_1...t_n\}$ (see 3.3 and proof 4.1.5) is a type of the form $p(u_1..u_{\#p})$, where #p is the arity of p, and $\{u_1..u_{\#p}\} \subseteq \{t_1...t_n\}$.

Node contexts provide a convenient meta-notation for nodes whose children are all type variables. For example, the type $\rightarrow(r,r)$ can be denoted by the node context $\rightarrow[r]$ or (redundantly) by $\rightarrow[r,s,t]$, and the type $\rightarrow(r,s)$ by $\rightarrow[s,r]$ among others.

Note that a node context $p[t_1...t_n]$ is contractive in each t_i, because either t_i is prefixed by p or does not occur in the type.

5.2.4 Definition

A type $\alpha\in$ Type is *equationally characterized* (*eq. char.*) if there are types $\alpha_1..\alpha_n$ with $\alpha\equiv\alpha_1$, and there are node contexts $p_i[t_1...t_n]$, $i\in 1..n$, for which $\vdash \alpha_i=p_i[\alpha_1...\alpha_n]$.

An equation, $t_j=p_j[t_1...t_n]$, in a system is reachable from a variable t_k if k=j, or if it is reachable from the variables in $p_k[t_1...t_n]$ (see 4.1.3). An equation is reachable from another if it is reachable from any of the variables in the other.

5.2.5 Lemma *(Building an equational characterization)*

Every term $\alpha\in$ Type has an equational characterization such that all equations are reachable from the first one.

Proof

The construction is basically the same as the one in 4.1.5. It is enough to prove by induction on the structure of γ that every term in μTp is equationally characterized. Then the lemma follows by 5.1.3, and by the invariance of equational characterization modulo provable equivalence. □

5.2.6 Lemma

Assume $T\alpha=T\beta$ and $\vdash \alpha=p(\alpha_1..\alpha_{\#p})$, $\vdash \beta=q(\beta_1..\beta_{\#q})$, where $p,q \in L$. Then p=q and $T\alpha_i=T\beta_i$ for all $i \in 1..\#p$.

Proof

By soundness, $T\alpha=Tp(\alpha_1..\alpha_{\#p})$ and $T\beta=Tq(\beta_1..\beta_{\#q})$. Hence, p=q and $T\alpha_i=T\beta_i$ by definition of T. □

5.2.7 Theorem *(Completeness of type equivalence rules)*

If $T\alpha=T\beta$ then $\alpha=_R\beta$

Proof

The idea of the proof is as follows: given α and β such that $T\alpha=T\beta$ we produce their corresponding equational characterizations, say $ec(\alpha)$ and $ec(\beta)$. By a collapse of "equivalent" equations we derive a new equational characterization $ec(\gamma)$. The solutions of the (smaller) system associated with $ec(\gamma)$ can be replicated to produce solutions for the systems associated with $ec(\alpha)$ and $ec(\beta)$. Hence we can apply twice Lemma 5.2.2 (uniqueness of solutions) and then transitivity to conclude $\alpha=_R\beta$.

Let $T\alpha=T\beta$; by Lemma 5.2.5, α,β are equationally characterized by α_i, $t_i=p_i[t_1...t_n]$ and β_j, $t_j=q_j[t_1...t_m]$ so that all equations are reachable from the first ones.

From these α_i,β_j we generate a sequence of pairs (A^h,B^h) where A^h,B^h are equivalence classes of α_i and β_j respectively. Moreover, for each h, $\alpha_{i1},\alpha_{i2}\in A^h$, and $\beta_{j1},\beta_{j2}\in B^h$, we shall have the invariant $T\alpha_{i1}=T\alpha_{i2}=T\beta_{j1}=T\beta_{j2}$.

We start with the pair $(A^1,B^1)\equiv(\{\alpha\},\{\beta\})$. At each step we consider all the pairs α_i,β_j such that $\alpha_i\in A^h$ and $\beta_j\in B^h$ for some h. We indicate by $\alpha_{(i',i'')}$ some α_i depending on both i' and i''; similarly for $\beta_{(j',j'')}$. If $\alpha_i=p_i(\alpha_{(i,1)}...\alpha_{(i,\#p_i)})$ and $\beta_j=q_j(\beta_{(j,1)}...\beta_{(j,\#q_j)})$, we have, by Lemma 5.2.6, $p_i=q_j$ and $T\alpha_{(i,1)}=T\beta_{(j,1)}$ ... $T\alpha_{(i,\#p_i)}=T\beta_{(j,\#p_i)}$. We add all the pairs $(\alpha',\beta')\in\{(\alpha_{(i,1)},\beta_{(j,1)}),...,(\alpha_{(i,\#p_i)},\beta_{(j,\#p_i)})\}$ in the following way, respecting the invariant above:

- if $\alpha'\in A^h$ and $\beta'\in B^h$ for some h, then nothing is done;
- else, if $\alpha'\in A^{h1}$, and $\beta'\in B^{h2}$, with $h1\neq h2$, then we replace the pairs (A^{h1},B^{h1}) and (A^{h2},B^{h2}) by $(A^{h1}\cup A^{h2},B^{h1}\cup B^{h2})$;
- else, if $\alpha'\in A^h$ we replace the pair (A^h,B^h) by $(A^h,B^h\cup\{\beta'\})$;
- else, if $\beta'\in B^h$ we replace the pair (A^h,B^h) by $(A^h\cup\{\alpha'\},B^h)$;
- else we add a new pair $(\{\alpha'\},\{\beta'\})$.

We stop when the list of pairs no longer changes. This process terminates because there are at most $n\cdot m$ pairs to consider.

The process above produces two partitions of α_i and β_j of size $k\leq n$, $k\leq m$, for some k. These are total partitions since all equations are reachable from the first ones. These partitions determine two functions $\sigma:1..n\to1..k$ and $\pi:1..m\to1..k$ such that:

- $\sigma(i)=\pi(j)\Leftrightarrow\alpha_i\in A^h\ \beta_j\in B^h$ for some h
- $\sigma(i1)=\sigma(i2)\Leftrightarrow\alpha_{i1},\alpha_{i2}\in A^h$ for some h
- $\pi(i1)=\pi(i2)\Leftrightarrow\beta_{i1},\beta_{i2}\in B^h$ for some h

Given these partitions, we now define a system of k equations $t_h=r_h[t_1..t_k]$, which will turn out to be equivalent both to the p_i and the q_j systems. For $h\in1..k$ we have:

$t_h = r_h[t_1..t_k]$ where
$r_{\sigma(i)}[t_1..t_k] \equiv p_i[t_{\sigma(1)}...t_{\sigma(n)}]$
$r_{\pi(j)}[t_1..t_k] \equiv q_j[t_{\pi(1)}...t_{\pi(m)}]$

We need to argue that this is a proper definition, since we can have, for example, $\sigma(i)=\pi(j)$ for some i,j. We show that when this happens, we also have by construction that $r_{\sigma(i)}[t_1..t_k] \equiv r_{\pi(j)}[t_1..t_k]$. Similarly for the other possible conflicts: $\sigma(i1)=\sigma(i2)$ for some i1,i2, and $\pi(i1)=\pi(i2)$ for some j1,j2. To show these facts, we further investigate the properties of σ and π.

- $\sigma(1) = \pi(1)$ since α,β start in the same pair (A^1,B^1).

- if $\sigma(i) = \pi(j)$ then $p_i \equiv q_j$. Moreover, let $\alpha_i = p_i[\alpha_1...\alpha_n] \equiv p_i(\alpha_{(i,1)}...\alpha_{(i,\#p_i)})$ and $\beta_j = q_j[\beta_1...\beta_m] \equiv q_j(\beta_{(j,1)}...\beta_{(j,\#qj)})$ be the i-th and j-th equations in the respective systems. Then $\alpha_i \in A^h$, $\beta_j \in B^h$ for some h (property above); the pair α_i,β_j was considered in the process above; that is, the pairs $\alpha_{(i,1)},\beta_{(j,1)}$ ... $\alpha_{(i,\#p_i)},\beta_{(j,\#p_i)}$ were also added to the list. Therefore $\sigma(i,1)=\pi(j,1)$ ... $\sigma(i,\#p_i)=\pi(j,\#q_j)$, and $p_i(t_{\sigma(i,1)}...t_{\sigma(i,\#p_i)}) \equiv q_j(t_{\pi(j,1)}...t_{\pi(j,\#q_j)})$. This is the same as saying $p_i[t_{\sigma(1)}...t_{\sigma(n)}] \equiv q_j[t_{\pi(1)}...t_{\pi(m)}]$.

- if $\sigma(i1) = \sigma(i2)$ then $p_{i1} \equiv p_{i2}$. Moreover, let $\alpha_{i1} = p_{i1}(\alpha_{(i1,1)}...\alpha_{(i1,\#p_{i1})})$ and $\alpha_{i2} = p_{i2}(\alpha_{(i2,1)}...\alpha_{(i2,\#p_{i2})})$ be the i1-th and i2-th equations in the α system. Then $\alpha_{i1},\alpha_{i2} \in A^h$ for some h (property above). Consider any $\beta_j \in B^h$; the pairs α_{i1},β_j, α_{i2},β_j were considered in the process above, that is the pairs $\alpha_{(i1,1)},\beta_{(j,1)}$, $\alpha_{(i2,1)},\beta_{(j,1)}$ were also added to the list. Therefore $\sigma(i1,1)=\pi(j,1)=\sigma(i2,1)$, and similarly up to $\sigma(i1,\#p_{i1})=\sigma(i2,\#p_{i2})$. Hence: $p_{i1}(t_{\sigma(i1,1)}...t_{\sigma(i1,\#p_{i1})}) \equiv p_{i2}(t_{\sigma(i2,1)}...t_{\sigma(i2,\#p_{i2})})$. This is the same as saying $p_{i1}[t_{\sigma(1)}...t_{\sigma(n)}] \equiv p_{i2}[t_{\sigma(1)}...t_{\sigma(n)}]$.

- similarly for $\pi(i1) = \pi(i2)$.

Hence we conclude:

- if $\sigma(i) = \pi(j)$ then $r_{\sigma(i)}[t_1..t_k] \equiv p_i[t_{\sigma(1)}...t_{\sigma(n)}] \equiv q_j[t_{\pi(1)}...t_{\pi(m)}] \equiv r_{\pi(j)}[t_1..t_k]$
- if $\sigma(i1)=\sigma(i2)$ then $r_{\sigma(i1)}[t_1..t_k] \equiv p_{i1}[t_{\sigma(1)}...t_{\sigma(n)}] \equiv p_{i2}[t_{\sigma(1)}...t_{\sigma(n)}] \equiv r_{\sigma(i2)}[t_1..t_k]$
- similarly for $\pi(j1)=\pi(j2)$

Now by Lemma 5.2.1 we can construct a solution of the system $t_h = r_h[t_1..t_k]$; that is, we can obtain $\gamma_1..\gamma_k$ such that $\vdash \gamma_h = r_h[\gamma_1..\gamma_k]$.

Then $\vdash \gamma_{\sigma(i)} = r_{\sigma(i)}[\gamma_1..\gamma_k] \equiv p_i[\gamma_{\sigma(1)}...\gamma_{\sigma(n)}]$ for all i. Therefore, the γ's (when appropriately replicated) satisfy the same system as the α's, and by Lemma 5.2.2 we have $\vdash \alpha_i = \gamma_{\sigma(i)}$. Similarly, the γ's satisfy the β's system, and $\vdash \beta_j = \gamma_{\pi(j)}$. Moreover, $\sigma(1) = \pi(1)$, hence $\vdash \alpha \equiv \alpha_1 = \gamma_{\sigma(1)} = \gamma_{\pi(1)} = \beta_1 \equiv \beta$ by transitivity. □

This constructive proof is based on the one in [25] (see also [21]), but differs in an important point as, in addition, we must deal with equivalence classes of types.

5.2.8 Example

In this example, arising from a discussion with Mario Coppo, we consider the types:

$\alpha \triangleq \mu t.t \to (t \to t)$ $\beta \triangleq \mu t.(t \to t) \to t$

We have $T\alpha=T\beta$, but note that there is no single context that can prove them equivalent by the (contract) rule. We must find a third type γ which is independently provably equal to α and β by (contract), and then we can obtain $\vdash \alpha=\beta$ by transitivity. To find this γ, we instantiate the proof of 5.2.7.

We start with two equational characterizations for α and β:

$$\begin{array}{ll} \alpha_1 \triangleq \alpha & \beta_1 \triangleq \beta \\ \alpha_2 \triangleq \alpha\to\alpha & \beta_2 \triangleq \beta\to\beta \\ p_1[t_1,t_2] \triangleq t_1\to t_2 & q_1[t_1,t_2] \triangleq t_2\to t_1 \\ p_2[t_1,t_2] \triangleq t_1\to t_1 & q_2[t_1,t_2] \triangleq t_1\to t_1 \end{array}$$

That is, the following are provable, by (fold-unfold):

$$\begin{array}{ll} \alpha_1 = \alpha_1\to\alpha_2 & \beta_1 = \beta_2\to\beta_1 \\ \alpha_2 = \alpha_1\to\alpha_1 & \beta_2 = \beta_1\to\beta_1 \end{array}$$

Starting with the list $(\{\alpha_1\},\{\beta_1\})$, we must match the equations for α_1 and β_1. This involves equating the pairs α_1,β_2 (obtaining $(\{\alpha_1\},\{\beta_1,\beta_2\})$, and α_2,β_1 (obtaining $(\{\alpha_1,\alpha_2\},\{\beta_1,\beta_2\})$. Matching the newly inserted pairs does not further modify the situation, hence we have reached termination with the partitions:

$(\{\alpha_1,\alpha_2\},\{\beta_1,\beta_2\})$ with k=1 and $\sigma=\pi=\{1\mapsto 1,\ 2\mapsto 1\}$

The associated system of one equation is $t_1 = r_1[t_1]$, where:

$$r_1[t_1] = \left\{ \begin{array}{c} r_{\sigma(1)}[t_1] \equiv p_1[t_{\sigma(1)},t_{\sigma(2)}] \\ r_{\sigma(2)}[t_1] \equiv p_2[t_{\sigma(1)},t_{\sigma(2)}] \\ r_{\pi(1)}[t_1] \equiv q_1[t_{\pi(1)},t_{\pi(2)}] \\ r_{\pi(2)}[t_1] \equiv q_2[t_{\pi(1)},t_{\pi(2)}] \end{array} \right\} = t_1\to t_1$$

We now generate a solution for this system:

$\gamma_1 \triangleq \mu t.t\to t$ such that $\vdash \gamma_1 = \gamma_1\to\gamma_1 = r_1[\gamma_1]$

We can verify that (γ_1,γ_1) solves the α and β systems:

$$\begin{array}{ll} \vdash \gamma_1=p_1[\gamma_1,\gamma_1] & \vdash \gamma_1=q_1[\gamma_1,\gamma_1] \\ \vdash \gamma_1=p_2[\gamma_1,\gamma_1] & \vdash \gamma_1=q_2[\gamma_1,\gamma_1] \end{array}$$

Hence a proof of $\vdash\alpha_1=\gamma_1$ can be constructed by Lemma 5.2.2 (more simply, by unfolding $\vdash \alpha_1 = \alpha_1\to(\alpha_1\to\alpha_1)$ and $\vdash \gamma_1 = \gamma_1\to\gamma_1 = \gamma_1\to(\gamma_1\to\gamma_1)$, hence $\vdash \alpha_1 = \gamma_1$ by (contract)). Similarly, $\vdash \gamma_1 = \beta_1$. Hence by transitivity, $\vdash \alpha_1 = \beta_1$.

5.3 Subtyping Rules

At first it is not clear how to define a rule for the subtyping of recursive types that is sufficiently powerful. In particular, observe that the computational rule (μ_A) in section 4.2 does not have any apparent logical meaning as the premise is always valid under a classical reading of the entailment relation.

We now introduce a rule, (μ_R), whose soundness is clear. Later, in section 5.4, we will show that in conjunction with the type equivalence rules, (μ_R) leads to a subtyping system complete with respect to the tree ordering.

We denote with Γ a set $\{t_1 \le s_1, ..., t_n \le s_n\}$ of subtyping assumptions on type variables. We write a subtype judgment as: $\Gamma \supset \alpha \le \beta$.

Define a formal system for deriving this kind of judgments as follows; this is based on the $\alpha = \beta$ congruence in 5.1:

(eq_R)	$\alpha = \beta \;\Rightarrow\; \Gamma \supset \alpha \le \beta$
$(trans_R)$	$\Gamma \supset \alpha \le \beta,\;\; \Gamma \supset \beta \le \gamma \;\Rightarrow\; \Gamma \supset \alpha \le \gamma$
$(assmp_R)$	$t \le s \in \Gamma \;\Rightarrow\; \Gamma \supset t \le s$
$(\bot_R)$	$\Gamma \supset \bot \le \alpha$
$(\top_R)$	$\Gamma \supset \alpha \le \top$
$(\to_R)$	$\Gamma \supset \alpha' \le \alpha,\;\; \Gamma \supset \beta \le \beta' \;\Rightarrow\; \Gamma \supset \alpha \to \beta \le \alpha' \to \beta'$
(μ_R)	$\Gamma \cup \{t \le s\} \supset \alpha \le \beta \;\Rightarrow\; \Gamma \supset \mu t.\alpha \le \mu s.\beta$ with t only in α; s only in β; t,s not in Γ

We say $\alpha \le_R \beta$ if we can derive $\emptyset \supset \alpha \le \beta$. The last rule was proposed in [10] in the specification of the Amber programming language as a first attempt to define a theory for the subtyping of recursive types.

5.3.1 Proposition *(Soundness of the rule ordering w.r.t. the tree ordering)*

If $\alpha \le_R \beta$ then $\alpha \le_T \beta$.

Proof

We prove the more general statement:

If $\vdash_R \{t_1 \le s_1, .., t_n \le s_n\} \supset \alpha \le \beta$
and $\alpha_1 \le_T \beta_1, .., \alpha_n \le_T \beta_n$ so that $\{t_1, s_1, .., t_n, s_n\} \cap FV(\alpha_1, \beta_1, .., \alpha_n, \beta_n) = \emptyset$
then $[\alpha_1/t_1, \beta_1/s_1, .., \alpha_n/t_n, \beta_n/s_n]\alpha \le_T [\alpha_1/t_1, \beta_1/s_1, .., \alpha_n/t_n, \beta_n/s_n]\beta$.

The proof goes by induction on the length of the derivation $\vdash_R$. The only interesting cases arise for (μ_R) and (eq_R).

For brevity we write lists such as $t_1 \le s_1, ..., t_n \le s_n$ in the form $t_i \le s_i$ for a free i.

Case (μ_R) $\{t_i \le s_i, t \le s\} \supset \alpha \le \beta \;\Rightarrow\; \{t_i \le s_i\} \supset \mu t.\alpha \le \mu s.\beta$
with $t \notin FV(\beta)$; $s \notin FV(\alpha)$; $t,s \ne t_i, s_i$ for any i.

By induction hypothesis:

$\forall \alpha_i \le_T \beta_i, \underline{\alpha} \le_T \underline{\beta}$ such that $\{t_i, s_i, t, s\} \cap FV(\alpha_i, \beta_i, \underline{\alpha}, \underline{\beta}) = \emptyset$.
$[\alpha_i/t_i, \beta_i/s_i, \underline{\alpha}/t]\alpha \le_T [\alpha_i/t_i, \beta_i/s_i, \underline{\beta}/s]\beta$

Define $\alpha^0 \triangleq \bot$ $\quad \alpha^{n+1} \triangleq [\alpha_i/t_i, \beta_i/s_i, \alpha^n/t]\alpha$
$\beta^0 \triangleq \bot$ $\quad \beta^{n+1} \triangleq [\alpha_i/t_i, \beta_i/s_i, \beta^n/s]\beta$

Applying the induction hypothesis with $\underline{\alpha} = \alpha^n, \underline{\beta} = \beta^n$ we obtain
$\alpha^{n+1} \le_T \beta^{n+1}$ for every n.

For every k we can then choose an n sufficiently large so that:
$(\mu t.[\alpha_i/t_i, \beta_i/s_i]\alpha)_{|k} =_T \alpha^n{}_{|k} \le_T \beta^n{}_{|k} =_T (\mu s.[\alpha_i/t_i, \beta_i/s_i]\beta)_{|k}$
(such n is found by examining how t and s occur in α and β).

Hence, by definition of $\le_T$ for recursive types, we have shown:
$[\alpha_i/t_i, \beta_i/s_i](\mu t.\alpha) \le_T [\alpha_i/t_i, \beta_i/s_i](\mu s.\beta)$

Case (eq_R) $\quad \alpha =_R \beta \ \Rightarrow \ \{t_i \le s_i\} \supset \alpha \le \beta$

Since $=_R$ is a congruence, we have $[\alpha_i/t_i, \beta_i/s_i]\alpha \ =_R \ [\alpha_i/t_i, \beta_i/s_i]\beta$.
By soundness of $=_R$ we have $[\alpha_i/t_i, \beta_i/s_i]\alpha \ =_T \ [\alpha_i/t_i, \beta_i/s_i]\beta$. Finally,
$[\alpha_i/t_i, \beta_i/s_i]\alpha \ \le_T \ [\alpha_i/t_i, \beta_i/s_i]\beta$ since $\le_T$ is a preorder. □

Remarks

5.3.2 It is easy to observe that if we prove something in the system without using (eq_R) and ($trans_R$) then all the assumptions, $t \le s$, inserted in Γ when applying the rule (μ_R) can be used only with respect to a pair of *positive* occurrences of t in α and s in β.

5.3.3 Then one may wonder whether the following rule suffices for our purposes [5]:

$\alpha \le \beta$, Monotonic(t,α), Monotonic(t,β) $\Rightarrow \ \mu t.\alpha \le \mu t.\beta$
where Monotonic(t,α) iff t does not occur negatively in α.

Unfortunately it does not, as we cannot prove inclusions involving negative occurrences, as in $\mu t.t \to t \le \mu t.\bot \to t$.

Moreover, one must be careful in defining "t does not occur negatively in α" for recursive types, in order to ensure that α is really monotonic in t (for example, $\mu s.s \to t$ is not monotonic in t):

PosAlso(t,t) $\triangleq$ True, PosAlso(t,s) $\triangleq$ False $(s \neq t)$
PosAlso(t,$\bot$) $\triangleq$ False, PosAlso(t,$\top$) $\triangleq$ False
PosAlso(t,$\alpha \to \beta$) $\triangleq$ NegAlso(t,α) $\vee$ PosAlso(t,β)
PosAlso(t,$\mu s.\alpha$) $\triangleq$ (NegAlso(s,α) $\wedge$ $t \in FV(\alpha)$) $\vee$ PosAlso(t,α) $(s \neq t)$

NegAlso(t,s) $\triangleq$ False (even when s=t)
NegAlso(t,$\bot$) $\triangleq$ False, NegAlso(t,$\top$) $\triangleq$ False
NegAlso(t,$\alpha \to \beta$) $\triangleq$ PosAlso(t,α) $\vee$ NegAlso(t,β)
NegAlso(t,$\mu s.\alpha$) $\triangleq$ (NegAlso(s,α) $\wedge$ $t \in FV(\alpha)$) $\vee$ NegAlso(t,α) $(s \neq t)$

Monotonic(t, α) $\triangleq \ \neg$ NegAlso(t, α)

Under these conditions, it is possible to show that the rule above is provable from the system in 5.3.

5.4 Completeness of Subtyping Rules.

In proving the completeness of the subtyping rules w.r.t. the tree ordering, it seems helpful to go through the algorithm. The rather obvious approach of extracting a proof from a successful execution tree is complicated by the lack of correspondence between the computational rule (μ_A) and the rule (μ_R), as the former can be applied repeatedly on the same variable, whereas the latter can be applied at most once.

One may wonder if it is possible to rearrange the regular systems, while preserving type equivalence, so that during the execution we never have to expand twice the same variable by means of (μ_A).

Naively this corresponds to a *controlled* unfolding of the recursive types so that the corresponding μ's appear at the same time in the visit of the trees. [3]For example, to prove $\mu t.t \to t$

$\leq_R \mu s.(\mu s'.\top\to s')\to s$, we unfold the first type to $\mu t.(\mu t'.t'\to t')\to t$; note that this is not the unfolding given by the (fold-unfold) rule.

If $\vdash_A \Sigma,\varepsilon\supset t\leq s$ (see 4.2.3), we say that (the successful execution tree of) the initial goal $\Sigma,\varepsilon\supset t\leq s$ has the *one-expansion property* iff the following is true: for every $t\in Dom(\varepsilon)$ and for each path p of the execution tree, t is expanded in a (μ_A) node of p at most once.

It follows that with one-expansion, each variable can be inserted in Σ in a unique way, so that for each pair of assumptions $t_1\leq s_1$, $t_2\leq s_2 \in \Sigma$ we have that t_1,s_1,t_2,s_2 are pairwise distinct. Moreover, if we consider two (μ_A) nodes $\Sigma,\varepsilon\supset t_1\leq s_1$, $\Sigma,\varepsilon\supset t_2\leq s_2$ on the same path then t_1,s_1,t_2,s_2 are pairwise distinct, and if we consider a (μ_A) node $\Sigma,\varepsilon\supset t_1\leq s_1$ and an ($assmp_A$) node $\Sigma,\varepsilon\supset t_2\leq s_2$ on the same path then either $t_1\equiv t_2$, $s_1\equiv s_2$ or t_1,s_1,t_2,s_2 are pairwise distinct.

5.4.1 Lemma *(Putting recursions in lockstep)*

If $\vdash_A\Sigma,\varepsilon\supset t\leq s$ then there are θ, r, u such that $\vdash_A\Sigma,\theta\supset r\leq u$, $Sol(r,\theta) = Sol(t,\varepsilon)$, $Sol(u,\theta) = Sol(s,\varepsilon)$ and $\Sigma,\theta\supset r\leq u$ satisfies the one-expansion property.

Proof

Given the initial goal $\Sigma,\varepsilon\supset t\leq s$ and the related successful execution tree we build a new judgment $\Sigma,\theta\supset r\leq u$ such that the following properties hold:

(a) $\Sigma,\theta\supset r\leq u$ is an initial goal.
(b) $Sol(r,\theta) = Sol(t,\varepsilon)$ and $Sol(u,\theta) = Sol(s,\varepsilon)$.
(c) $\vdash_A\Sigma,\theta\supset r\leq u$, and the execution tree is equal to the one for $\Sigma,\varepsilon\supset t\leq s$ modulo variable renaming.
(d) $\Sigma,\theta\supset r\leq u$ satisfies the one-expansion property.

First we build the execution tree of $\Sigma,\varepsilon\supset t\leq s$. Then we associate with every node of the tree a couple (r, u) (or (u, r) on negative branches) of fresh variables with the following constraint; with every assumption leaf for $t\leq s$ we associate the same pair of variables as with the μ node where the assumption $t\leq s$ has been introduced into Σ (if any).

Next generate θ according to the following cases:

Case (μ-$\bot$). Say we are in the situation: $\Sigma',\varepsilon\supset\bot\leq\beta \Rightarrow \Sigma',\varepsilon\supset t\leq s_0$ where $\varepsilon(t)=\bot$. If (r,u_0) is the pair of variables associated with the μ-node add the equations:

$r=\bot$
$[u_0/s_0,u_1/s_1, ..., u_n/s_n](s_i=\varepsilon(s_i))$ for $i\in 0..n$

where $u_1...u_n$ are fresh variables and $s_1...s_n$ are the variables reachable from s_0 in the system ε, that is $\{s_1...s_n\} = Reach(s_0, \varepsilon)\cap Dom(\varepsilon)$.

Case (μ-$\top$). Analogous.

Case (μ-var). Say we are in the situation $\Sigma',\varepsilon\supset a\leq a \Rightarrow \Sigma',\varepsilon\supset t\leq s$. If (r,u) is the pair of variables associated with the μ-node, we add a pair of equations: r=a, u=a.

Case (μ-$\to$). Say we are in the situation:

$\Sigma'\cup\{t\leq s\},\varepsilon\supset s_1\leq t_1,\ \Sigma'\cup\{t\leq s\},\varepsilon\supset t_2\leq s_2$
$\Rightarrow \Sigma'\cup\{t\leq s\},\varepsilon\supset t_1\to t_2\leq s_1\to s_2 \Rightarrow \Sigma',\varepsilon\supset t\leq s$

where we have the fresh variables r, r_1, r_2 for t, t_1, t_2 and u, u_1, u_2 for s, s_1, s_2 (the variables associated to an $\rightarrow$-node are inessential) then we generate the equations $r = r_1 \rightarrow r_2$ and $u = u_1 \rightarrow u_2$.

Case (μ-assmp1). Say we are in the situation: $\Sigma', \varepsilon \supset a \leq b \Rightarrow \Sigma', \varepsilon \supset t \leq s$ where $a \leq b \in \Sigma$. If (r,u) is the pair of variables associated with the μ-node, we add a pair of equations: r=a, u=b.

Case (μ-assmp2). Finally, if we visit a node in which we apply the rule (assmp_A) w.r.t. an assumption added during the computation then we do not generate any equation. In fact, the equations corresponding to those variables are defined in the corresponding μ-node in which the assumption was made.

Let us now consider the properties (a-d):
(a) Follows from the use of fresh variables.
(b) In the first place one establishes a relation R, say, between the variables reachable from t and those reachable from r. In general we will have a situation in which a variable t may correspond to many variables $r_1 ... r_n$. Next, prove by induction on the lowest level of the appearance of r in the execution tree and $|\pi|$ that $(t, r) \in R$ implies $\text{Sol}(r, \theta)(\pi) = \text{Sol}(t, \varepsilon)(\pi)$.
(c) By construction at each step we can apply the same computational rule.
(d) This is a consequence of the constraint on the assignment of fresh variables to nodes. □

5.4.2 Example

Consider the types: $\mu t.\top \rightarrow t \leq_\top \bot \rightarrow (\mu s.s \rightarrow s)$. These types are in minimal form, i.e. they are the smallest types that can describe the corresponding regular trees, but still the recursions are not in lockstep; we need to transform them into more redundant forms, in order to synchronize them. In the following we pedantically apply the procedure described in the proof of the previous lemma.

Let us assume that the types are described by the canonical system ε:

$$\varepsilon \triangleq \varepsilon_1 \cup \varepsilon_2,\ \varepsilon_1 \triangleq \{t_1 = t_2 \rightarrow t_1,\ t_2 = \top\},\ \varepsilon_2 \triangleq \{s_1 = s_2 \rightarrow s_3,\ s_2 = \bot,\ s_3 = s_3 \rightarrow s_3\}.$$

The following describes the successful execution tree associated to the initial goal $\emptyset, \varepsilon \supset t_1 \leq s_1$:

$$\begin{array}{ccccc}
 & & (\top_A) & & (\text{assmp}_A) \\
 & & \Rightarrow \{t_1 \leq s_1, t_1 \leq s_3\}, \varepsilon \supset s_3 \leq t_2 & & \Rightarrow \{t_1 \leq s_1, t_1 \leq s_3\}, \varepsilon \supset t_1 \leq s_3 \\
 & (\bot_A) & & \Rightarrow \{t_1 \leq s_1, t_1 \leq s_3\}, \varepsilon \supset t_2 \rightarrow t_1 \leq s_3 \rightarrow s_3 & \\
\Rightarrow \{t_1 \leq s_1\}, \varepsilon \supset s_2 \leq t_2 & & & \Rightarrow \{t_1 \leq s_1\}, \varepsilon \supset t_1 \leq s_3 & \\
 & \Rightarrow \{t_1 \leq s_1\}, \varepsilon \supset t_2 \rightarrow t_1 \leq s_2 \rightarrow s_3 & & & \\
 & \Rightarrow \emptyset, \varepsilon \supset t_1 \leq s_1 & & &
\end{array}$$

Observe that this execution tree does *not* have the one-expansion property as the variable t_1 is expanded twice. Hence we start associating fresh variables to each node according to the rules described in the proof. The following describes which rule is being applied at each node of the execution tree, and which pair of fresh variables we associate to each node.

$(\top_A)\ (u_6,r_6)$ $(assmp_A)\ (r_3,u_3)$
$(\rightarrow_A)\ (r_5,u_5)$
$(\bot_A)\ (u_4,r_4)$ $(\mu_A)\ (r_3,u_3)$
$(\rightarrow_A)\ (r_2,u_2)$
$(\mu_A)\ (r_1,u_1)$

We now compute the new type environment $\theta = \theta_1 \cup \theta_2$, where:

$\theta_1 = \{r_1=r_4\rightarrow r_3,\ r_4=\top,\ r_3=r_6\rightarrow r_3,\ r_6=\top\}$,
$\theta_2 = \{u_1=u_4\rightarrow u_3,\ u_4=\bot,\ u_3=u_6\rightarrow u_3,\ u_6=u_6\rightarrow u_6\}$

Observe here that the equation $u_6=u_6\rightarrow u_6$ is generated by calculating: $[u_6/s_3](s_3=\varepsilon(s_3))$. No more equations are needed as s_3 is the only variable reachable from s_3. Verify that:

$T(\mu t.\top\rightarrow t) = Sol(t_1,\varepsilon) = Sol(r_1,\theta)$, $\quad T(\bot\rightarrow(\mu s.s\rightarrow s)) = Sol(s_1,\varepsilon) = Sol(u_1,\theta)$.

We finally compute the successful execution tree, *with one expansion property*, associated to the initial goal $\emptyset,\theta\supset r_1\leq u_1$:

$(\top_A)$ $(assmp_A)$
$\Rightarrow\ \{r_1\leq u_1, r_3\leq u_3\},\theta\supset u_6\leq r_6$ $\Rightarrow\ \{r_1\leq u_1, r_3\leq u_3\},\theta\supset r_3\leq u_3$
$(\bot_A)$ $\Rightarrow\ \{r_1\leq u_1, r_3\leq u_3\},\theta\supset r_6\rightarrow r_3\leq u_6\rightarrow u_3$
$\Rightarrow\ \{r_1\leq u_1\},\theta\supset u_4\leq r_4$ $\Rightarrow\ \{r_1\leq u_1\},\theta\supset r_3\leq u_3$
$\Rightarrow\ \{r_1\leq u_1\},\theta\supset r_4\rightarrow r_3\leq u_4\rightarrow u_3$
$\Rightarrow\ \emptyset,\theta\supset r_1\leq u_1$

5.4.3 Lemma *(From the execution tree to the proof tree)*
If $\vdash_A \Sigma,\varepsilon\supset t\leq s$ (see 4.2.3) and its execution tree has the one-expansion property, then $\vdash_R \Sigma \supset \langle t,\varepsilon\rangle \leq \langle s,\varepsilon\rangle$.

Proof

We proceed by induction on the depth k of the successful execution tree of an initial goal $\Sigma,\varepsilon\supset t\leq s$ (see 4.2). Depth is measured by the number of adjacent pairs of nodes (μ_A)-$(\rightarrow_A)$ in the longest branch from the root. In the inductive case, each subgoal is converted into an initial goal of the same depth, in order to apply the induction hypothesis.

Case k=0.

The tree consists of a (μ_A) root (since the goal is initial) and a single leaf which is either $(assmp_A)$, $(\bot_A)$, $(\top_A)$, or (var_A). Then after the application of the (μ_A) rule, with $s,t\in Dom(\varepsilon)$, we are in a terminal case $\Sigma\cup\{t\leq s\},\varepsilon \supset \varepsilon(t)\leq\varepsilon(s)$.

Subcase $(assmp_A)$. $\Sigma\cup\{t\leq s\},\varepsilon \supset a\leq b$, where $\varepsilon(t)=a$, $\varepsilon(s)=b$, and $a\leq b\in\Sigma$.
Then $a,b\notin Dom(\varepsilon)$ (by definition of Tenv), and $\langle t,\varepsilon\rangle=\mu t.a=a$, $\langle s,\varepsilon\rangle=\mu s.b=b$.
By $(assmp_R)$, $a\leq b\in\Sigma \Rightarrow \vdash_R \Sigma\supset a\leq b$.
Conclude by (eq_R): $\vdash_R \Sigma\supset \mu t.a\leq a$, $\vdash_R \Sigma\supset b\leq\mu s.b$, and $(trans_R)$.

Subcase $(\bot_A)$. $\Sigma\cup\{t\leq s\},\varepsilon \supset \bot\leq\varepsilon(s)$, where $\varepsilon(t)=\bot$.
Then $\langle t,\varepsilon\rangle=\mu t.\bot=\bot$ and we have $\vdash_R \Sigma\supset \bot\leq\langle s,\varepsilon\rangle$.
Conclude by (eq_R) and $(trans_R)$.

Subcase ($\top_A$). Similar.

Subcase (var_A). $\Sigma\cup\{t\le s\},\varepsilon \supset a\le a$, where $\varepsilon(t)=\varepsilon(s)=a$ and $a\notin \mathrm{Dom}(\varepsilon)$.

Then $\langle t,\varepsilon\rangle=\mu t.a=a=\mu s.a=\langle s,\varepsilon\rangle$. We can apply (eq_R): $\Sigma \supset a\le a$,

then conclude with (eq_R) and (trans_R).

Case k>0.

The tree has a (μ_A) root with a $(\to_A)$ child, hence $\varepsilon(t)=t_1\to t_2$, $\varepsilon(s)=s_1\to s_2$, where by definition of Tenv $t_1,t_2,s_1,s_2\in\mathrm{Dom}(\varepsilon)$:

$\Sigma\cup\{t\le s\},\varepsilon \supset s_1\le t_1,\ \Sigma\cup\{t\le s\},\varepsilon \supset t_2\le s_2$

$\Rightarrow\ \Sigma\cup\{t\le s\},\varepsilon \supset t_1\to t_2 \le s_1\to s_2$

$\Rightarrow\ \Sigma,\varepsilon \supset t\le s$

We initially focus on one of the subgoals of depth k-1:

(A) $\Sigma\cup\{t\le s\},\varepsilon\supset t_2\le s_2$

Let us consider the following goal (B), which we intend to subject, instead of (A), to the induction hypothesis:

(B) $\Sigma\cup\{t\le s\}, \varepsilon'\supset \sigma(t_2)\le\sigma(s_2)$

where $\sigma\triangleq[t'/t, s'/s]$ is a substitution with fresh variables t' and s', and $\varepsilon' \triangleq\sigma(\varepsilon\backslash t\backslash s)\cup\{t'=t, s'=s\}$.

First we show that the goal (B) is initial. Since $\vdash_A \Sigma,\varepsilon\supset t\le s$ is initial we have:

$\mathrm{Vars}(\Sigma)\cap\mathrm{Dom}(\varepsilon)=\emptyset$

$\varepsilon=\varepsilon_1\cup\varepsilon_2$ with $\mathrm{Dom}(\varepsilon_1)\cap\mathrm{Dom}(\varepsilon_2)=\emptyset$, such that $t\in\mathrm{Dom}(\varepsilon_1)$, $s\in\mathrm{Dom}(\varepsilon_2)$

Hence we also have:

$t_1,t_2\in\mathrm{Dom}(\varepsilon_1)$ (only); $s_1,s_2\in\mathrm{Dom}(\varepsilon_2)$ (only)

$\varepsilon'=\varepsilon'_1\cup\varepsilon'_2$ where $\varepsilon'_1\triangleq\sigma(\varepsilon_1\backslash t)\cup\{t'=t\}$, $\varepsilon'_2\triangleq\sigma(\varepsilon_2\backslash s)\cup\{s'=s\}$

From which we conclude:

$\mathrm{Vars}(\Sigma\cup\{t\le s\})\cap\mathrm{Dom}(\varepsilon')=\emptyset$

$\mathrm{Dom}(\varepsilon'_1)\cap\mathrm{Dom}(\varepsilon'_2)=\emptyset$

$\sigma(t_2)\in\mathrm{Dom}(\varepsilon'_1)$, because:

if $t_2\equiv t$ then $\sigma(t_2)\equiv t'$ and $t'\in\mathrm{Dom}(\varepsilon'_1)$; ($t_2\equiv s$ is not possible)

if $t_2\not\equiv t$ then $\sigma(t_2)\equiv t_2$; since $t_2\in\mathrm{Dom}(\varepsilon_1)$, we have $\sigma(t_2)\in\mathrm{Dom}(\varepsilon'_1)$

$\sigma(s_2)\in\mathrm{Dom}(\varepsilon'_2)$, similarly.

Second, let Tree(A) be the execution subtree of root (A), and Tree(B) be the execution tree of root (B). We show, by induction on the *length* of the longest path in Tree(A), that we can build a tree T such that: (1) T has the same *depth* as Tree(A); (2) T succeeds; (3) T expands the same variables as Tree(A) in (μ_A) nodes, with the exception of t',s'; (4) T has the one-expansion property; and (5) T = Tree(B). (Hence, we also have $\vdash_A$ (B).)

We proceed by induction on each subgoal $\underline{A} = \underline{\Sigma}\cup\{t\le s\},\varepsilon\supset\underline{\alpha}\le\underline{\beta}$ of Tree(A), for which we build a subtree $\underline{T}$ of the shape $\underline{\Sigma}\cup\{t\le s\},\varepsilon'\supset\sigma(\underline{\alpha})\le\sigma(\underline{\beta})$.

For the case (assmp_A) we have $\underline{\Sigma}\cup\{t\le s\},\varepsilon\supset\underline{t}\le\underline{s}$ with $\underline{t}\le\underline{s}\in\underline{\Sigma}\cup\{t\le s\}$. By the properties of one-expansion noted in 5.4, we only need to consider the cases when either $\underline{t}\equiv t$ and $\underline{s}\equiv s$, or $\underline{t},\underline{s},t,s$ are pairwise distinct.

If $\underline{t} \equiv t$, $\underline{s} \equiv s$ then Tree($\underline{A}$) is $\underline{\Sigma} \cup \{t \leq s\}, \varepsilon \supset t \leq s$, and $\underline{T}$ is taken to be $\underline{\Sigma} \cup \{t \leq s, t' \leq s'\}, \varepsilon' \supset t \leq s \Rightarrow \underline{\Sigma} \cup \{t \leq s\}, \varepsilon' \supset t' \leq s'$, which is successful by (assmp_A) and (μ_A), and has one-expansion. This $\underline{T}$ is longer but it still has depth 0.

If $\underline{t}, t, \underline{s}, s$ are pairwise distinct then Tree($\underline{A}$) is $\underline{\Sigma} \cup \{\underline{t} \leq \underline{s}, t \leq s\}, \varepsilon \supset \underline{t} \leq \underline{s}$, and $\underline{T}$ is taken to be $\underline{\Sigma} \cup \{\underline{t} \leq \underline{s}, t \leq s\}, \varepsilon' \supset \underline{t} \leq \underline{s}$ which is successful by (assmp_A), has one-expansion, and has depth 0.

For the case (μ_A) we must have, by one-expansion, $\underline{t}, t, \underline{s}, s$ pairwise distinct. Then Tree($\underline{A}$) has the shape:

$$\underline{\Sigma} \cup \{\underline{t} \leq \underline{s}, t \leq s\}, \varepsilon \supset \varepsilon(\underline{t}) \leq \varepsilon(\underline{s}) \Rightarrow \underline{\Sigma} \cup \{t \leq s\}, \varepsilon \supset \underline{t} \leq \underline{s} \quad \text{with } \underline{t}, \underline{s} \in \text{Dom}(\varepsilon).$$

Now $\sigma(\underline{t}) \equiv \underline{t}$, $\sigma(\underline{s}) \equiv \underline{s}$, and since $\underline{t}, \underline{s} \in \text{Dom}(\varepsilon')$ we have $\varepsilon'(\underline{t}) = \sigma(\varepsilon(\underline{t}))$, $\varepsilon'(\underline{s}) = \sigma(\varepsilon(\underline{s}))$. The tree $\underline{T}$ is then chosen with the shape:

$$\underline{\Sigma} \cup \{\underline{t} \leq \underline{s}, t \leq s\}, \varepsilon' \supset \sigma(\varepsilon(\underline{t})) \leq \sigma(\varepsilon(\underline{s})) \Rightarrow \underline{\Sigma} \cup \{t \leq s\}, \varepsilon' \supset \underline{t} \leq \underline{s}$$

hence preserving success and depth by (μ_A) and the induction hypothesis. One-expansion is preserved because, by induction hypothesis, $\underline{T}$ expands the same variables in (μ_A) node as Tree($\underline{A}$), which has one-expansion; except that t',s' are expanded in the (assmp_A) case, but in the present situation $\underline{t}, \underline{s}, t', s'$ are distinct.

The other cases do not pose difficulties. One-expansion for the ($\rightarrow_A$) case follows from one-expansion of the two branches, since one-expansion is defined path-wise.

Hence we can apply the induction hypothesis to (B), obtaining:

$$\vdash_R \Sigma \cup \{t \leq s\} \supset \langle \sigma t_2, \varepsilon' \rangle \leq \langle \sigma s_2, \varepsilon' \rangle$$

Then, by the equivalences $\langle \sigma t_2, \varepsilon' \rangle =_R \langle t_2, \varepsilon \backslash t \rangle$, $\langle \sigma s_2, \varepsilon' \rangle =_R \langle s_2, \varepsilon \backslash s \rangle$, and ($\text{eq}_R$), ($\text{trans}_R$):

$$\vdash_R \Sigma \cup \{t \leq s\} \supset \langle t_2, \varepsilon \backslash t \rangle \leq \langle s_2, \varepsilon \backslash s \rangle$$

By a similar argument on $\Sigma \cup \{t \leq s\}, \varepsilon \supset s_1 \leq t_1$ we obtain:

$$\vdash_R \Sigma \cup \{t \leq s\} \supset \langle s_1, \varepsilon \backslash s \rangle \leq \langle t_1, \varepsilon \backslash t \rangle$$

Finally:

$$\begin{aligned}
&\Rightarrow \Sigma \cup \{t \leq s\} \supset \langle t_1, \varepsilon \backslash t \rangle \rightarrow \langle t_2, \varepsilon \backslash t \rangle \leq \langle s_1, \varepsilon \backslash s \rangle \rightarrow \langle s_2, \varepsilon \backslash s \rangle && (\rightarrow_R) \\
&\Leftrightarrow \Sigma \cup \{t \leq s\} \supset \langle \varepsilon(t), \varepsilon \backslash t \rangle \leq \langle \varepsilon(s), \varepsilon \backslash s \rangle \\
&\Rightarrow \Sigma \supset \mu t. \langle \varepsilon(t), \varepsilon \backslash t \rangle \leq \mu s. \langle \varepsilon(s), \varepsilon \backslash s \rangle && (\mu_R) \\
&\Leftrightarrow \Sigma \supset \langle t, \varepsilon \rangle \leq \langle s, \varepsilon \rangle && \square
\end{aligned}$$

5.4.4 Example

We describe how to associate a proof tree to the execution tree with one expansion property built in 5.4.2, by repeatedly applying the inductive proof just presented. For convenience we rewrite here the execution tree:

$$\begin{array}{ccc}
 & & (\top_A) \qquad\qquad\qquad (\text{assmp}_A) \\
 & & \Rightarrow \{r_1 \leq u_1, r_3 \leq u_3\}, \theta \supset u_6 \leq r_6 \qquad \Rightarrow \{r_1 \leq u_1, r_3 \leq u_3\}, \theta \supset r_3 \leq u_3 \\
 & (\bot_A) & \Rightarrow \{r_1 \leq u_1, r_3 \leq u_3\}, \theta \supset r_6 \rightarrow r_3 \leq u_6 \rightarrow u_3 \\
\Rightarrow \{r_1 \leq u_1\}, \theta \supset u_4 \leq r_4 & & \Rightarrow \{r_1 \leq u_1\}, \theta \supset r_3 \leq u_3 \\
 & \Rightarrow \{r_1 \leq u_1\}, \theta \supset r_4 \rightarrow r_3 \leq u_4 \rightarrow u_3 & \\
 & \Rightarrow \emptyset, \theta \supset r_1 \leq u_1 &
\end{array}$$

where $\theta = \theta_1 \cup \theta_2$, with:

$\theta_1 = \{r_1 = r_4 \to r_3,\ r_4 = \top,\ r_3 = r_6 \to r_3,\ r_6 = \top\}$,
$\theta_2 = \{u_1 = u_4 \to u_3,\ u_4 = \bot,\ u_3 = u_6 \to u_3,\ u_6 = u_6 \to u_6\}$

Proceeding from the root we first fall on an inductive case. Hence we reapply the procedure to the modified subgoals:

$\{r_1 \le u_1\}, \theta' \supset u_4 \le r_4$ $\qquad$ $\{r_1 \le u_1\}, \theta' \supset r_3 \le u_3$

where: $\theta' = \theta'_1 \cup \theta'_2$, $\sigma = [r'/r_1, u'/u_1]$,

$\theta'_1 = \sigma(\theta_1 \backslash r_1) \cup \{r' = r_1\} = \{r_4 = \top,\ r_3 = r_6 \to r_3,\ r_6 = \top,\ r' = r_1\}$,
$\theta'_2 = \sigma(\theta_2 \backslash u_1) \cup \{u' = u_1\} = \{u_4 = \bot,\ u_3 = u_6 \to u_3,\ u_6 = u_6 \to u_6,\ u' = u_1\}$.

The first modified subgoal, $\{r_1 \le u_1\}, \theta' \supset u_4 \le r_4$, leads to a subcase ($\bot_A$). Hence we have:

$\vdash_R \{r_1 \le u_1\} \supset \langle u_4, \theta' \rangle \le \langle r_4, \theta' \rangle,\ \langle u_4, \theta' \rangle = \bot,\ \langle r_4, \theta' \rangle = \top$ (a)

The second modified subgoal, $\{r_1 \le u_1\}, \theta' \supset r_3 \le u_3$, leads again to an inductive case. Hence we generate two new modified subgoals:

$\{r_1 \le u_1, r_3 \le u_3\}, \theta'' \supset u_6 \le r_6$ $\qquad$ $\{r_1 \le u_1, r_3 \le u_3\}, \theta'' \supset r_3 \le u_3$

where: $\theta'' = \theta''_1 \cup \theta''_2$, $\sigma = [r''/r_3, u''/u_3]$,

$\theta''_1 = \sigma(\theta'_1 \backslash r_3) \cup \{r'' = r_3\} = \{r_4 = \top,\ r_6 = \top,\ r' = r_1,\ r'' = r_3\}$,
$\theta''_2 = \sigma(\theta'_2 \backslash u_3) \cup \{u'' = u_3\} = \{u_4 = \bot,\ u_6 = u_6 \to u_6,\ u' = u_1,\ u'' = u_3\}$.

The first modified subgoal, $\{r_1 \le u_1, r_3 \le u_3\}, \theta'' \supset u_6 \le r_6$, leads to a subcase ($\top_A$). Hence we have:

$\vdash_R \{r_1 \le u_1, r_3 \le u_3\} \supset \langle u_6, \theta'' \rangle \le \langle r_6, \theta'' \rangle$, (b)
$\langle u_6, \theta'' \rangle = \mu u_6.u_6 \to u_6$, $\langle r_6, \theta'' \rangle = \top$

The second modified subgoal, $\{r_1 \le u_1, r_3 \le u_3\}, \theta'' \supset r_3 \le u_3$, leads to a subcase ($assmp_A$). Hence we have:

$\vdash_R \{r_1 \le u_1, r_3 \le u_3\} \supset \langle r_3, \theta'' \rangle \le \langle u_3, \theta'' \rangle$, (c)
$\langle r_3, \theta'' \rangle = r_3,\ \langle u_3, \theta'' \rangle = u_3$

We can now build the proof tree, bottom up, using the proofs (a), (b), (c) as leaves:

$\{r_1 \le u_1, r_3 \le u_3\} \supset \mu u_6.u_6 \to u_6 \le \top$ $\qquad$ $\{r_1 \le u_1, r_3 \le u_3\} \supset r_3 \le u_3$
$\{r_1 \le u_1, r_3 \le u_3\} \supset \top \to r_3 \le (\mu u_6.u_6 \to u_6) \to u_3$
$\{r_1 \le u_1\} \supset \bot \le \top$ $\qquad$ $\{r_1 \le u_1\} \supset \mu r_3.(\top \to r_3) \le \mu u_3.((\mu u_6.u_6 \to u_6) \to u_3)$
$\{r_1 \le u_1\} \supset (\top \to \mu r_3.(\top \to r_3)) \le (\bot \to \mu u_3.((\mu u_6.u_6 \to u_6) \to u_3))$
$\emptyset \supset \mu r_1.(\top \to \mu r_3.(\top \to r_3)) \le \mu u_1.(\bot \to \mu u_3.((\mu u_6.u_6 \to u_6) \to u_3))$

It just remains to observe the following equivalences to get back to the types we started with in 5.4.2:

$\mu r_1.(\top \to \mu r_3.(\top \to r_3)) =_R \top \to \mu r_3.(\top \to r_3) =_R \mu r_3.(\top \to r_3)$, and
$\mu u_1.(\bot \to \mu u_3.((\mu u_6.u_6 \to u_6) \to u_3)) =_R \bot \to \mu u_3.((\mu u_6.u_6 \to u_6) \to u_3) =_R$
$\bot \to (\mu u_6.u_6 \to u_6)$. $\square$

5.4.5 Theorem *(Completeness of the subtyping rules)*

If $\alpha \leq_T \beta$ then $\alpha \leq_R \beta$.

Proof

If $\alpha \leq_T \beta$ then $\alpha \leq_A \beta$ by completeness of the algorithm (4.3.7). Consider the corresponding successful execution tree and apply the lockstep recursion lemma 5.4.1, obtaining a tree for $\alpha' \leq_A \beta'$ with $\alpha =_T \alpha'$ and $\beta =_T \beta'$. By lemma 5.4.3 we can now extract from the new execution tree a proof of $\alpha' \leq_R \beta'$. Applying the completeness of the rules for type equivalence we conclude $\alpha =_R \alpha'$ and $\beta =_R \beta'$. Finally we derive $\alpha \leq_R \beta$ by (eq_R) and $(trans_R)$. □

6. A Per Model

We sketch the main features of a model described in [1] (see also [14] for a related work) based on *complete uniform pers* over a D_∞ λ-model [26].

Per (partial equivalence relation) models provide an interpretation of subtyping as set-theoretic containment of the relations [7]. In addition, these structures have very interesting categorical properties (in particular cartesian closure and interpretation of second-order quantification as intersection, see [19]) that entail a satisfying interpretation of higher-order typed λ-calculi. The particular class of pers considered here preserves the previous properties while providing a solution of recursive domain equations up to equality. This result is obtained by an application of Banach's theorem on the uniqueness of the fixpoint of a contractive operator over a complete metric space.

6.1 Realizability Structure

Consider the functor $G(D) \triangleq D^D + D \times D + At$ defined in the category of complete partial orders (cpo's) and projection pairs. The cpo At is a collection of atomic values, and + is the coalesced sum. The morphism part of G is standard.

The cpo D_∞ is the initial fixpoint of the functor G, that is the colimit of the following ω-diagram:

$$D_0 \triangleq O \qquad \text{(O is the initial object; the cpo with one element)}$$
$$D_{n+1} \triangleq D_n^{D_n} + D_n \times D_n + At = G(D_n)$$

with uniquely determined projection pairs $(i_{n,n+1}, j_{n+1,n}) : D_n \to D_{n+1}$.

Let (i_n, j_n) be the projection pair between D_n and D_∞. Let $e_n \triangleq i_n(j_n(e))$ for $e \in D_\infty$. We have $\bigsqcup_{n<\omega}\{e_n\} = e$, where "$\bigsqcup$" denotes, as usual, the join. The cpo's $D_\infty^{D_\infty}$ and $D_\infty \times D_\infty$ are projected into D_∞ by means of the projection pairs: (i, j) and ([,], p). The operation of application on D_∞ is defined as usual as: $fd \triangleq j(f)(d)$.

6.2 Complete Uniform Pers

A per A over D_∞ is complete and uniform[3] (henceforth cuper) iff

[3] A term suggested by M. Abadi and G. Plotkin.

(1) $(\bot_{D_\infty}, \bot_{D_\infty}) \in A$ ($\bot_{D_\infty}$ is the least element of the cpo D_∞)
(2) If $X \subseteq A$ is directed in $D_\infty \times D_\infty$ then $\bigsqcup X \in A$
(3) If $(e,e') \in A$ then $\forall n.\ (e_n, e'_n) \in A$

We will consider the *full* subcategory of complete and uniform pers, therefore the *morphisms* are defined as usual as:

$$\text{cuper}[A, B] \triangleq \{f: D_\infty/A \to D_\infty/B \mid \exists \phi \in D_\infty. \forall d \in D_\infty.\ (d,d) \in A \Rightarrow \phi d \in f([d]_A)\ \}$$
where $[d]_A \triangleq \{e \in D_\infty \mid (d, e) \in A\}$, and $D_\infty/A \triangleq \{[d]_A \mid (d, d) \in A\}$

Let $A_{|n} \triangleq A \cap i_n(D_n) \times i_n(D_n)$. Given A, B cupers we can define as for ideals (see [20]):

closeness: $c(A,B) \triangleq \infty$, if $A=B$; $\max\{n \mid A_{|n} = B_{|n}\}$, o.w.
distance: $d(A,B) \triangleq 0$, if $c(A,B) = \infty$; $2^{-c(A,B)}$, o.w.

6.2.1 Subtype Interpretation

Following [11] and [7] we say that the cuper A is a subtype of the cuper B iff $A \subseteq B$. This is easily shown to correspond to the existence of a unique map in the category that is *realized* by the identity. Such maps play the role of *coercions* from A to B.

6.2.2 Type Interpretation

A *type environment* η is a map from type variables to cupers: $\eta: \text{Tvar} \to \text{cuper}$. A *type interpretation* of a type α in an environment η is written as $[\![\alpha]\!]\eta$.

In view of the interpretation of subtyping, the interpretation of type variables and type constants is naturally given as follows:

$$[\![\bot]\!]\eta \triangleq \{(\bot_{D_\infty}, \bot_{D_\infty})\} \qquad [\![\top]\!]\eta \triangleq D_\infty \times D_\infty \equiv \text{Top} \qquad [\![t]\!]\eta \triangleq \eta(t)$$

As we already mentioned, cuper is a cartesian closed category. In particular, given A, B cupers the *exponent* B^A is defined as follows:

$$(f, g) \in B^A \iff \forall\ d,e.\ (d, e) \in A \Rightarrow (fd, ge) \in B$$

This interpretation of the arrow is sometime referred to as *simple.*

In general, every object exp(A, B) *isomorphic* to the simple interpretation will enjoy the same categorical properties. Therefore, we assume exp is a binary operator on cupers satisfying:

$$\exp(A, B) \cong B^A$$

However, not any choice will be satisfying from our point of view. In order to complete the interpretation we need two more properties of the operator exp, namely, *contractiveness* and *(anti-)monotonicity*.

6.2.3 Contractiveness

The set of cupers endowed with the metric d is a *complete metric space.* We require that the behavior of exp at level n+1 is determined by the value of the arguments up to level n:

$$\exp(A, B)_{|n+1} = \exp(A_{|n}, B_{|n})_{|n+1}$$

Under this condition the exponentiation operator is *contractive* on the space (cuper, d) as it satisfies the following property:

$$A|_n = A'|_n \ , \ B|_n = B'|_n \ \Rightarrow \ \exp(A, B)|_{n+1} = \exp(A', B')|_{n+1}.$$

It turns out that every definable type operator is either contractive or the identity, and therefore admits a least fixpoint. The type-interpretation w.r.t. a contractive exponent exp(A, B) is completed as follows:

$$[\![\alpha\to\beta]\!]\eta \triangleq \exp([\![\alpha]\!]\eta, [\![\beta]\!]\eta) \quad [\![\mu t.\alpha]\!]\eta \triangleq \mathrm{Lfp}(\lambda A.[\![\alpha]\!]\eta_{[A/t]}) \ \text{(Lfp≡least fixpoint).}$$

6.2.4 Soundness of the (→) subtyping rule

In order to have a sound interpretation of the (→) rule in 3.1 it is convenient that the operator exp satisfies the following additional condition:

$$A' \subseteq A, \ B \subseteq B' \ \Rightarrow \ \exp(A, B) \subseteq \exp(A', B')$$

Proviso

We can summarize our discussion as follows. We assume to have a binary operator, exp: cuper×cuper→cuper, satisfying the following three properties, for any A, A', B, B':

$$\exp(A, B) \cong B^A$$
$$\exp(A, B)|_{n+1} = \exp(A|_n, B|_n)|_{n+1}$$
$$A' \subseteq A, \ B \subseteq B' \ \Rightarrow \ \exp(A, B) \subseteq \exp(A', B')$$

The *simple* interpretation defined above provides an example of such operator. The F-interpretation discussed in 6.3 provides yet another example.

We can interpret the types parametrically in the operator exp as follows:

$$[\![\bot]\!]\eta \triangleq \{(\bot_{D_\infty}, \bot_{D_\infty})\} \qquad [\![\top]\!]\eta \triangleq D_\infty \times D_\infty \equiv \mathrm{Top} \qquad [\![t]\!]\eta \triangleq \eta(t)$$

$$[\![\alpha\to\beta]\!]\eta \triangleq \exp([\![\alpha]\!]\eta, [\![\beta]\!]\eta) \quad [\![\mu t.\alpha]\!]\eta \triangleq \mathrm{Lfp}(\lambda A.[\![\alpha]\!]\eta_{[A/t]}) \ \text{(Lfp≡least fixpoint).}$$

The three conditions above are also *sufficient* to obtain the following soundness theorem. We write $\vDash \alpha\le\beta$ iff, given any operator exp, with relative type-interpretation $[\![\ ]\!]$, we have $[\![\alpha]\!]\eta \subseteq [\![\beta]\!]\eta$ for any η: Tvar→cuper.

We also write $\vDash \Gamma \supset \alpha\le\beta$. As usual this means: $\forall\eta.\ (\eta\vDash\Gamma \Rightarrow \eta\vDash\alpha\le\beta)$.

6.2.5 Theorem *(Soundness of the tree ordering w.r.t. the model)*

Given α, β types, if $\alpha\le_T\beta$ then $\vDash\alpha\le\beta$.

Proof (sketch)

Given a per A we define its *completion* cmpl(A) as the least cuper that contains A:

$$\mathrm{cmpl}(A) \ \triangleq \ \bigcap \{B \text{ cuper} \mid A \subseteq B\}$$

Given a tree A in Tree(L) we define its interpretation as the completion of the set-theoretic union of the interpretations of its syntactic approximants:

$$[\![A]\!]\eta \ \triangleq \ \mathrm{cmpl}(\bigcup_{k<\omega}[\![A|_k]\!]\eta)$$

It is easy to observe that $\{[\![A_{\lfloor k}]\!]\eta \mid k<\omega\}$ is a growing chain of cupers.

Now we need the following fact (see [1]):

$$\forall n,\alpha.\ \exists N.\ \forall k\geq N.\ [\![(\alpha)_n]\!]\eta = [\![(\alpha_{\lfloor k})_n]\!]\eta$$

where by definition $[\![(\beta)_n]\!]\eta \triangleq [\![\beta]\!]\eta \cap i_n(D_n)\times i_n(D_n)$.

In other words, if we are interested in the interpretation of the type α up to the n-th level of the construction of D_∞ , it is enough to unfold α up to a certain level N and just consider the interpretation of this *finite* part of the associated tree expansion.

Next we use the fact that $[\![\alpha]\!]\eta = \mathrm{cmpl}(\bigcup_{n<\omega}[\![(\alpha)_n]\!]\eta)$. From this we can conclude $[\![\alpha]\!]\eta \subseteq [\![T\alpha]\!]\eta$.

Vice versa observe that $\forall k.\ [\![\alpha_{\lfloor k}]\!]\eta \subseteq [\![\alpha]\!]\eta$. Hence $[\![\alpha]\!]\eta = [\![T\alpha]\!]\eta$.

Finally, $T\alpha \leq_\infty T\beta \ \Rightarrow\ \forall k.\ (\alpha_{\lfloor k} \leq \beta_{\lfloor k}) \ \Rightarrow\ \forall k.\ [\![\alpha_{\lfloor k}]\!]\eta \subseteq [\![\beta_{\lfloor k}]\!]\eta \ \Rightarrow\ [\![\alpha]\!]\eta \subseteq [\![\beta]\!]\eta$. □

6.2.6 Proposition *(Soundness of the rule ordering w.r.t. the model)*

If $\vdash_R \Gamma \supset \alpha\leq\beta$ then $\models \Gamma \supset \alpha\leq\beta$.

Proof

For the soundness of the type equivalence rules (5.1) one observes that the contractiveness of α in t is a sufficient (and necessary) condition to enforce the contractiveness of the following functional on the space $\mathrm{cuper}_{D_\infty}$(6.2):

$$G_{\alpha,\eta,t}(A) \triangleq [\![\alpha]\!]\eta[A/t] \qquad A\in \mathrm{cuper}_{D_\infty}$$

As for the subtyping rules (5.3) the problem is to check the soundness of (μ_R). Suppose $\eta\models\Gamma$. By hypothesis we have:

$$\forall A,B \text{ cuper.}\ A \subseteq B \ \Rightarrow\ G_\alpha(A) \triangleq [\![\alpha]\!]\eta[A/t] \subseteq [\![\beta]\!]\eta[B/t] \triangleq G_\beta(B)$$

Therefore we have: $\forall n.\ G_\alpha^n(\mathrm{Bot}) \subseteq G_\beta^n(\mathrm{Bot})$, where $\mathrm{Bot} = \{(\bot_{D_\infty}, \bot_{D_\infty})\}$.

It can be proved (see [1]) that for any type γ:

$$[\![(\mu t.\gamma)_n]\!]\eta \triangleq [\![\mu t.\gamma]\!]\eta \cap D_n\times D_n = G\gamma^n(\mathrm{Bot}) \cap D_n\times D_n$$

And from $[\![\mu t.\gamma]\!]\eta = \mathrm{cmpl}(\bigcup_{n<\omega}[\![(\mu t.\gamma)_n]\!]\eta)$ we have the thesis. □

6.3 Completeness of an F-interpretation

We now consider an *F-interpretation* of $\rightarrow$ (see [27]) that is isomorphic to the simple interpretation and still satisfies the properties in 6.2.3 and 6.2.4. We will also use this interpretation for the completeness theorem 6.3.4.

Define: $$(B^A)_F \triangleq B^A \cap F^2 \cup \{(\bot, f), (f, \bot), (f, f)\}$$

where F is the embedding of the functional space $D_\infty{}^{D_\infty}$ into D_∞ and f is the embedding of a distinct symbol of At into D_∞.

Roughly speaking $(B^A)_F$ is built from B^A by selecting among those elements that are "functions" in the underlying λ-model D_∞ and by attaching to $\bot$ a label f. We introduce the label f in order to distinguish the functional type $\top\rightarrow\bot$ from $\bot$ (see lemma 6.3.3). As an exercise one can try to give the complete rules for the "pure" version of the F-interpretation: $(B^A)_{F'} \triangleq B^A \cap$

F^2. A more difficult exercise is to define a complete system for the simple semantics. In this case further identifications like $\mu t.t\to t = \top$ take place.

6.3.1 F-theory of subtyping

Rather than giving some abstract definition of model that naively reflects the conditions for the soundness theorem and look for some ad hoc completeness result, we prefer to concentrate on a specific interpretation.

As a typical example, we *characterize the subtypings valid in every F-interpretation.* We write $\vDash_F \alpha\le\beta$ iff for any type structure M constructed as just described, we have $[\![\alpha]\!]\eta \subseteq [\![\beta]\!]\eta$ (or equivalently $\eta\vDash_F \alpha\le\beta$) with respect to the induced F-interpretation and for any type environment η.

In order to prove the completeness of the theory it will be enough to use the *elementary substructure* of ideals. Ideals are cupers with just one equivalence class; they are closed w.r.t. the standard operations over cupers.

Consider the type $\alpha\to\top$. Both in the simple interpretation and in the F-interpretation its meaning is essentially *independent from* α (this is not clearly the case for the tree equivalence).

In particular in the F-interpretation one has:

$$(\Phi) \qquad \alpha\to\beta \le \gamma\to\top$$

where $\gamma\to\top$ plays the role of *supertype of all the functional types* as:

$$[\![\gamma\to\top]\!]\eta \;=\; [\![\bot\to\top]\!]\eta \;=\; F^2 \cup \{(\bot, f), (f, \bot), (f, f)\}$$

Add to the subtyping system in 3.1 the axiom (Φ). Denote with $\vdash_\Phi$ formal derivability in this new system. Write $\alpha\le_\Phi\beta$ iff $\vdash_\Phi \alpha\le\beta$.

By examining the twenty five possible combinations of rules and axioms it turns out that the relation $\le_\Phi$ on the collection of non-recursive types is a preorder (as in 3.1, one shows the transitive rule is derived by case analysis).

Next, extend the preorder $\le_\Phi$ to recursive types by defining an ordering $\le_{\Phi\infty}$ on trees as: $A \le_{\Phi\infty} B$ iff $\forall k.\ (A_{\restriction k}\le_\Phi B_{\restriction k})$. Also define: $\alpha\le_{\Phi_T}\beta$ iff $T\alpha\le_{\Phi\infty}T\beta$.

6.3.2 Lemma

Let α be a recursive type and η be a type environment. If $T\alpha\neq\top$ and for each type variable t free in α we have $\eta(t)\neq$ Top then $[\![\alpha]\!]\eta \neq$ Top.

Proof

By induction on the structure of α. In particular, if $\alpha \equiv \mu t.\beta$ then either $T\alpha=\bot$ and the interpretation is the least cuper, or $T\alpha=t$ and we can use the hypothesis on η, or the interpretation is a cuper A that solves the equation:

$$A = (G_1(A)^{G_2(A)})_F \quad \text{for some definable operators } G_1 \text{ and } G_2 .$$

This forces $A\subset$Top. □

6.3.3 Lemma *(Separation)*

Suppose D_∞ is an algebraic cpo. There is a type environment η such that whenever α (β) matches an element of the column (row) then $[\![\alpha]\!]\eta \subseteq [\![\beta]\!]\eta$ iff the situation described at the corresponding intersection occurs:

$\leq$	$\bot$	$\top$	s	$\bot\to\top$	$\alpha'\to\beta'$ ($\beta'\neq\top$)
$\bot$	yes	yes	yes	yes	yes
$\top$	no	yes	no	no	no
t	no	yes	if $t\equiv s$	no	no
$\bot\to\top$	no	yes	no	yes	no
$\alpha\to\beta$ ($\beta\neq\top$)	no	yes	no	yes	$\alpha'\leq\alpha,\beta\leq\beta'$

Proof

As we already mentioned, it will be enough to consider ideals, that is, subsets of D_∞ with particular closure properties.

Let us choose an environment η s.t. $\eta(t) = \{\bot, \lambda_t\}$ where λ_t is an element of the flat cpo At. Of course $t\neq s \Rightarrow \lambda_t\neq\lambda_s$ and $\lambda_t\neq f$.

The only interesting problem is to show that in the case $(\alpha\to\beta,\alpha'\to\beta')$ the condition $\alpha'\leq\alpha$, $\beta\leq\beta'$ is in fact necessary.

First observe that $[\![\beta]\!]\eta \subseteq [\![\beta']\!]\eta$. Otherwise pick up $d\in[\![\beta]\!]\eta\setminus[\![\beta']\!]\eta$ and consider the constant map $\lambda x.d$ that belongs to $[\![\alpha\to\beta]\!]\eta\setminus[\![\alpha'\to\beta']\!]\eta$.

On the other hand, since $\beta'\neq\top$ by lemma 6.3.2 $\exists e\in Top\setminus[\![\beta']\!]\eta$. If the set $[\![\alpha']\!]\eta\setminus[\![\alpha]\!]\eta$ is not empty then it contains a compact element d_o. Consider the continuous function $step_{d_o,e}$ that evaluates to e for elements greater than or equal to d_o, and to $\bot$ otherwise. Then such a function belongs to $[\![\alpha\to\beta]\!]\eta\setminus[\![\alpha'\to\beta']\!]\eta$.

Note that we use the downward closure property of ideals to prove that the elements greater than or equal to d_o do not belong to $[\![\alpha]\!]\eta$. □

6.3.4 Proposition *(Completeness for $\leq_{\Phi_T}$)*

Given recursive types α and β, $\vDash_F \alpha\leq\beta$ iff $\alpha\leq_{\Phi_T}\beta$.

Proof

The soundness follows from the discussion in 6.3.1 and the more general soundness result presented in 6.2.5.

For showing completeness, consider the type structure and the type environment η in lemma 6.3.3. Given α,β, we want to show $\forall k.\ \alpha_{|k} \leq_\Phi \beta_{|k}$ whenever $\eta \vDash \alpha\leq\beta$.

Observe that the relation $\leq_{\Phi_T}$ is invariant under unfolding and under transformations of types of the shape $\alpha\to\top$ to $\bot\to\top$.

Fix k and unfold the types so that no μ appears before the k-th level. Transform all the subtypes of the shape $\alpha\to\top$ in $\bot\to\top$.

Proceed by induction on k to show that the conditions in the table 6.3.3 force $\alpha_{|k} \leq_\Phi \beta_{|k}$. □

In the remaining part of the paper we will be concerned with the tree ordering as it is very simple to analyze and it is valid in every interpretation satisfying the conditions of theorem 6.2.5. However, the previous study suggests that the tree ordering is very close to the model ordering so that, for example, the decision algorithm that we discuss in section 4 for the former can be easily adapted to the latter.

7. Coercions

Coercions and subtyping are closely related topics; see for example [3], [6]. We now show that the standard coercions $c_{\alpha,\beta}$ between two types $\alpha \leq \beta$ are definable in an extension of the basic calculus. This can be interpreted as saying that subtyping does not add any expressive power to such calculus (only convenience).

Then we show that the coercions implicit in a calculus with subsumption can be automatically synthesized. This fact is related to an algorithm for inferring the minimum type of a term.

7.1 Definability.

In this section we show how to associate with each successful execution tree a λ-term whose denotation in the model is a coercion, that is, the unique map between the corresponding types that is realized by the identity.

7.1.1 Building the λ-term.

We can show that if we consider types up to tree equivalence, $=_T$, then for every initial goal $\Sigma,\varepsilon \supset t \leq s$ such that $\vdash_A \Sigma,\varepsilon \supset t \leq s$ there is a term $M(x_1, ..., x_n) : \langle t \to s, \varepsilon \rangle$ where $\Sigma = \{t_1 \leq s_1, ..., t_n \leq s_n\}$ and x_i (i=1,..,n) are the free variables of M of type $\langle t_i \to s_i, \varepsilon \rangle$.

For the sake of readability the type labels on bound variables and on the fold and unfold constants are often omitted.

We recall that it is possible to define a *fixpoint combinator* as follows:
$\mathbf{Y} \equiv \lambda f^{\alpha\to\alpha}.(\lambda x^{\mu t.t\to\alpha}.\ f((\text{unfold } x)x))\ (\text{fold}(\lambda x^{\mu t.t\to\alpha}.\ f\ ((\text{unfold } x)x))) : (\alpha\to\alpha)\to\alpha$.

Proceed by induction on the structure of the execution tree (see 4.2.1). We refer to 4.1.7 for the properties 1..6, of the translation $\langle - , - \rangle$:

Case (assmp) $x^{\langle t\to s,\varepsilon\rangle}$.

Case ($\bot$) $\lambda x^{\bot}.\ \mathbf{Y}(\lambda x^{\langle\beta,\varepsilon\rangle}.\ x)\ :\ \langle \bot\to\beta,\varepsilon\rangle =_T \bot\to\langle\beta,\varepsilon\rangle$ by 1,5.

Case ($\top$) $\lambda x^{\langle\alpha,\varepsilon\rangle}.\ \mathbf{Y}(\lambda x^{\top}.\ x)\ :\ \langle\alpha\to\top,\varepsilon\rangle =_T \langle\alpha,\varepsilon\rangle\to\top$ by 2,5.

Case (var) $\lambda x^{a}.\ x\ :\ \langle a\to a,\varepsilon\rangle =_T a\to a$ by 3,5.

Case ($\to$) $\lambda f^{\langle\alpha\to\beta,\varepsilon\rangle}.\ \lambda x^{\langle\alpha',\varepsilon\rangle}.\ M_2(f(M_1(x)))\ :\ \langle(\alpha\to\beta)\to(\alpha'\to\beta'),\varepsilon\rangle$ by 5.
where by induction hypothesis $M_2 : \langle\beta\to\beta',\varepsilon\rangle$ and $M_1 : \langle\alpha'\to\alpha,\varepsilon\rangle$.

Case (μ) by induction hypothesis we have $M(x^{\langle t\to s,\varepsilon\rangle})\ :\ \langle\varepsilon(t)\to\varepsilon(s),\ \varepsilon\rangle$;
by 4, 5 $\langle t\to s,\varepsilon\rangle =_T \langle\varepsilon(t)\to\varepsilon(s),\ \varepsilon\rangle$ therefore we can type a term:
$\mathbf{Y}(\lambda y^{\langle\varepsilon(t)\to\varepsilon(s),\ \varepsilon\rangle}.\ M(y))\ :\ \langle\varepsilon(t)\to\varepsilon(s),\ \varepsilon\rangle =_T \langle t\to s,\varepsilon\rangle$, by 4.

Remark

In a similar fashion one can associate a λ-term with a proof of the judgment $\Gamma \supset \alpha \leq \beta$ in the system in 5.3 . The only difficulty arises for the rule (μ_R). Suppose we have inductively built a term $M(x^{t \to s}) : \alpha(t) \to \beta(s)$ then it is possible to transform it into a term $M'(x^{\mu t.\alpha \to \mu s.\beta}) : [\mu t.\alpha/t]\alpha \to [\mu s.\beta/s]\beta$. The term associated with the conclusion of the (μ_R) rule can be defined as:

$\mathbf{Y}(\lambda x^{\mu t.\alpha \to \mu s.\beta}.\ \lambda y^{\mu t.\alpha}.\ (\text{fold } (M''(x)(\text{unfold } y))))$

7.1.2 Proposition *(Coercions are definable)*

Let $\alpha, \beta \in$ Type and suppose $\alpha \leq_A \beta$. Let M be the term associated in 7.1.1 with the execution tree of $\emptyset, E\alpha \cup E\beta \supset \alpha^* \leq \beta^*$. Then the denotation of the term in the model is the unique coercion map from the interpretation of α to the interpretation of β.

Proof.

Since we have not given the term interpretation explicitly (see [1]), we can only sketch an idea of the proof.

In the first place we need some facts about the interpretation of terms:

(a) By erasing the type information and the constants fold, unfold from a typed term M, we obtain an untyped λ-term *er*(M). We denote these untyped λ-terms with P, Q, ... It is a basic property of these interpretations that the interpretation of *er*(M) gives a representative for the equivalence class that corresponds to the interpretation of M. We shortly refer to this fact by saying that *er*(M) is a realizer for M.

(b) Showing that the interpretation of M is a coercion from α to β means proving that the identity map, id, is a realizer for M. Equivalently id and *er*(M) are equivalent in $\alpha \to \beta$. Note that here and in the following for the sake of readability we simply refer to syntactic objects but we really intend to speak of their denotations in the model.

(c) The realizer for **Y** is an element Fix with functionality: $\lambda g. \bigsqcup g^n(\bot_{D_\infty})$.

In order to prove the theorem by induction on the structure of the execution tree one needs to generalize somewhat.

In the first place one observes that if in the execution tree of $\emptyset, \varepsilon \supset t \leq s$ we never use (assmp) then the interpretation of the associated term $M : \langle t \to s, \varepsilon \rangle$ is a coercion in $\langle t \to s, \varepsilon \rangle$.

However, this is not enough to make the induction go through in the case where the term $M(x^{\Sigma})$ really depends on the assumption variable. One has to observe that $M(x^{\Sigma})$ also enjoys a property of *contractiveness*.

Let us suppose that (μ) is the last rule applied. By construction assume we have a term M(x) that is a functional from coercions to coercions. We would like to show that $\mathbf{Y}(\lambda x.M(x))$ is still a coercion.

Observe that after a (μ) rule we always have a (→) rule. Therefore the term M(x) has the structure $\lambda f.\lambda y.M_2(x)(f(M_1(x)y))$.

Now observe that a realizer for $\mathbf{Y}(\lambda x.M(x))$ will be something like $\bigsqcup g^n(\bot_{D_\infty})$ for $g = \lambda x.\lambda f.\lambda y.P_2(x)(f(P_1(x)y))$ where P_i is a realizer for M_i (i=1,2). We have to show that this realizer is equivalent to id in a type with the structure $C \equiv (A \Rightarrow B) \Rightarrow (A' \Rightarrow B')$, where $A \Rightarrow B \equiv \exp(A, B)$. Since the type is a complete per, it will be enough to show that for each n $g^n(\bot_{D_\infty})$ is equivalent to id in the appropriate type.

To do this we need a last remark, Denote with $A_{|n}$ the approximation at the n-th level of the cuper A as in 6.2. One observes that if $(P, id) \in C_{|n}$ then $(g(P), id) \in C_{|n+1}$. This follows easily from the structure of g and the assumption (6.2.3). Hence we have $\forall n.\ (g^n(\bot_{D_\infty}), id) \in C_{|n}$ that implies $(\bigsqcup g^n(\bot_{D_\infty}), id) \in C$. □

7.2 Inference

Let $\lambda^{\to\mu}$ be the calculus in section 2. Given a term in $\lambda^{\to\mu}$, possibly not typeable, we are interested in the problem of determining if it can be *well-typed modulo the insertion of appropriate coercions.*

We refer to this problem as *coercion inference.* We will define a simple algorithm that, given a term M, succeeds exactly when M is typeable modulo the insertion of coercions. In this case the algorithm returns the *least type* among the types that can be assigned to M.

A similar problem was solved in [2] for a second-order lambda calculus with records, and in [18] for a second-order lambda calculus including a form of bounded quantification.

All these results rely on the *structural properties of the subtype relation* that are stated, in this case, as Proposition 7.2.4.

Notation.

In this section $\alpha \precsim \beta$ and $\alpha \simeq \beta$ are shorthands for $T\alpha \leq_\infty T\beta$ and $T\alpha = T\beta$.

7.2.1 Typing modulo coercions

We can formalize the idea of *typing modulo coercions* in two ways:

(a) Subsumption. Add to the typing system in 2.2 and 3.1 the following rule based on the tree order $\leq_\infty$. The version based on $\leq_{fin}$ is often referred to as Subsumption:

(Sub_∞) $\quad M: \alpha,\ \alpha \precsim \beta \ \Rightarrow\ M: \beta$

We denote formal derivability in this new system with $\vdash_{Sub}$.

(b) Explicit Coercions. Extend the term language with a collection of constants $\{c_{\alpha,\beta} \mid \alpha, \beta$ types$\}$ and add to the typing system in 3.1 the following rule:

$(ExpCoer_\infty)$ $\quad M: \alpha,\ \alpha \precsim \beta \ \Rightarrow\ (c_{\alpha,\beta} M): \beta$

Denote formal derivability in this new system with $\vdash_c$, and denote the corresponding term language with $\lambda^{\to\mu c}$. Moreover, denote with er_c (mnemonic for erase coercions) the obvious function that takes a term in $\lambda^{\to\mu c}$, erases all the constants $c_{\alpha,\beta}$, and returns a term in $\lambda^{\to\mu}$.

The use of these rules is justified by the finitary axiomatization of $\precsim$ given in section 5.

Note that in both these systems the $(fold_{\mu t.\alpha} M)$ and $(unfold_{\mu t.\alpha} M)$ terms become redundant.

7.2.2 Definition *(coercion inference)*

We define inductively on the structure of the term M in $\lambda^{\to\mu}$ a function[4]

CI: $(\lambda^{\to\mu}) \to (\lambda^{\to\mu c} \cup \{FAIL\})$ (CI for coercion inference)

that either fails or returns a well-typed term N in $\lambda^{\to\mu c}$ such that $er_c(N) \equiv M$.

The clauses (fold), (unfold) have priority on the clause (apl).

[4]Actually the following specification determines a *class* of algorithms that suffices for our purposes.

(var) $CI(x^{\alpha}) \triangleq x^{\alpha}$

(abs) $CI(\lambda x^{\alpha}.M) \triangleq$ if $CI(M) : \beta$ then $\lambda x^{\alpha}.CI(M)$ else FAIL

(apl) $CI(MN) \triangleq$
if $CI(M) : \alpha'$ and $CI(N): \gamma$ then
if $\alpha' \simeq \alpha \to \beta$ and $\gamma \precsim \alpha$ then $(c_{\alpha',\alpha\to\beta}\, CI(M))(c_{\gamma,\alpha}\, CI(N))$
else if $\alpha' \simeq \bot$ then $(c_{\alpha',\alpha\to\bot}\, CI(M))\, CI(N)$ else FAIL
else FAIL

(fold) $CI(fold_{\mu t.\alpha}\, M) \triangleq$
if $CI(M): \beta$ and $\beta \precsim \mu t.\alpha$ then
$fold_{\mu t.\alpha}(c_{\beta,[\mu t.\alpha/t]\alpha} CI(M))$
else FAIL

(unfold) $CI(unfold_{\mu t.\alpha}\, M) \triangleq$
if $CI(M): \beta$ and $\beta \precsim \mu t.\alpha$ then
$unfold_{\mu t.\alpha}(c_{\beta,\mu t.\alpha} CI(M))$
else FAIL □

Clearly CI can also be used to define an inference algorithm for $\vdash_{Sub}$; just consider the type of the term synthesized by CI. We prove in 7.2.5 that this algorithm computes the minimal type of a term (if any). To achieve this result we need the following simple properties.

7.2.3 Proposition

Let M be a term in $\lambda^{\to\mu}$ then:
(1) $\vdash_{Sub} M : \alpha$ iff for some N: $\vdash_c N : \alpha$ and $er_c(N) \equiv M$.
(2) If $CI(M): \beta$ then $er_c(CI(M)) \equiv M$.

Proof

(1) Every introduction of an explicit coercion corresponds to an application of subsumption and vice versa.

(2) By induction on the definition of CI. □

7.2.4 Proposition

Let $\alpha, \beta, \ldots$ be recursive types then:
(1) If $\alpha \precsim \beta_1 \to \beta_2$ then either $\alpha \simeq \bot$ or $\alpha \simeq \alpha_1 \to \alpha_2$, $\beta_1 \precsim \alpha_1$, and $\alpha_2 \precsim \beta_2$.
(2) If $\alpha_1 \to \alpha_2 \precsim \beta$ then either $\beta \simeq \top$ or $\beta \simeq \beta_1 \to \beta_2$, $\beta_1 \precsim \alpha_1$, and $\alpha_2 \precsim \beta_2$.

Proof

(1) α can be rewritten, by unfolding, to an equivalent type of the shape $\bot$, $\top$, t or $\alpha_1 \to \alpha_2$. The definition of the tree ordering and the hypothesis $\alpha \precsim \beta_1 \to \beta_2$ lead to the conclusion by a simple case analysis.

(2) Analogous. □

7.2.5 Theorem *(Terms have a least type)*

Let M be a term in $\lambda^{\to\mu}$ then $\vdash_{Sub} M : \alpha$ implies $CI(M):\beta$ and $\beta \precsim \alpha$.

Proof

By induction on the structure of M.

C(N) is a meta-notation for $c\alpha_{n-1,}\alpha_n$ (...($c\alpha_{1,}\alpha_2$ N)..), where: N: α_1, $n \geq 1$, $\alpha_i \precsim \alpha_{i+1}$.

By virtue of 7.2.3.(1) we may equivalently assume the existence of a well-typed term N in $\lambda^{\rightarrow\mu c}$ such that $er_c(N) \equiv M$.

Observe the crucial role of property 7.2.4 in proving the rather surprising fact that the algorithm is *complete* in the sense just stated above.

Case $M \equiv x^{\beta}$.

If $er_c(N) \equiv x^{\beta}$ then $N \equiv C\, x^{\beta} : \alpha$ and $\beta \precsim \alpha$. On the other hand $CI(x^{\beta}) \equiv x^{\beta}:\beta$.

Case $M \equiv (\lambda x\alpha.M')$.

If $er_c(N) \equiv \lambda x^{\alpha}.M'$ then $N \equiv C\,(\lambda x^{\alpha}.N'): \gamma$, $er_c(N') \equiv M'$, and $N':\beta'$.

By induction hypothesis $CI(M') : \beta$ and $\beta \precsim \beta'$, hence by definition. $CI(\lambda x^{\alpha}.M') : \alpha \rightarrow \beta$. Note that $\alpha \rightarrow \beta' \precsim \gamma$ by definition of N and this implies (by 7.2.4) either $\gamma \simeq \top$ (and in this case we are done as $\alpha \rightarrow \beta \precsim \top$) or $\gamma \simeq \gamma_1 \rightarrow \gamma_2$, $\gamma_1 \precsim \alpha$ and $\beta' \precsim \gamma_2$.

In the latter case $\beta \precsim \beta' \precsim \gamma_2$ implies $\alpha \rightarrow \beta \precsim \gamma$.

Case $M \equiv (M_1 M_2)$.

If $er_c(N) \equiv M_1 M_2$ then $N \equiv C(N_1 N_2): \gamma$, $er_c(N_i) \equiv M_i$ $i=1,2$, $N_1: \gamma_1 \rightarrow \gamma_2$, $N_2:\gamma_1$.

By induction hypothesis $CI(M_i) : \beta_i$ $i=1, 2$, $\beta_1 \precsim \gamma_1 \rightarrow \gamma_2$ and $\beta_2 \precsim \gamma_1$.

From (7.2.4) follows that $\beta_1 \simeq \bot$ or $\beta_1 \simeq \beta_1' \rightarrow \beta_1''$, $\gamma_1 \precsim \beta_1'$, $\beta_1'' \precsim \gamma_2$.

In the first case $CI(M_1 M_2) : \bot$ and we are done.

In the second $CI(M_1 M_2) : \beta_1''$ as $\beta_2 \precsim \gamma_1 \precsim \beta_1'$.

Finally observe: $\beta_1'' \precsim \gamma_2 \precsim \gamma$.

Case $M \equiv (\mathrm{fold}_{\mu t.\alpha}\, M')$.

If $er_c(N) \equiv \mathrm{fold}_{\mu t.\alpha}\, M'$ then $N \equiv C(\mathrm{fold}\ N'): \gamma$, $er_c(N') \equiv M'$, $N': [\mu t.\alpha/t]\alpha \equiv \gamma'$, $\gamma' \precsim \gamma$. By induction hypothesis $CI(M'): \beta'$, $\beta' \precsim \gamma'$. Hence by definition, $CI(\mathrm{fold}\ M'): \mu t.\alpha$ and we have $\mu t.\alpha \simeq \gamma' \precsim \gamma$.

Case $M \equiv (\mathrm{unfold}_{\mu t.\alpha}\, M')$.

Analogous. □

Remarks

7.2.6 One can think of substituting the explicit coercions with the definable coercions constructed in section 7.1. The resulting term is now typeable in an extension of the calculus in section 2 including the rule: $M:\alpha,\ \alpha=\beta \Rightarrow M:\beta$.

We recall that this rule is soundly interpreted by the model.

7.2.7 Observe that in general there are many possible well-typed terms of the same type to which the erase-coercions map assigns the *same term*, that is:

$\vdash_c N_1 : \alpha$, $\vdash_c N_2 : \alpha$ and $er_c(N_1) \equiv er_c(N_2)$

However, N_1 and N_2 receive the same interpretation in the model.

It is an appealing aspect of our semantic approach to the interpretation of subtyping that many hard coherence problems (see [18]) simply disappear by recalling the uniqueness of the coercion in the model.

7.2.8 The following is a trivial example of a term that can be typed in the type system with subsumption but *not* in the system described in 2.2: $\vdash_{Sub} \lambda f^{t\to s}.\ \lambda x^{t}.\ (\lambda y^{\top}.\ x)(f\ x): (t\to s)\to t$.

8. Conclusion

We have used a subtyping relation based on infinite trees as the central concept of our work. In our experience this relation has arisen naturally, giving insights about both the subtypings valid in certain per-models and the behavior of the Amber implementation. In fact we have shown that this relation can be used to characterize sound and complete theories for a certain class of per models and that it can be simply and efficiently implemented. We have also shown the soundness and completeness of certain rules and the definability of coercions within the calculus (modulo a strengthening of the notion of type equality). Finally, we have observed that the whole process of inferring coercions and minimal types can be automated.

In conclusion, let us consider the problem of the extension of our results.

The notions of tree expansion and finite approximation (section 3) can be easily adapted to larger languages, both with first-order type constructors like products, sums, records and variants, and with higher-order type constructors like second-order universal quantification. The important point is that the tree resulting from the expansion is regular. Under this assumption it seems possible to adapt algorithms and rules to obtain results of soundness and completeness (sections 4, 5). Caution is necessary in extensions to bounded quantification since some of those systems are undecidable [24].

About the relationship between the tree ordering and the model, we expect the extension of the soundness theorem (6.2) to be straightforward. On the other hand we expect technical problems from the completeness theorem (6.3) when introducing higher-order type constructors like second-order universal quantification. In particular, in this case, it is not clear how to extend the separation lemma (6.3.3).

The result on the definability of the coercions has already been obtained for several calculi with records, variants, and bounded quantification (but without recursion). It is a reassuring result that shows that the subtyping theory is in good harmony with the calculus.

The fact that terms have a least type has a clear impact on the implementation of the type-checker. This appears to be a very desirable property towards an automatic treatment of coercions. The result, at the present state of the art, clearly relies on the *structural properties* of the subtyping relation.

Finally, we observe that challenging extensions arise when dealing with non-ground collections of subtyping assumptions (see [3]). In this case much work remains to be done.

9. Acknowledgments

We would like to thank Martín Abadi for comments on an early draft, and the referees for their efforts in improving the presentation of the paper.

References

[1] Amadio, R. **Recursion over realizability structures**, Information and Computation., 91, 1, pp 55-85, 1991. Preliminary version appeared as TR1/89, Dipartimento di Informatica, Università di Pisa.

[2] Amadio, R. **Formal theories of inheritance for typed functional languages**, TR 28/89, Dipartimento di Informatica, Università di Pisa, 1989.

[3] Amadio, R. **Typed equivalence, type assignment and type containment**, in Proc. Conditional and Typed Rewriting Systems 90, eds. Kaplan&Okada, Lecture Notes in Computer Science, vol. 516, Springer-Verlag.

[4] Arnold, A., and Nivat, M. **The metric space of infinite trees. Algebraic and topological properties**, Fundamenta Informaticae III, pp 445-476, 1980.

[5] Breazu-Tannen, V., Coquand T., Gunter, C., and Scedrov, A. **Inheritance as implicit coercion**, Information and Computation, Vol 93, pp 172-221, 1991.

[6] Breazu-Tannen, V., Gunter, C., and Scedrov, A. **Denotational semantics for subtyping between recursive types**, Report MS-CIS 89 63, Logic of Computation 12, Dept of Computer & Information Science, University of Pennsylvania, 1989.

[7] Bruce, K., and Longo, G. **A modest model of records, inheritance and bounded quantification**, Information and Computation, Vol 87, No 1/2, pp 196-240, 1990.

[8] Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J.C. **F-bounded polymorphism for object-oriented programming**, in Proc. Functional Programming and Computer Architecture 89, 1989.

[9] Cardelli, L., and Wegner, P. **On understanding types, data abstraction and polymorphism**, Computing Surveys, 17, 4, pp 471-522, December 1985.

[10] Cardelli, L. **Amber**, Combinators and Functional Programming Languages, Proc. of the 13th Summer School of the LITP, Le Val D'Ajol, Vosges (France), May 1985. Lecture Notes in Computer Science, vol. 242, Springer-Verlag.

[11] Cardelli, L. **A semantics of multiple inheritance**, Info.&Comp., 76, pp138-164 (Preliminary version in LNCS 173, Springer-Verlag, 1984).

[12] Cardelli, L. **Typeful programming**, in Formal Description of Programming Concepts, E.J.Neuhold and M.Paul Eds., pp 431-507, Springer-Verlag, 1991.

[13] Cardelli, L., and Longo, G. **A semantic basis for Quest**, Journal of Functional Programming, Vol 1, Part 4, pp 417-458, Cambridge University Press, Oct 1991.

[14] Cardone, F., and Coppo, M. **Type inference with recursive types: syntax and semantics**, Info.&Comp., 92, 1, pp 48-80.

[15] Cook, W. **A denotational semantics of inheritance**, Ph.D. thesis, Brown University, 1989.

[16] Courcelle, B. **Fundamental properties of infinite trees**, Theoretical Computer Science, 25, pp 95-169, 1983.

[17] Courcelle, B. **Equivalence and transformation of regular systems - applications to recursive program schemes and grammars**, Theoretical Computer Science, 42, pp 1-122, 1986.

[18] Curien, P.-L., and Ghelli, G. **Coherence of Subsumption, minimum typing and type-checking in $F_{\leq}$**, Mathematical Structures in Computer Science, vol 2, pp 55-91, 1992.

[19] Hyland, M. **A small complete category**, Annals of Pure and Applied Logic 40, 2, pp 135-165, 1989.

[20] MacQueen, D., Plotkin, G., and Sethi, R. **An ideal model for recursive polymorphic types**, Info.&Comp., 71, 1-2,1986.

[21] Milner, R. **A complete inference system for a class of regular behaviours**, Journal of Computer and System Science, 28, pp. 439-466, 1984.

[22] G.Nelson (ed.): **Systems Programming with Modula-3**, Prentice Hall, 1991.

[23] Park, D.M.R. **Concurrency and automata on infinite sequences**, Proc. 5th GI conference, Lecture Notes in Computer Science, vol. 104, pp. 167-183, Springer-Verlag, 1981.

[24] Pierce, B.C. **Bounded Quantification is Undecidable**, Proc ACM Symposium on Principles of Programming Languages, pp 305-315, 1992.

[25] Salomaa, A. **Two complete systems for the algebra of regular events**, Journal of ACM, 13,1, 1966.

[26] Scott, D. **Continuous lattices**, Toposes, Algebraic Geometry and Logic, Lawvere (ed.), Lecture Notes in Mathematics 274, pp 97-136, Springer-Verlag, 1972.

[27] Scott, D. **Data types as lattices**, SIAM J. of Computing, 5, pp 522-587, 1976.

[28] van Wijngaarden et al. ed., **Revised report on the algorithmic language Algol68**, pp 103-107, Springer-Verlag, 1976.

[29] Wadsworth, C. **The relation between computational and denotational properties for Scott's D_{∞} models of the lambda-calculus**, SIAM J. of Computing, 5, pp 488-521, 1976.