Recursive Subtyping Revealed

Functional Pearl

Vladimir Gapeyev Michael Y. Levin Benjamin C. Pierce

Department of Computer and Information Science
University of Pennsylvania

{vgapeyev,milevin,bcpierce}@cis.upenn.edu

ABSTRACT

Algorithms for checking subtyping between recursive types lie at the core of many programming language implementations. But the fundamental theory of these algorithms and how they relate to simpler declarative specifications is not widely understood, due in part to the difficulty of the available introductions to the area. This tutorial paper offers an "end-to-end" introduction to recursive types and subtyping algorithms, from basic theory to efficient implementation, set in the unifying mathematical framework of coinduction.

1. INTRODUCTION

Recursively defined types in programming languages and lambda-calculi come in two distinct varieties. Consider, for example, the type `X` described by the equation

$$\mathtt{X} \;=\; \mathtt{Nat}\to(\mathtt{Nat}\times\mathtt{X}).$$

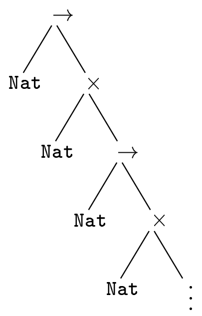
An element of `X` is a function that maps a number to a pair consisting of a number and a function of the same form. This type is often written more concisely as $\mu\mathtt{X}.\mathtt{Nat}\to(\mathtt{Nat}\times\mathtt{X})$. A variety of familiar recursive types such as lists and trees can be defined analogously.

In the **iso-recursive** formulation, the type $\mu\mathtt{X}.\mathtt{Nat}\to(\mathtt{Nat}\times\mathtt{X})$ is considered *isomorphic* to its one-step unfolding, $\mathtt{Nat}\to(\mathtt{Nat}\times(\mu\mathtt{X}.\mathtt{Nat}\to(\mathtt{Nat}\times\mathtt{X})))$. The term language provides a pair of built-in coercion functions for each recursive type $\mu\mathtt{X}.\mathtt{T}$,

$$\begin{array}{lcl} \mathtt{unfold} & \in & \mu\mathtt{X}.\mathtt{T} \to \{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T} \\ \mathtt{fold} & \in & \{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T} \to \mu\mathtt{X}.\mathtt{T} \end{array}$$

witnessing the isomorphism (as usual, $\{\mathtt{X}\mapsto\mathtt{S}\}\mathtt{T}$ denotes the substitution of `S` for free occurrences of `X` in `T`).

In the **equi-recursive** formulation, on the other hand, a recursive type and its one-step unfolding are considered *equivalent*—interchangeable for all purposes. In effect, the equi-recursive treatment views a type like $\mu\mathtt{X}.\mathtt{Nat}\to(\mathtt{Nat}\times\mathtt{X})$ as merely an abbreviation for the infinite tree obtained by unrolling the recursion "out to infinity":[1]

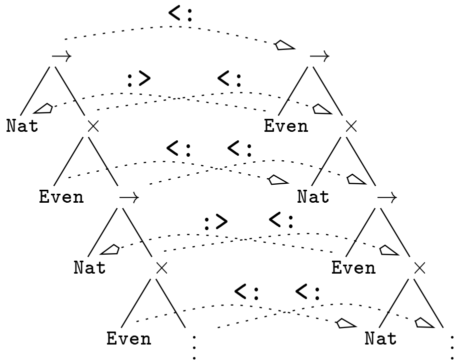


The equi-recursive view can make terms easier to write, since it saves annotating programs with `fold` and `unfold` coercions, but it raises some tricky problems for the compiler, which must deal with these infinite structures and operations on them in terms of appropriate finite representations. Moreover, in the presence of these infinite types, even the *definitions* of other features such as subtyping can become hard to understand. For example, supposing that the type `Even` is a subtype of `Nat`, what should be the relation between the types $\mu\mathtt{X}.\mathtt{Nat}\to(\mathtt{Even}\times\mathtt{X})$ and $\mu\mathtt{X}.\mathtt{Even}\to(\mathtt{Nat}\times\mathtt{X})$?

The simplest way to think through such questions is often to view them "in the limit." In the present example, the elements inhabiting both types can be thought of as simple reactive processes: given a number, they return another number plus a new process that is ready to receive another number, and so on. Processes belonging to the first type always yield even numbers and are capable of accepting arbitrary numbers. Those belonging to the second type yield arbitrary numbers, but expect always to be given even numbers. The constraints both on what arguments the function must accept and on what results it may return are more demanding for the first type, so intuitively we expect the first to be a subtype of the second. We can draw a picture summarizing our calculations as follows:

[1]The two different formulations of recursive types have been around since the beginning, but the pleasantly mnemonic terms "iso-recursive" and "equi-recursive" are a relatively new coinage, due to Crary, Harper, and Puri [9].

ICFP 2000 Montreal, Canada



Can such arguments be made precise? Indeed they can. The basic ideas can be found in several places, going back to Amadio and Cardelli's comprehensive study [3], which remains the standard reference in the area. Unfortunately, the available literature is not as friendly to newcomers as might be wished. More recent treatments tend to be rather condensed, assuming that the reader is already familiar with some of the relevant intuitions. On the other hand, Amadio and Cardelli's original paper, while complete, is also quite complex and, in some technical respects, beginning to be slightly dated. More efficient subtyping algorithms are now known (e.g., [15, 6, 14]). Also, it is now widely agreed that framing definitions and proofs in terms of *coinduction* (rather than limits of sequences of approximations) substantially simplifies both intuitions and formalities.

Our purpose in this tutorial is not to announce new results, but rather to formulate known techniques as lucidly as possible, beginning from fundamental definitions and leading, by simple steps, to efficient algorithms for checking subtyping. We also try to make clear, at every point, the analogy between the coinductive structures we define and those found in the familiar, inductive world of finite types and ordinary subtyping.

We begin by reviewing the basic theory of inductive and coinductive definitions and their associated proof principles (Section 2). Sections 3 and 4 instantiate this general theory for the case of subtyping, defining both the familiar inductive subtyping relation on finite types and its coinductive generalization to infinite types. Section 5 makes a brief detour to consider some issues connected with the rule of transitivity (a notorious troublemaker in subtyping systems). Section 6 derives simple algorithms for checking membership in inductively and co-inductively defined sets; Section 7 considers more refined algorithms. These algorithms are applied to subtyping for the important special case of "regular" infinite trees in Section 8. Section 9 introduces μ-types as a finite notation for representing regular trees and presents a theorem that the more complex (but finitely realizable) subtyping relation on μ-types coincides with the ordinary coinductive definition of subtyping between regular trees. Section 10 brings together all the preceding material to derive a concrete subtyping algorithm for μ-types and proves its termination. Finally, Section 11 discusses a well-known simplified version of the algorithm and shows that has exponential behaviour. Several sections are accompanied by exercises for the reader; solutions to these can be found at the end of the paper.

In the interest of brevity, some of the less interesting proofs are omitted in this short version.

We deal with a very simple language of types, containing just arrow types, binary products, and a maximal `Top` type. Additional type constructors such as records, variants, etc., can be added with no changes to the basic theory. Binding constructs such as universal and existential quantifiers can also be formulated in the same framework [11], but they are trickier, since they require working with infinite trees "modulo renaming of bound variables." Constructs such as type operators that introduce nontrivial equivalences between type expressions pose additional problems.

No previous understanding of the metatheory of recursive types or background in the theory of coinduction is required, though the development will assume a certain degree of mathematical sophistication and some familiarity with type systems and subtyping.

2. INDUCTION AND COINDUCTION

Assume we have fixed some **universal set** $\mathcal{U}$ as the domain of discourse for our inductive and coinductive definitions. $\mathcal{U}$ represents the set of "everything in the world"; the role of an inductive or coinductive definition will be to pick out some subset of $\mathcal{U}$. (Later on, we are going to choose $\mathcal{U}$ to be the set of all pairs of types, so that subsets of $\mathcal{U}$ are relations on types. But for the present discussion, an arbitrary set $\mathcal{U}$ will do.) The powerset of $\mathcal{U}$, i.e., the set of all the subsets of $\mathcal{U}$, is written $\mathcal{P}(\mathcal{U})$.

2.1 Definition: A function $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ is **monotone** if $X \subseteq Y$ implies $F(X) \subseteq F(Y)$.

In what follows, we will assume that F is some monotone function on $\mathcal{P}(\mathcal{U})$. We refer to F as a **generating function**.

2.2 Definition: Let X be a subset of $\mathcal{U}$.

1. X is F-**closed** if $F(X) \subseteq X$.
2. X is F-**consistent** if $X \subseteq F(X)$.
3. X is a **fixed point** of F if $F(X) = X$.

A useful intuition for these definitions is to think of the elements of $\mathcal{U}$ as some sort of statements or assertions, and of F as representing a "justification" relation that, given some set of statements (premises), tells us what new statements (conclusions) follow from them. An F-closed set, then, is one that cannot be made any bigger by adding in new elements justified by F—it already contains all conclusions justified by its members. An F-consistent set is one that is "self-justifying": every assertion in it is justified by other assertions that are also in it. A fixed point of F includes precisely the justifications required by its members, the conclusions that follow from its members, and nothing else.

2.3 Example: Consider the following generating function on the three-element universe $\mathcal{U} = \{a, b, c\}$:

$$\begin{array}{lcl} E_1(\emptyset) & = & \{c\} \\ E_1(\{a\}) & = & \{c\} \\ E_1(\{b\}) & = & \{c\} \\ E_1(\{c\}) & = & \{b, c\} \\ E_1(\{a, b\}) & = & \{c\} \\ E_1(\{a, c\}) & = & \{b, c\} \\ E_1(\{b, c\}) & = & \{a, b, c\} \\ E_1(\{a, b, c\}) & = & \{a, b, c\} \end{array}$$

There is just one E_1-closed set—$\{a, b, c\}$—and four E_1-consistent sets—$\emptyset$, $\{c\}$, $\{b, c\}$, $\{a, b, c\}$.

This function can be represented compactly by a collection of **inference rules**:

$$\frac{}{c} \qquad \frac{c}{b} \qquad \frac{b \quad c}{a}$$

Each inference rule states that if all of the elements above the bar are in the input set, then the element below is in the output set. (We often omit the bar when a rule has no premises.)

2.4 Theorem [Knaster-Tarski [20]]: The intersection of all F-closed sets is the least fixed point of F. The union of all F-consistent sets is the greatest fixed point of F.

2.5 Definition: The least fixed point of F is written μF. The greatest fixed point of F is written νF.

Note that μF itself is F-closed (hence, it is the smallest F-closed set) and that νF is F-consistent (hence, it is the largest F-consistent set).

2.6 Example: For the sample generating function E_1 shown above, we have $\mu E_1 = \nu E_1 = \{a, b, c\}$.

2.7 Exercise: Suppose a generating function E_2 on the universe $\{a, b, c\}$ is defined by the following inference rules:

$$\frac{}{a} \qquad \frac{c}{b} \qquad \frac{a \quad b}{c}$$

Write out the set of pairs in the relation E_2 explicitly, as we did for E_1 above. List all the E_2-closed and E_2-consistent sets. What are μE_2 and νE_2?

An immediate consequence of the Knaster-Tarski theorem is the following pair of fundamental reasoning principles:

2.8 Corollary [of 2.4]:

1. **Principle of induction**: If X is F-closed, then $\mu F \subseteq X$.

2. **Principle of coinduction**: If X is F-consistent, then $X \subseteq \nu F$.

The intuition behind these principles comes from thinking of the set X as a predicate (represented as its characteristic set, the subset of $\mathcal{U}$ for which the property is true). Then showing that property X is true of an element x is the same as showing that x is in the set X. Now, the induction principle says that, if there is a property whose characteristic set X is closed under F (i.e., the property is preserved by F), then the property is true of all the elements of the inductively defined set μF. The coinduction principle, on the other hand, tells us how to prove that an element x is *in* the coinductively defined set νF. To prove $x \in \nu F$, it suffices to find a set X such that $x \in X$ and X is F-consistent.

We will use the principles of induction and coinduction heavily throughout the paper. (We will not write out every inductive argument in terms of generating functions and predicates; in the interest of brevity, we will sometimes fall back on familiar abbreviations such as structural induction.)

3. FINITE AND INFINITE TYPES

Having introduced coinduction, our next job will be to instantiate these definitions with the specifics of subtyping. Before we can do this, though, we need to define precisely how to view types as (finite or infinite) trees.

For brevity, we deal in this paper with just three type constructors: $\rightarrow$, $\times$, and `Top`. We define types as (possibly infinite) trees with nodes labeled by one of the symbols $\rightarrow$, $\times$, or `Top`. The definition is specialized to our present needs; for a general treatment of infinite labeled trees see [8].

We write $\{1, 2\}^*$ for the set of sequences of 1s and 2s. The empty sequence is written $\bullet$, and i^k stands for k copies of i. If π and σ are sequences, then $\pi \cdot \sigma$ denotes the concatenation of π and σ.

3.1 Definition: A **tree type** (or, simply, a **tree**) is a partial function $\mathtt{T} \in \{1, 2\}^* \rightharpoonup \{\rightarrow, \times, \mathtt{Top}\}$ satisfying the following constraints:

- $\mathtt{T}(\bullet)$ is defined
- if $\mathtt{T}(\pi \cdot \sigma)$ is defined then $\mathtt{T}(\pi)$ is defined
- if $\mathtt{T}(\pi) = \rightarrow$ or $\mathtt{T}(\pi) = \times$ then $\mathtt{T}(\pi \cdot 1)$ and $\mathtt{T}(\pi \cdot 2)$ are defined
- if $\mathtt{T}(\pi) = \mathtt{Top}$ then $\mathtt{T}(\pi \cdot 1)$ and $\mathtt{T}(\pi \cdot 2)$ are undefined.

A tree type `T` is **finite** if $dom(\mathtt{T})$ is finite. The set of all tree types is denoted $\mathcal{T}$; its subset, the set of all finite tree types, is $\mathcal{T}_f$.

The set of finite types can be defined more compactly by a grammar:

$$\begin{array}{lcl} \mathtt{T} & ::= & \mathtt{Top} \\ & & \mathtt{T}\times\mathtt{T} \\ & & \mathtt{T}\rightarrow\mathtt{T} \end{array}$$

The force of such a grammar is that the set of types `T` is the least fixed point of the evident generating function. The universe of this generating function is the set of *all* finite and infinite labeled trees (which can be defined along similar lines to 3.1). The full set of tree types can be derived from the same generating function by taking the greatest fixed point instead of the least fixed point.

For notational convenience, we write `Top` for the tree `T` such that $\mathtt{T}(\bullet) = \mathtt{Top}$. Similarly, when $\mathtt{T}_1$ and $\mathtt{T}_2$ are trees, we write $\mathtt{T}_1\times\mathtt{T}_2$ for the tree with $(\mathtt{T}_1\times\mathtt{T}_2)(\bullet) = \times$ and $(\mathtt{T}_1\times\mathtt{T}_2)(i \cdot \pi) = \mathtt{T}_i(\pi)$ and $\mathtt{T}_1\rightarrow\mathtt{T}_2$ for the tree such that $(\mathtt{T}_1\rightarrow\mathtt{T}_2)(\bullet) = \rightarrow$ and $(\mathtt{T}_1\rightarrow\mathtt{T}_2)(i \cdot \pi) = \mathtt{T}_i(\pi)$ for $i = 1, 2$.

3.2 Example: The type expression `(Top×Top)→Top` denotes the finite tree type `T` defined by the function with $\mathtt{T}(\bullet) = \rightarrow$ and $\mathtt{T}(1) = \times$ and $\mathtt{T}(2) = \mathtt{T}(1 \cdot 1) = \mathtt{T}(1 \cdot 2) = \mathtt{Top}$. Precise use of similar notation for non-finite types is problematic, since the corresponding linear terms would be infinitely long. In examples we informally use ellipsis for this purpose. For example, the expression `Top→(Top→(Top→...))` corresponds to the type `T` defined by $\mathtt{T}(2^k) = \rightarrow$, for all $k \geq 0$, and $\mathtt{T}(2^k \cdot 1) = \mathtt{Top}$, for all $k \geq 0$.

4. SUBTYPING

We define subtyping relations on finite tree types and on tree types in general as least and greatest fixed points, respectively, of monotone functions on certain universes. For subtyping on finite tree types the universe is the set $\mathcal{T}_f \times \mathcal{T}_f$ of pairs of finite tree types; our generating functions will map subsets of this universe—that is, relations on $\mathcal{T}_f$—to other subsets, and their fixed points will also be relations on $\mathcal{T}_f$. For subtyping on trees the universe is $\mathcal{T} \times \mathcal{T}$, pairs of arbitrary (finite or infinite) trees.

4.1 Definition [Finite subtyping]: Two finite tree types S and T are in the subtyping relation ("S is a subtype of T") if $(\mathtt{S},\mathtt{T}) \in \mu S_f$, where the monotone function $S_f \in (\mathcal{T}_f \times \mathcal{T}_f) \to (\mathcal{T}_f \times \mathcal{T}_f)$ is defined by

$$\begin{array}{rcl} S_f(R) & = & \{(\mathtt{T},\mathtt{Top}) \mid \mathtt{T} \in \mathcal{T}_f\} \\ & \cup & \{(\mathtt{S}_1{\times}\mathtt{S}_2, \mathtt{T}_1{\times}\mathtt{T}_2) \mid (\mathtt{S}_1,\mathtt{T}_1),\ (\mathtt{S}_2,\mathtt{T}_2) \in R\} \\ & \cup & \{(\mathtt{S}_1{\to}\mathtt{S}_2, \mathtt{T}_1{\to}\mathtt{T}_2) \mid (\mathtt{T}_1,\mathtt{S}_1),\ (\mathtt{S}_2,\mathtt{T}_2) \in R\}. \end{array}$$

This generating function precisely captures the effect of the usual definition of the subtyping relation by a collection of inference rules:

$$\mathtt{T} <: \mathtt{Top}$$

$$\frac{\mathtt{S}_1 <: \mathtt{T}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1{\times}\mathtt{S}_2 <: \mathtt{T}_1{\times}\mathtt{T}_2}$$

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1{\to}\mathtt{S}_2 <: \mathtt{T}_1{\to}\mathtt{T}_2}$$

In these rules, the pair of types $(\mathtt{S},\mathtt{T})$ is written $\mathtt{S} <: \mathtt{T}$. The statement $\mathtt{S} <: \mathtt{T}$ above the line in the second and third rules should be read as "if the pair $(\mathtt{S},\mathtt{T})$ is in the argument to S_f" and below the line as "then $(\mathtt{S},\mathtt{T})$ is in the result."

4.2 Definition [Infinite subtyping]: Two tree types S and T are in the subtyping relation if $(\mathtt{S},\mathtt{T}) \in \nu S$, where $S \in (\mathcal{T} \times \mathcal{T}) \to (\mathcal{T} \times \mathcal{T})$ is defined by:

$$\begin{array}{rcl} S(R) & = & \{(\mathtt{T},\mathtt{Top}) \mid \mathtt{T} \in \mathcal{T}\} \\ & \cup & \{(\mathtt{S}_1{\times}\mathtt{S}_2, \mathtt{T}_1{\times}\mathtt{T}_2) \mid (\mathtt{S}_1,\mathtt{T}_1),\ (\mathtt{S}_2,\mathtt{T}_2) \in R\} \\ & \cup & \{(\mathtt{S}_1{\to}\mathtt{S}_2, \mathtt{T}_1{\to}\mathtt{T}_2) \mid (\mathtt{T}_1,\mathtt{S}_1),\ (\mathtt{S}_2,\mathtt{T}_2) \in R\}. \end{array}$$

Note that the inference rule presentation of this relation is precisely the same as for the inductive relation above: all that changes is that we consider a larger universe of types and take a greatest instead of a least fixed point.

4.3 Exercise: Check that νS is not the whole of $\mathcal{T} \times \mathcal{T}$ by exhibiting a pair $(\mathtt{S},\mathtt{T})$ that is not in νS.

4.4 Exercise: Can you find a pair of types $(\mathtt{S},\mathtt{T})$ that are related by νS, but not by μS? What about a pair of types $(\mathtt{S},\mathtt{T})$ that are related by νS_f, but not by μS_f?

One fundamental fact about the subtype relation—transitivity—should be checked right away. (If the subtype relation were *not* transitive, the crucial property of preservation of types under reduction would immediately fail. To see this, suppose that there were types S, T, and U with S<:T and T<:U but not S<:U. Let s be a value of type S and f a function of type U→Top. Then the term (λx:T. f x) s can be typed, using the rule of subsumption once for each application, but reduces in one step to the ill-typed term f s.)

4.5 Definition: The **transitive closure** of a binary relation $R \subseteq \mathcal{U} \times \mathcal{U}$ is the least relation Q containing R and closed under transitivity: if $(\mathtt{S},\mathtt{U}) \in Q$ and $(\mathtt{U},\mathtt{T}) \in Q$, then $(\mathtt{S},\mathtt{T}) \in Q$. Equivalently, the transitive closure of R is the least fixed point, μTC_R, of the generating function

$$\begin{array}{l} TC_R(X) = \\ \quad R \cup \{(\mathtt{S},\mathtt{T}) \mid (\mathtt{S},\mathtt{U}),\ (\mathtt{U},\mathtt{T}) \in X \text{ for some } \mathtt{U} \in \mathcal{U}\}. \end{array}$$

The transitive closure of R is written R^+.

4.6 Lemma: If $R \subseteq \mathcal{T} \times \mathcal{T}$ is S-consistent, then so is its transitive closure R^+.

4.7 Corollary: νS is transitive.

Proof: Since νS is S-consistent, its transitive closure, $(\nu S)^+$, is S-consistent by Lemma 4.6. Therefore, $(\nu S)^+ \subseteq \nu S$ by the principle of coinduction. Since $\nu S \subseteq (\nu S)^+$ by the definition of transitive closure, we have $\nu S = (\nu S)^+$. The latter relation is obviously transitive.

4.8 Exercise: Show that the subtype relation is also reflexive.

5. A DIGRESSION ON TRANSITIVITY

Standard formulations of inductively defined subtyping relations generally come in two forms: a *declarative* presentation that is optimized for readability and an *algorithmic* presentation that corresponds more or less directly to an implementation. In simple systems, the two are generally similar; in more complex systems, they can be quite different, and proving that the two presentations define the same relation on types can sometimes pose a significant challenge.

One of the most distinctive differences between declarative and algorithmic presentations is that declarative presentations generally include an explicit rule of transitivity—if S<:U and U<:T then S<:T—while algorithmic systems never do. This rule is useless in an algorithm, since applying it in a goal-directed manner would involve guessing U.

The rule of transitivity plays two useful roles in declarative systems. First, it reassures the reader that the subtype relation is, indeed, transitive. In algorithmic presentations, transitivity is not obvious, but must be proved, as we did above. Second, transitivity often allows other rules to be stated in simpler, more primitive forms; in algorithmic presentations, these simple rules need to be combined into heavier mega-rules that take into account all possible combinations of the simpler ones. For example, in the presence of transitivity, the rules for "depth subtyping" within record fields, "width subtyping" by adding new fields, and "permutation" of fields can be stated separately, making them all easier to understand. Without transitivity, the three rules must be merged into a single rule taking width, depth, and permutation into account all at once.

Interestingly, the viability of a declarative presentation with a rule of transitivity is a consequence of a "trick" that can be played with inductive, but not coinductive, definitions. To see why, observe that transitivity is a *closure property*—it demands that the subtype relation be closed under the transitivity rule. Since the subtype relation itself is defined as the closure of a set of rules, we can achieve closure under transitivity simply by adding it to the main subtyping rules. This is a general property of inductive definitions and closure properties: given two relations, each defined inductively from sets of inference rules, the union of the two sets of rules will generate the least relation that is closed under both sets of rules. This fact can be formulated more abstractly in terms of generating functions:

5.1 Proposition: Suppose F and G are monotone functions, and let $H(X) = F(X) \cup G(X)$. Then μH is the smallest set that is both F-closed and G-closed.

Unfortunately, this trick does not work with coinductive definitions. As the following exercise shows, adding transitivity to the rules generating a coinductively defined relation does not give us the relation we want.

5.2 Exercise: Show that the generating function

$$\begin{array}{rcl} S^{tr}(R) & = & S(R) \\ & \cup & \{(\mathtt{S},\mathtt{T}) \mid (\mathtt{S},\mathtt{U}),\ (\mathtt{U},\mathtt{T}) \in R \text{ for some } \mathtt{U} \in \mathcal{T}\} \end{array}$$

is degenerate, in the sense that its greatest fixed point is the *total* relation $\mathcal{T} \times \mathcal{T}$.

In the coinductive setting, it appears we are stuck with algorithmic presentations and mega-rules.

6. MEMBERSHIP CHECKING

We now turn our attention to the question of how to decide, given a generating function F on some universe $\mathcal{U}$ and an element $x \in \mathcal{U}$, whether or not x falls in the greatest fixed point of F.

A given element $x \in \mathcal{U}$ can, in general, be generated by F in many ways. That is, there can be more than one set $X \subseteq \mathcal{U}$ such that $x \in F(X)$. Call any such set X a "generating set" for x. Because, due to monotonicity of F, any superset of a generating set for x is a generating set for x, it makes sense to restrict attention to minimal generating sets. We go even further and consider the class of "invertible" functions, where each x has at most one minimal generating set.

6.1 Definition: A generating function F is said to be **invertible** if, for all $x \in \mathcal{U}$, the collection of sets

$$G_x = \{X \subseteq \mathcal{U} \mid x \in F(X)\}$$

either is empty or contains a unique smallest member. For an invertible F, the partial function $support_F \in \mathcal{U} \rightharpoonup \mathcal{P}(\mathcal{U})$ is defined as follows[2]:

$$support_F(x) = \begin{cases} X & \text{if } X \in G_x \text{ and } \forall X' \in G_x.\ X \subseteq X' \\ \uparrow & \text{if } G_x = \emptyset \end{cases}$$

The function is lifted to sets as follows:

$$support_F(X) = \begin{cases} \bigcup_{x \in X} support_F(x) & \text{if } \forall x \in X.\ support_F(x) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

When F is clear from context, we will often omit the subscript in $support_F$ (and other F-based functions defined later).

From now on, we focus our attention on invertible generating functions.

6.2 Definition: An element x is **supported** if $support(x)$ is defined; otherwise, x is **unsupported**. A supported element is **ground** if $support(x) = \emptyset$.

Note that an unsupported element x cannot appear in $F(X)$ for any X, while a ground x is in every $F(X)$.

An invertible generating function can be visualized as a "support graph". For example, Figure 1 defines a function E on the universe $\{a, b, c, d, e, f, g, h, i\}$ by showing which elements are needed to support a given element of the universe: for a given x, its $support(x)$ consists of all y for which there is an arrow from x to y. An unsupported element is denoted by a slashed circle. In this example, i is the only unsupported element and g is the only ground element. (Note that, according to our definition, h *is* supported.)

6.3 Exercise: Write out a set of inference rules corresponding to this function, as we did in Example 2.3. Check that $E(\{b, c\}) = \{g, a, d\}$, that $E(\{a, i\}) = \{g, h\}$, and that the sets of elements marked in the figure as μE and νE are indeed the least and the greatest fixed points of E.

[2] We write $\uparrow$ throughout the paper to mean "undefined," and $f(x)\downarrow$ to mean "$f(x)$ is defined."

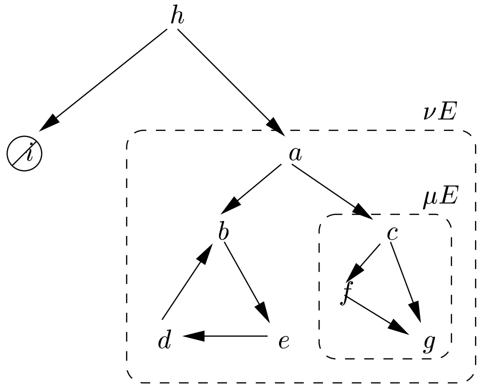


Figure 1: A sample *support* **function**

Looking at the example of Figure 1 might lead us to conjecture that an element x is in the greatest fixed point if and only if no unsupported element is reachable from x. This suggests an algorithmic strategy for checking whether x is in νF: enumerate all elements reachable from x via the *support* function; return failure if an unsupported element occurs in the enumeration; otherwise, succeed. Observe, however, that there can be cycles of reachability between the elements, and the enumeration procedure should take some precautions against falling into an infinite loop.

6.4 Definition: Suppose F is an invertible function. Define the function gfp_F (or just gfp) as follows:

$$\begin{array}{rcl} gfp(X) & = & \text{if } support(X)\uparrow, \text{ then } false \\ & & \text{else if } support(X) \subseteq X, \text{ then } true \\ & & \text{else } gfp(support(X) \cup X). \end{array}$$

Intuitively, gfp starts from X and uses *support* to enrich it until either it becomes consistent or an unsupported element is found.

The extension of gfp to single elements is given by $gfp(x) = gfp(\{x\})$.

6.5 Exercise: Another observation that is clear from Figure 1 is that an element x of νF is not a member of μF if x participates in a cycle in the support graph (or there is a path from x to an element that participates in a cycle). Is the converse also true? If x is a member of νF but not μF, is it necessarily the case that x leads to a cycle?

The rest of this section is devoted to proving correctness and termination of gfp. (First-time readers may want to skip this material and jump to the next section.) We start by observing a couple of properties of the *support* function.

6.6 Lemma: $X \subseteq F(Y)$ iff $support(X) \subseteq Y$.

6.7 Lemma: Suppose P is a fixed point of F. Then $X \subseteq P$ iff $support(X) \subseteq P$.

Now we can prove partial correctness of gfp.

6.8 Theorem:

1. If $gfp(X) = true$, then $X \subseteq \nu F$.
2. If $gfp(X) = false$, then $X \not\subseteq \nu F$.

Proof: The proof of each clause proceeds by induction on the recursive structure of a run of the algorithm.

1. From the definition of *gfp*, it is easy to see that there are two cases where $gfp(X)$ can return *true*. If $gfp(X) = true$ because $support(X) \subseteq X$, then, by Lemma 6.6, we have $X \subseteq F(X)$, i.e., X is F-consistent; thus, $X \subseteq \nu F$ by the coinduction principle. On the other hand, if $gfp(X) = true$ because $gfp(support(X) \cup X) = true$, then, by the induction hypothesis, $support(X) \cup X \subseteq \nu F$, and so $X \subseteq \nu F$.

2. Let $gfp(X) = false$ because $support(X)$ is undefined. Then $X \not\subseteq \nu F$ by Lemma 6.7. Let $gfp(X) = false$ because $gfp(support(X) \cup X) = false$. By the induction hypothesis, $support(X) \cup X \not\subseteq \nu F$. Equivalently, $X \not\subseteq \nu F$ or $support(X) \not\subseteq \nu F$. Either way, $X \not\subseteq \nu F$ (using Lemma 6.7 in the second case).

We are going to specify a sufficient termination condition for *gfp* by giving a class of generating functions for which the algorithm terminates. To describe the class, we need some additional terminology.

6.9 Definition: For a generating function F and an element $x \in \mathcal{U}$, the set of predecessors of x is

$$pred_F(x) = \begin{cases} \emptyset & \text{if } support_F(x)\uparrow \\ support_F(x) & \text{if } support_F(x)\downarrow \end{cases}$$

and its extension to sets $X \subseteq \mathcal{U}$ is

$$pred_F(X) = \bigcup_{x \in X} pred_F(x).$$

The set of all elements reachable from a set X via the $support_F$ is defined by the following function

$$reachable_F(X) = \bigcup_{n \geq 0} pred_F^n(X).$$

and its extension to single elements $x \in \mathcal{U}$ is

$$reachable_F(x) = reachable_F(\{x\}).$$

We say that an element $y \in \mathcal{U}$ is **F-reachable** from an element x if $y \in reachable_F(x)$.

6.10 Definition: A generating function F is said to be **finite state** if $reachable_F(x)$ is finite for each $x \in \mathcal{U}$.

Finite state functions form a class of generating functions for which *gfp* terminates:

6.11 Theorem: If $reachable(X)$ is finite, then $gfp(X)$ is defined. Consequently, if F is finite state, then $gfp(X)$ terminates for any finite $X \subseteq \mathcal{U}$.

Proof: For each possible recursive call $gfp(Y)$ in the call graph generated by the original invocation $gfp(X)$, we have $Y \subseteq reachable(X)$. Moreover, Y strictly increases on each call. Since $reachable(\mathrm{X})$ is finite, $m(Y) = |reachable(X)| - |Y|$ serves as a termination measure for *gfp*.

7. MORE EFFICIENT ALGORITHMS

Although the *gfp* algorithm is correct, it is not very efficient, since it has to recompute the *support* of the whole set X every time it makes a recursive call. Consider the following trace of *gfp* on the function E from Figure 1.

$$\begin{array}{ll} & gfp(\{a\}) \\ = & gfp(\{a,b,c\}) \\ = & gfp(\{a,b,c,e,f,g\}) \\ = & gfp(\{a,b,c,e,f,g,d\}) \\ = & true. \end{array}$$

Note that the algorithm recomputes $support(a)$ four times. We can refine the algorithm to eliminate this redundant recomputation by maintaining a set A of **assumptions** whose *support* sets have already been considered and a set X of **goals** whose *support* has not yet been considered.

7.1 Definition: Suppose F is an invertible function. Define the function gfp^a as follows (the superscript "a" is for "assumptions"):

$$\begin{array}{l} gfp^a(A, X) \quad = \\ \quad \text{if } support(X)\uparrow\text{, then } false \\ \quad \text{else if } X = \emptyset\text{, then } true \\ \quad \text{else } gfp^a(A \cup X,\ support(X) \setminus (A \cup X)). \end{array}$$

In order to check $x \in \nu F$, compute $gfp^a(\emptyset, \{x\})$.

This algorithm computes the support of each element at most once. A trace for the above example looks like this:

$$\begin{array}{ll} & gfp^a(\emptyset,\ \{a\}) \\ = & gfp^a(\{a\},\ \{b,c\}) \\ = & gfp^a(\{a,b,c\},\ \{e,f,g\}) \\ = & gfp^a(\{a,b,c,e,f,g\},\ \{d\}) \\ = & gfp^a(\{a,b,c,e,f,g,d\},\ \emptyset) \\ = & true. \end{array}$$

The correctness statement for this algorithm is slightly more elaborate than the ones we saw in the previous section:

7.2 Proposition:

1. If $support(A) \subseteq A \cup X$ and $gfp^a(A, X) = true$, then $A \cup X \subseteq \nu F$.
2. If $gfp^a(A, X) = false$, then $X \not\subseteq \nu F$.

To make the membership checking algorithm more similar to the known implemetations of recursive subtyping, we make more modifications. First, the order of computation is made more explicit by computing *support* for one element at a time. Second, we start using newly computed assumptions as soon as they become available by threading the set of assumptions through recursive calls. (Note that this makes the algorithm non-tail-recursive.)

7.3 Definition: Given an invertible function F, define the function gfp^t as follows ("t" stands for "threading")[3]:

$$\begin{array}{l} gfp^t(A, x) \quad = \\ \quad \text{if } x \in A\text{, then } A \\ \quad \text{else if } support(x)\uparrow\text{, then } \uparrow \\ \quad \text{else} \\ \quad\quad \text{let } x_1, \ldots, x_n = support(x) \text{ in} \\ \quad\quad \text{let } A_0 = A \cup \{x\} \text{ in} \\ \quad\quad \text{let } A_1 = gfp^t(A_0, x_1) \text{ in} \\ \quad\quad \ldots \\ \quad\quad \text{let } A_n = gfp^t(A_{n-1}, x_n) \text{ in} \\ \quad\quad A_n. \end{array}$$

[3]We use the following convention for undefinedness: if an expression B is undefined, then "let $A = B$ in C" is also taken to be undefined. This avoids the need to write explicit "exception handling" clauses for every recursive invocation of gfp^t.

To check $x \in \nu F$, compute $gfp^t(\emptyset, x)$: if the result is a set of assumptions then $x \in \nu F$; if the result is undefined, then $x \notin \nu F$.

The correctness statement must again be refined, taking into account the non-tail-recursive nature of this formulation by positing a "stack" X of elements whose supports must still be checked.

7.4 Proposition:

1. If $gfp^t(A, x) = A'$, then $A \cup \{x\} \subseteq A'$.
2. For all X, if $support(A) \subseteq A \cup X \cup \{x\}$ and $gfp^t(A, x) = A'$, then $support(A') \subseteq A' \cup X$.
3. If $gfp^t(A, x) \uparrow$, then $x \notin \nu F$.

7.5 Corollary: If $gfp^t(\emptyset, x) = A'$, then $x \in \nu F$.

Proof: By the first part of the proposition, $x \in A'$. Instantiating the second statement of the proposition with $X = \emptyset$, we obtain $support(A') \subseteq A'$—that is, A' is F-consistent, and so $A' \subseteq \nu F$ by coinduction.

Since all of the algorithms in this section examine the reachable set in one way or another, the termination condition for all of them is the same as that of the original *gfp* algorithm: they terminate on all inputs when F is finite state.

8. REGULAR TREES

At this point, we have developed generic algorithms for checking membership in a greatest fixed point, and separately shown how to define subtyping between infinite trees as the greatest fixed point of a particular generating function S. The obvious next step is to instantiate one of the algorithms with S. Of course, this concrete algorithm will not terminate on all inputs, since in general the set of states reachable from a given pair of infinite types can be infinite. But, as we shall see in this section, if we restrict ourselves to infinite types of a certain well-behaved form, called **regular types**, then the sets of reachable states are guaranteed to remain finite and the subtype checking algorithm will always terminate.

8.1 Definition: A tree type S is a **subtree** of a tree type T if S can be presented in the form $\mathtt{S} = \lambda\sigma.\ \mathtt{T}(\pi \cdot \sigma)$ for some π.We write *subtrees*(T) for the set of all subtrees of T.

8.2 Definition: A tree type $\mathtt{T} \in \mathcal{T}$ is **regular** if *subtrees*(T) is finite, i.e., T has only finitely many distinct subtrees. The set of regular tree types is denoted by $\mathcal{T}_r$.

8.3 Examples:

1. Any finite tree type is regular, since the number of subtrees is bounded by the number of nodes.
2. The number of distinct subtrees of a type can be strictly less than the number of nodes. For example, T = Top→(Top×Top) has five nodes but only three distinct subtrees.
3. Some infinite trees are regular. For example, the tree

   $$\mathtt{T} = \mathtt{Top} \times (\mathtt{Top} \times (\mathtt{Top} \times \ldots))$$

   has just two distinct subtrees (T itself and Top).
4. The type

   $$\mathtt{T} = \mathtt{B}\times(\mathtt{A}\times(\mathtt{B}\times(\mathtt{A}\times(\mathtt{A}\times(\mathtt{B}\times(\mathtt{A}\times(\mathtt{A}\times(\mathtt{A}\times(\mathtt{B}\times\ldots)$$

   where pairs of consecutive Bs are separated by increasingly many As, is not regular. Note that, because T is irregular, the set *reachable*(T, T) containing all the subtyping pairs needed to justify the statement T<:T is infinite.

8.4 Observation: The restriction S_r of S to regular tree types is finite state.

This means that we can obtain a decision procedure for the subtype relation by instantiating one of the membership algorithms with S. Of course, in order to implement such a decision procedure we would need to be able to decide when two regular trees are equal (to calculate the unions, set equality and membership, etc. used by the algorithms). The μ-notation in the next section can be used for this purpose, but we will go a step further, showing directly how to build a subtyping algorithm for μ-types.

9. μ-TYPES

We now formalize the finite μ-notation for regular types.

9.1 Definition: Let X range over a fixed countable set $\{\mathtt{X}_1, \mathtt{X}_2, \ldots\}$ of type variables. The set $\mathcal{T}_m^{raw}$ of **raw μ-types** is the set of expressions defined by the following grammar:

$$\begin{array}{lll} \mathtt{T} & ::= & \mathtt{X} \\ & & \mathtt{Top} \\ & & \mathtt{T}\times\mathtt{T} \\ & & \mathtt{T}\to\mathtt{T} \\ & & \mu\mathtt{X}.\mathtt{T} \end{array}$$

The syntactic operator μ is a binder, and gives rise, in the standard way, to notions of bound and free variables, closed raw μ-types, and equivalence of raw μ-types up to renaming of bound variables. $FV(\mathtt{T})$ denotes the set of free variables of a raw μ-type T. The capture-avoiding substitution $\{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{T}$ of a raw μ-type S for free occurrences of X in a raw μ-type T is defined in the usual way.

Raw μ-types have to be restricted a little to achieve a tight correspondence with regular trees: we want to be able to "read off" a tree type as the infinite unfolding of a given μ-type, but there are raw μ-types that *cannot* be reasonably interpreted as representations of tree types. These types have the form $\mu\mathtt{X}.\mu\mathtt{X}_1\ldots\mu\mathtt{X}_n.\mathtt{X}$, where the $\mathtt{X}_1$ through $\mathtt{X}_n$ are distinct from X. For example, consider $\mathtt{T} = \mu\mathtt{X}.\mathtt{X}$. Unfolding of T gives T again, so we cannot read off any tree by unfolding T. This leads us to the following restriction.

9.2 Definition: A raw μ-type T is **contractive** if, for any subexpression of T of the form $\mu\mathtt{X}.\mu\mathtt{X}_1\ldots\mu\mathtt{X}_n.\mathtt{S}$, the body S is not X. Alternatively, a raw μ-type is contractive if every occurrence of a μ-bound variable in it is separated from its binder by at least one $\to$ or $\times$.

A raw μ-type is called simply a **μ-type** if it is contractive. The set of μ-types is written $\mathcal{T}_m$.

When T is a μ-type, we write μ-$height(T)$ for the number of μ-bindings at the front of T.

The common understanding of μ-types as finite notation for infinite regular tree types is formalized by the following function *treeof*.

9.3 Definition: Define the function *treeof* mapping closed μ-types to tree types inductively as follows:

$$\begin{array}{lcl} treeof(\mathtt{Top})(\bullet) & = & \mathtt{Top} \\ treeof(\mathtt{T}_1\to\mathtt{T}_2)(\bullet) & = & \to \\ treeof(\mathtt{T}_1\to\mathtt{T}_2)(i\cdot\pi) & = & treeof(\mathtt{T}_i)(\pi) \\ treeof(\mathtt{T}_1\times\mathtt{T}_2)(\bullet) & = & \times \\ treeof(\mathtt{T}_1\times\mathtt{T}_2)(i\cdot\pi) & = & treeof(\mathtt{T}_i)(\pi) \\ treeof(\mu\mathtt{X}.\mathtt{T})(\pi) & = & treeof(\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T})(\pi) \end{array}$$

The mapping is lifted to the pairs of types in the standard way: $treeof(\mathtt{S},\mathtt{T}) = (treeof(\mathtt{S}), treeof(\mathtt{T}))$.

To verify that this definition is proper (i.e., exhaustive and terminating), note the following:

1. Every "recursive call" on the right-hand side reduces the lexicographic size of the pair $(|\pi|, \mu\text{-}height(\mathtt{T}))$: the cases for $\mathtt{S}\to\mathtt{T}$ and $\mathtt{S}\times\mathtt{T}$ reduce $|\pi|$ and the case for $\mu\mathtt{X}.\mathtt{T}$ preserves $|\pi|$, but reduces $\mu\text{-}height(\mathtt{T})$.
2. All recursive calls preserve contractiveness and closure of the argument types. In particular, the type $\mu\mathtt{X}.\mathtt{T}$ is contractive and closed iff its unfolding $\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T}$ is. This justifies the reference to unfolding in the definition of $treeof(\mu\mathtt{X}.\mathtt{T})$.

The subtyping relation for tree types was defined in Section 4 as the greatest fixed point of generating function S. We are going to define subtyping for μ-types similarly, using a generating function that, together with subtyping rules used in S, incorporates the rules for μ-types,

$$\frac{\mathtt{S} <: \{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T}}{\mathtt{S} <: \mu\mathtt{X}.\mathtt{T}} \quad \text{and} \quad \frac{\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{S}\}\mathtt{S} <: \mathtt{T}}{\mu\mathtt{X}.\mathtt{S} <: \mathtt{T}}$$

Below is the full definition.

9.4 Definition: Two μ-types $\mathtt{S}$ and $\mathtt{T}$ are in the subtyping relation if $(\mathtt{S},\mathtt{T}) \in \nu S_m$, where the monotone function $S_m \in \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m) \to \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m)$ is defined by:

$$\begin{array}{rcl} S_m(R) & = & \{(\mathtt{S},\mathtt{Top}) \mid \mathtt{S}\in\mathcal{T}_m\} \\ & \cup & \{(\mathtt{S}_1\times\mathtt{S}_2, \mathtt{T}_1\times\mathtt{T}_2) \mid (\mathtt{S}_1,\mathtt{T}_1), (\mathtt{S}_2,\mathtt{T}_2)\in R\} \\ & \cup & \{(\mathtt{S}_1\to\mathtt{S}_2, \mathtt{T}_1\to\mathtt{T}_2) \mid (\mathtt{T}_1,\mathtt{S}_1), (\mathtt{S}_2,\mathtt{T}_2)\in R\} \\ & \cup & \{(\mathtt{S},\mu\mathtt{X}.\mathtt{T}) \mid (\mathtt{S},\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T})\in R\} \\ & \cup & \{(\mu\mathtt{X}.\mathtt{S},\mathtt{T}) \mid (\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{S}\}\mathtt{S},\mathtt{T})\in R, \\ & & \qquad \mathtt{T}\neq\mathtt{Top}, \text{ and } \mathtt{T}\neq\mu\mathtt{Y}.\mathtt{T}_1\}. \end{array}$$

(Note the asymmetry between the final and penultimate clauses of S_m, needed to make S_m invertible. Otherwise, the clauses would overlap.) The *support* function corresponding to S_m is:

$$support_{S_m}(\mathtt{S},\mathtt{T}) = \begin{cases} \emptyset & \text{if } \mathtt{T}=\mathtt{Top} \\ \{(\mathtt{S}_1,\mathtt{T}_1), (\mathtt{S}_2,\mathtt{T}_2)\} & \text{if } \mathtt{S}=\mathtt{S}_1\times\mathtt{S}_2 \\ & \quad \text{and } \mathtt{T}=\mathtt{T}_1\times\mathtt{T}_2 \\ \{(\mathtt{T}_1,\mathtt{S}_1), (\mathtt{S}_2,\mathtt{T}_2)\} & \text{if } \mathtt{S}=\mathtt{S}_1\to\mathtt{S}_2 \\ & \quad \text{and } \mathtt{T}=\mathtt{T}_1\to\mathtt{T}_2 \\ \{(\mathtt{S},\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}_1\}\mathtt{T}_1)\} & \text{if } \mathtt{T}=\mu\mathtt{X}.\mathtt{T}_1 \\ \{(\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{S}_1\}\mathtt{S}_1,\mathtt{T})\} & \text{if } \mathtt{S}=\mu\mathtt{X}.\mathtt{S}_1 \text{ and} \\ & \quad \mathtt{T}\neq\mu\mathtt{X}.\mathtt{T}_1,\ \mathtt{T}\neq\mathtt{Top} \\ \uparrow & \text{otherwise.} \end{cases}$$

The two notions of subtyping, one for tree types and the other for μ-types, tightly correspond to each other, as the following theorem shows.

9.5 Theorem: Let $(\mathtt{S},\mathtt{T}) \in \mathcal{T}_m \times \mathcal{T}_m$. Then $(\mathtt{S},\mathtt{T}) \in \nu S_m$ iff $treeof(\mathtt{S},\mathtt{T}) \in \nu S$.

10. COUNTING SUBEXPRESSIONS

Instantiating the generic algorithm gfp^t (7.3) with the specific support function $support_{S_m}$ corresponding to the subtype relation on μ-types (9.4) yields the subtyping algorithm shown in Figure 2. Section 7 shows that the termination of this algorithm will be guaranteed if $reachable_{S_m}(\mathtt{S},\mathtt{T})$ is finite for any pair of μ-types $(\mathtt{S},\mathtt{T})$. The current section is devoted to proving that this is the case (cf. Proposition 10.11).

$subtype(A, \mathtt{S}, \mathtt{T}) =$
 if $(\mathtt{S},\mathtt{T}) \in A$, then
  A
 else let $A_0 = A \cup (\mathtt{S},\mathtt{T})$ in
  if $\mathtt{T} = \mathtt{Top}$, then
   A_0
  else if $\mathtt{S} = \mathtt{S}_1\times\mathtt{S}_2$ and $\mathtt{T} = \mathtt{T}_1\times\mathtt{T}_2$, then
   let $A_1 = subtype(A_0, \mathtt{S}_1, \mathtt{T}_1)$ in
   $subtype(A_1, \mathtt{S}_2, \mathtt{T}_2)$
  else if $\mathtt{S} = \mathtt{S}_1\to\mathtt{S}_2$ and $\mathtt{T} = \mathtt{T}_1\to\mathtt{T}_2$, then
   let $A_1 = subtype(A_0, \mathtt{T}_1, \mathtt{S}_1)$ in
   $subtype(A_1, \mathtt{S}_2, \mathtt{T}_2)$
  else if $\mathtt{S} = \mu\mathtt{X}.\mathtt{S}_1$, then
   $subtype(A_0,\ \{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{S}_1\}\mathtt{S}_1,\ \mathtt{T})$
  else if $\mathtt{T} = \mu\mathtt{X}.\mathtt{T}_1$, then
   $subtype(A_0,\ \mathtt{S},\ \{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}_1\}\mathtt{T}_1)$
  else
   $\uparrow$

Figure 2: The concrete subtyping algorithm

At first glance, this property seems plausible, but proving it rigorously requires a bit of work. In fact, there are two possible ways of defining the set of "closed subexpressions" of a μ-type: one (which we call *top-down* subexpressions) directly corresponding to the subexpressions generated by $support_{S_m}$, and another (*bottom-up* subexpressions) for which it is easy to show that the set of closed subexpressions of every closed μ-type is finite. The termination proof proceeds by defining both of these sets and showing that the former is a subset of the latter. Our development is based on Brandt and Henglein [6].

10.1 Definition: A μ-type $\mathtt{S}$ is a **top-down subexpression** of a μ-type $\mathtt{T}$, written $\mathtt{S} \sqsubseteq \mathtt{T}$, if the pair $(\mathtt{S},\mathtt{T})$ is in the least fixed point of the following generating function:

$$\begin{array}{rcl} Td(R) & = & \{(\mathtt{T},\mathtt{T}) \mid \mathtt{T}\in\mathcal{T}_m\} \\ & \cup & \{(\mathtt{S},\mathtt{T}_1\times\mathtt{T}_2) \mid (\mathtt{S},\mathtt{T}_1)\in R\} \\ & \cup & \{(\mathtt{S},\mathtt{T}_1\times\mathtt{T}_2) \mid (\mathtt{S},\mathtt{T}_2)\in R\} \\ & \cup & \{(\mathtt{S},\mathtt{T}_1\to\mathtt{T}_2) \mid (\mathtt{S},\mathtt{T}_1)\in R\} \\ & \cup & \{(\mathtt{S},\mathtt{T}_1\to\mathtt{T}_2) \mid (\mathtt{S},\mathtt{T}_2)\in R\} \\ & \cup & \{(\mathtt{S},\mu\mathtt{X}.\mathtt{T}) \mid (\mathtt{S},\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T})\in R\} \end{array}$$

10.2 Exercise: Give an equivalent definition of the relation $\mathtt{S} \sqsubseteq \mathtt{T}$ as a set of inference rules.

From the definition of $support_{S_m}$ it is easy to see that, for any μ-types $\mathtt{S}$ and $\mathtt{T}$, the pairs contained in $support_{S_m}(\mathtt{S},\mathtt{T})$ are formed from top-down subexpressions of $\mathtt{S}$ and $\mathtt{T}$:

10.3 Lemma: If $(\mathtt{S}',\mathtt{T}') \in support_{S_m}(\mathtt{S},\mathtt{T})$, then either $\mathtt{S}' \sqsubseteq \mathtt{S}$ or $\mathtt{S}' \sqsubseteq \mathtt{T}$, and either $\mathtt{T}' \sqsubseteq \mathtt{S}$ or $\mathtt{T}' \sqsubseteq \mathtt{T}$.

The top-down subexpression relation is transitive:

10.4 Lemma: If $\mathtt{S} \sqsubseteq \mathtt{U}$ and $\mathtt{U} \sqsubseteq \mathtt{T}$, then $\mathtt{S} \sqsubseteq \mathtt{T}$.

Combining the two previous lemmas gives the proposition which motivated the introduction of top-down subexpressions.

10.5 Proposition: If $(\mathtt{S}', \mathtt{T}') \in \mathit{reachable}_{\mathcal{S}_m}(\mathtt{S}, \mathtt{T})$, then $\mathtt{S}' \sqsubseteq \mathtt{S}$ or $\mathtt{S}' \sqsubseteq \mathtt{T}$, and $\mathtt{T}' \sqsubseteq \mathtt{S}$ or $\mathtt{T}' \sqsubseteq \mathtt{T}$.

The finiteness of $\mathit{reachable}(\mathtt{S}, \mathtt{T})$ follows from the above proposition and the fact that any μ-type $\mathtt{U}$ has only a finite number of top-down subexpressions. Unfortunately, the latter fact is not obvious from the definition of $\sqsubseteq$. Attempting to prove it by structural induction on $\mathtt{U}$ using defining clauses of Td does not work because the last clause of Td would break the induction: to construct subexpressions of $\mathtt{U} = \mu\mathtt{X}.\mathtt{T}$ it refers to a larger expression $\{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{T}$. The alternative notion of bottom-up subexpressions avoids this problem by performing the substitution of μ-types for recursion variables *after* calculating the subexpressions instead of before. This leads to a simple proof of finiteness.

10.6 Definition: A μ-type $\mathtt{S}$ is a **bottom-up subexpression** of a μ-type $\mathtt{T}$, written $\mathtt{S} \preceq \mathtt{T}$, if the pair $(\mathtt{S}, \mathtt{T})$ is in the least fixed point of the following generating function:

$$\begin{array}{rcl} \mathit{Bu}(R) & = & \{(\mathtt{T}, \mathtt{T}) \mid \mathtt{T} \in \mathcal{T}_m\} \\ & \cup & \{(\mathtt{S}, \mathtt{T}_1\times\mathtt{T}_2) \mid (\mathtt{S}, \mathtt{T}_1) \in R\} \\ & \cup & \{(\mathtt{S}, \mathtt{T}_1\times\mathtt{T}_2) \mid (\mathtt{S}, \mathtt{T}_2) \in R\} \\ & \cup & \{(\mathtt{S}, \mathtt{T}_1\to\mathtt{T}_2) \mid (\mathtt{S}, \mathtt{T}_1) \in R\} \\ & \cup & \{(\mathtt{S}, \mathtt{T}_1\to\mathtt{T}_2) \mid (\mathtt{S}, \mathtt{T}_2) \in R\} \\ & \cup & \{(\{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{S}, \mu\mathtt{X}.\mathtt{T}) \mid (\mathtt{S}, \mathtt{T}) \in R\} \end{array}$$

The new notion of subexpressions differs from the one given earlier only in the clause for a type starting with a μ binder. To obtain the top-down subexpressions of such a type, we unfolded it first and then collected the subexpressions of the unfolding. To obtain the bottom-up subexpressions, we first collect the (not necessarily closed) subexpressions of the body, and then close them by applying the unfolding substitution.

10.7 Exercise: Give an equivalent definition of the relation $\mathtt{S} \preceq \mathtt{T}$ as a set of inference rules.

The fact that an expression has only finitely many bottom-up subexpressions is easily proved.

10.8 Lemma: $\{\mathtt{S} \mid \mathtt{S} \preceq \mathtt{T}\}$ is finite.

The next substitution lemma will be needed in the proof of the proposition that follows it.

10.9 Lemma: If $\mathtt{S} \preceq \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}$, then either $\mathtt{S} \preceq \mathtt{Q}$ or else $\mathtt{S} = \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{S}'$ for some $\mathtt{S}'$ with $\mathtt{S}' \preceq \mathtt{T}$.

The final piece of the proof establishes that all top-down subexpressions of a μ-type occur among its bottom-up subexpressions.

10.10 Proposition: If $\mathtt{S} \sqsubseteq \mathtt{T}$, then $\mathtt{S} \preceq \mathtt{T}$.

Proof: We want to show that $\mu\mathit{Td} \subseteq \mu\mathit{Bu}$. By the principle of induction, this follows from the fact that $\mu\mathit{Bu}$ is Td-closed, that is $\mathit{Td}(\mu\mathit{Bu}) \subseteq \mu\mathit{Bu}$. To obtain the latter, we just have to consider the effect of each clause of Td on $\mu\mathit{Bu}$. Since Td and Bu are defined similarly, the cases of all the clauses are trivial, except the last one, where we apply Lemma 10.9.

Combining Proposition 10.5, Proposition 10.10 and Lemma 10.8 gives the final result:

10.11 Proposition: For any μ-types $\mathtt{S}$ and $\mathtt{T}$, the set $\mathit{reachable}_{\mathcal{S}_m}(\mathtt{S}, \mathtt{T})$ is finite.

11. DIGRESSION: AN EXPONENTIAL ALGORITHM

The subtyping algorithm presented at the beginning of Section 10 can be simplified a bit more by making it return just a boolean value rather than a new set of assumptions (see Figure 3). The resulting procedure corresponds to Amadio and Cardelli's algorithm for checking subtyping [3]. It computes the same relation as the one computed by *subtype*, but much less efficiently because it does not remember pairs of types in the subtype relation across the recursive calls it makes in the $\to$ and $\times$ cases. This seemingly innocent change results in a blowup of the number of recursive calls the algorithm makes. Whereas the number of recursive calls made by *subtype* is proportional to the product of the sizes of the two argument types, in the case of $\mathit{subtype}^{ac}$ it is exponential.

$\mathit{subtype}^{ac}(A, \mathtt{S}, \mathtt{T}) =$
    if $(\mathtt{S}, \mathtt{T}) \in A$, then
      true
    else let $A_0 = A \cup (\mathtt{S}, \mathtt{T})$ in
      if $\mathtt{T} = \mathtt{Top}$, then
        true
      else if $\mathtt{S} = \mathtt{S}_1\times\mathtt{S}_2$ and $\mathtt{T} = \mathtt{T}_1\times\mathtt{T}_2$, then
        $\mathit{subtype}^{ac}(A_0, \mathtt{S}_1, \mathtt{T}_1)$ and
        $\mathit{subtype}^{ac}(A_0, \mathtt{S}_2, \mathtt{T}_2)$
      else if $\mathtt{S} = \mathtt{S}_1\to\mathtt{S}_2$ and $\mathtt{T} = \mathtt{T}_1\to\mathtt{T}_2$, then
        $\mathit{subtype}^{ac}(A_0, \mathtt{T}_1, \mathtt{S}_1)$ and
        $\mathit{subtype}^{ac}(A_0, \mathtt{S}_2, \mathtt{T}_2)$
      else if $\mathtt{S} = \mu\mathtt{X}.\mathtt{S}_1$, then
        $\mathit{subtype}^{ac}(A_0,\ \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{S}_1\}\mathtt{S}_1,\ \mathtt{T})$
      else if $\mathtt{T} = \mu\mathtt{X}.\mathtt{T}_1$, then
        $\mathit{subtype}^{ac}(A_0,\ \mathtt{S},\ \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}_1\}\mathtt{T}_1)$
      else
        false.

Figure 3: Amadio and Cardelli's subtyping algorithm

The exponential behavior of the Amadio and Cardelli's algorithm can be seen clearly in the following example. Define families of types $\mathtt{S}_n$ and $\mathtt{T}_n$ inductively as follows:

$$\begin{array}{rcl} \mathtt{S}_0 & = & \mu\mathtt{X}.\mathtt{Top}\times\mathtt{X} \\ \mathtt{T}_0 & = & \mu\mathtt{X}.\mathtt{Top}\times(\mathtt{Top}\times\mathtt{X}) \\ \mathtt{S}_{n+1} & = & \mu\mathtt{X}.\mathtt{X}\to\mathtt{S}_n \\ \mathtt{T}_{n+1} & = & \mu\mathtt{X}.\mathtt{X}\to\mathtt{T}_n. \end{array}$$

Since both $\mathtt{S}_n$ and $\mathtt{T}_n$ have just one occurrence of $\mathtt{S}_{n-1}$ and $\mathtt{T}_{n-1}$ correspondingly, their size will be linear in n after unfolding the abbreviations. Checking $\mathtt{S}_n < \mathtt{T}_n$ is going to generate an exponential derivation, however. This can be seen by the following sequence of recursive calls

$$\begin{array}{ll}
 & \mathit{subtype}^{ac}(\emptyset, \mathtt{S}_n, \mathtt{T}_n) \\
= & \mathit{subtype}^{ac}(\mathtt{A}_1, \mathtt{S}_n \to \mathtt{S}_{n-1}, \mathtt{T}_n) \\
= & \mathit{subtype}^{ac}(\mathtt{A}_2, \mathtt{S}_n \to \mathtt{S}_{n-1}, \mathtt{T}_n \to \mathtt{T}_{n-1}) \\
= & \mathit{subtype}^{ac}(\mathtt{A}_3, \mathtt{T}_n, \mathtt{S}_n) \text{ and } \underline{\mathit{subtype}^{ac}(\mathtt{A}_3, \mathtt{S}_{n-1}, \mathtt{T}_{n-1})} \\
= & \mathit{subtype}^{ac}(\mathtt{A}_3, \mathtt{T}_n, \mathtt{S}_n) \text{ and } \ldots \\
= & \mathit{subtype}^{ac}(\mathtt{A}_4, \mathtt{T}_n \to \mathtt{T}_{n-1}, \mathtt{S}_n) \text{ and } \ldots \\
= & \mathit{subtype}^{ac}(\mathtt{A}_5, \mathtt{T}_n \to \mathtt{T}_{n-1}, \mathtt{S}_n \to \mathtt{S}_{n-1}) \text{ and } \ldots \\
= & \mathit{subtype}^{ac}(\mathtt{A}_6, \mathtt{S}_n, \mathtt{T}_n) \text{ and } \underline{\mathit{subtype}^{ac}(\mathtt{A}_6, \mathtt{T}_{n-1}, \mathtt{S}_{n-1})} \\
 & \qquad\qquad\qquad \text{and } \ldots \\
= & \text{etc.,}
\end{array}$$

where

$$\begin{array}{lcl}
\mathtt{A}_1 & = & \{(\mathtt{S}_n, \mathtt{T}_n)\} \\
\mathtt{A}_2 & = & \mathtt{A}_1 \cup \{(\mathtt{S}_n \to \mathtt{S}_{n-1}, \mathtt{T}_n)\} \\
\mathtt{A}_3 & = & \mathtt{A}_2 \cup \{(\mathtt{S}_n \to \mathtt{S}_{n-1}, \mathtt{T}_n \to \mathtt{T}_{n-1})\} \\
\mathtt{A}_4 & = & \mathtt{A}_3 \cup \{(\mathtt{T}_n, \mathtt{S}_n)\} \\
\mathtt{A}_5 & = & \mathtt{A}_4 \cup \{(\mathtt{T}_n \to \mathtt{T}_{n-1}, \mathtt{S}_n)\} \\
\mathtt{A}_6 & = & \mathtt{A}_5 \cup \{(\mathtt{T}_n \to \mathtt{T}_{n-1}, \mathtt{S}_n \to \mathtt{S}_{n-1})\}.
\end{array}$$

Notice that the initial call $\mathit{subtype}^{ac}(\emptyset, \mathtt{S}_n, \mathtt{T}_n)$ results in the two underlined recursive calls of the same form involving $\mathtt{S}_{n-1}$ and $\mathtt{T}_{n-1}$. These, in turn, will each give rise to two recursive calls involving $\mathtt{S}_{n-2}$ and $\mathtt{T}_{n-2}$, and so on. The total number of recursive calls will clearly be proportional to 2^n.

12. FURTHER READING

For background on coinduction, readers are referred to Barwise and Moss' *Vicious Circles* [5], Gordon's tutorial on coinduction and functional programming [12], and Milner and Tofte's expository article on the use of coinduction in programming language semantics [17]. Basic information on monotone functions and fixed points can be found in [1] and [10].

The use of coinductive proof methods in computer science dates from the 1970s, for example in the work of Milner [16] and Park [19] on concurrency (also cf. Arbib and Manes's categorical discussion of duality in automata theory [4]). But the use of induction in its dual "co-" form was familiar to mathematicians considerably earlier and is developed explicitly, for example, in universal algebra and category theory. Aczel's seminal book [2] on non well-founded sets includes a brief historical survey.

Recursive types in computer science go back to (at least) Morris [18]. Basic syntactic and semantic properties (without subtyping) are collected in Cardone and Coppo [7]. Properties of infinite and regular trees are surveyed by Courcelle [8].

Amadio and Cardelli [3] gave the first subtyping algorithm for recursive types. Their paper defines three relations: an inclusion relation between infinite trees, an algorithm that checks subtyping between μ-types, and a reference subtyping relation between μ-types defined as the least fixed point of a set of declarative inference rules; these relations are proved to be equivalent, and connected to a PER model construction. Coinduction is not used. Instead, to reason about infinite trees, a notion of finite approximations of an infinite tree is introduced. This notion plays a key role in many of the proofs.

Brandt and Henglein [6] lay bare the underlying coinductive nature of Amadio and Cardelli's system. They give a new inductive axiomatization of the subtyping relation that is sound and complete with respect to that of Amadio and Cardelli. The so-called ARROW/FIX rule of the axiomatization embodies the coinductiveness of the system. The paper describes a general method for deriving an inductive axiomatization for relations that are naturally defined by coinduction and presents a detailed proof of termination for a subtyping algorithm. Section 10 of the present paper is essentially a sketch of the latter proof. Brand and Henglein establish that the complexity of their algorithm is $O(n^2)$.

Kozen, Palsberg, and Schwartzbach [15] describe an elegant quadratic subtyping algorithm for recursive types. They observe that a regular recursive type corresponds to an automaton with labeled states. Then, they define a form of product of two automata that yields a conventional word automaton accepting a word iff the types corresponding to the original automata are not in the subtype relation. A linear-time emptyness test now solves the subtyping problem. This fact, plus the quadratic complexity of product construction and linear-time conversion from types to automata, gives an overall quadratic complexity for the subtyping algorithm.

Hosoya, Vouillon, and Pierce [13] use a related automata-theoretic approach, associating recursive types (with unions) to tree automata in a subtyping algorithm tuned to XML processing applications.

Jim and Palsberg [14] address type inference for languages with subtyping and recursive types. Like us, they adopt a coinductive view of the subtype relation over infinite trees and motivate a subtype checking algorithm as a procedure building the minimal simulation (i.e., consistent set, in our terminology) from a given pair of types. They define the notions of consistency and $P1$-closure of a relation over types, which correspond to our consistency and reachable sets.

Acknowledgments

We are grateful for insights and encouragement from Robert Harper, Haruo Hosoya, Perdita Stevens, Jérôme Vouillon, and Philip Wadler, and for careful readings of the manuscript by Penny Anderson, Alan Schmitt, and the ICFP reviewers. This work was supported by the University of Pennsylvania and by NSF Career grant CCR-9701826, *Principled Foundations for Programming with Objects*.

13. REFERENCES

[1] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, number 90 in Studies in Logic and the Foundations of Mathematics, pages 739–782. North Holland, 1977.

[2] P. Aczel. *Non-Well-Founded Sets*. Stanford Center for the Study of Language and Information, 1988. CSLI Lecture Notes number 14.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Preliminary version in POPL '91 (pp. 104–118); also DEC Systems Research Center Research Report number 62, August 1990.

[4] M. Arbib and E. Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press, 1975.

[5] J. Barwise and L. Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena*.

Cambridge University Press, 1996. Originally appeared as CSLI Lecture Notes 60.
[6] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In R. Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, Apr. 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309-338, 1998.
[7] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, 1991.
[8] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
[9] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, May 1999.
[10] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.
[11] G. Ghelli. Recursive types are not conservative over $F_{\leq}$. In M. Bezen and J. Groote, editors, *Proceedings of thethe International Conference on Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 146–162, Berlin, March 1993. Springer Verlag.
[12] A. Gordon. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, pages 78–95. Springer Workshops in Computing, 1995.
[13] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.
[14] T. Jim and J. Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.
[15] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 419–428, 1993.
[16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer Verlag, 1980.
[17] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.
[18] J. H. Morris. Lambda calculus models of programming languages. Technical Report MIT-LCS//MIT/LCS/TR-57, Massachusetts Institute of Technology, Laboratory for Computer Science, Dec. 1968.
[19] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, Berlin, 1981.
[20] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

Solutions to Selected Exercises

Solution to 2.7:

$$\begin{array}{lcl}
E_2(\{\}) & = & \{a\} \\
E_2(\{a\}) & = & \{a\} \\
E_2(\{b\}) & = & \{a\} \\
E_2(\{c\}) & = & \{a,b\} \\
E_2(\{a,b\}) & = & \{a,c\} \\
E_2(\{a,c\}) & = & \{a,b\} \\
E_2(\{b,c\}) & = & \{a,b\} \\
E_2(\{a,b,c\}) & = & \{a,b,c\}
\end{array}$$

The E_2-closed sets are: $\{a\}$, $\{a,b,c\}$. The E_2-consistent sets are: $\emptyset$, $\{a\}$, $\{a,b,c\}$. The least fixed point of E_2 is $\{a\}$. The greatest fixed point is $\{a,b,c\}$.

Solution to 4.3: The pair ($\mathtt{Top}$, $\mathtt{Top}\times\mathtt{Top}$) is not in νS. To see this, just observe from the definition of S that this pair is not in $S(X)$ for any X. So there is no S-consistent set containing this pair, and in particular νS (which is S-consistent) does not contain it.

Solution to 4.4: For the first part, any pair of infinite types will do. For the second, there are no such types: the least and greatest fixed points of S_f coincide.

Solution to 4.8: Begin by defining the identity relation on tree types: $I = \{(\mathtt{T},\mathtt{T}) \mid \mathtt{T} \in \mathcal{T}\}$. Observe (straightforwardly) that I is S-consistent. From this and the coinduction principle, $I \subseteq \nu S$, that is, νS is reflexive.

Solution to 5.2: By the coinduction principle, it is enough to show that $\mathcal{T}\times\mathcal{T}$ is S^{tr}-consistent, i.e., $\mathcal{T}\times\mathcal{T} \subseteq S^{tr}(\mathcal{T}\times\mathcal{T})$. Suppose $(\mathtt{S},\mathtt{T}) \in \mathcal{T}\times\mathcal{T}$. Pick any $\mathtt{U} \in \mathcal{T}$. Then $(\mathtt{S},\mathtt{U}), (\mathtt{U},\mathtt{T}) \in \mathcal{T}\times\mathcal{T}$, and so, by the definition of S^{tr}, also $(\mathtt{S},\mathtt{T}) \in S^{tr}(\mathcal{T}\times\mathcal{T})$.

Solution to 6.3:

$$\frac{i \quad a}{h} \qquad \frac{b \quad c}{a} \qquad \frac{b}{d} \qquad \frac{d}{e} \qquad \frac{e}{b} \qquad \frac{f \quad g}{c} \qquad \frac{g}{f} \qquad \frac{}{g}$$

Solution to 10.2:

$$\mathtt{T} \sqsubseteq \mathtt{T}$$

$$\frac{\mathtt{S} \sqsubseteq \mathtt{T}_1}{\mathtt{S} \sqsubseteq \mathtt{T}_1\times\mathtt{T}_2} \qquad \frac{\mathtt{S} \sqsubseteq \mathtt{T}_2}{\mathtt{S} \sqsubseteq \mathtt{T}_1\times\mathtt{T}_2}$$

$$\frac{\mathtt{S} \sqsubseteq \mathtt{T}_1}{\mathtt{S} \sqsubseteq \mathtt{T}_1\to\mathtt{T}_2} \qquad \frac{\mathtt{S} \sqsubseteq \mathtt{T}_2}{\mathtt{S} \sqsubseteq \mathtt{T}_1\to\mathtt{T}_2}$$

$$\frac{\mathtt{S} \sqsubseteq \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{T}}{\mathtt{S} \sqsubseteq \mu\mathtt{X}.\mathtt{T}}$$

The Td generating function differs from the generating functions we have considered throughout this paper: it is not invertible. It could be seen by considering the assertion $\mathtt{B} \sqsubseteq \mathtt{A}\times\mathtt{B}\to\mathtt{B}\times\mathtt{C}$; it is supported by the two assertion sets $\{\mathtt{B} \sqsubseteq \mathtt{A}\times\mathtt{B}\}$ and $\{\mathtt{B} \sqsubseteq \mathtt{B}\times\mathtt{C}\}$, but neither of them is a subset of the other.

Solution to 10.7: All the rules for Bu are the same as the rules for Td given in the solution of Exercise 10.2, except the rule for μ-type:

$$\frac{\mathtt{S} \preceq \mathtt{T}}{\{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{S} \preceq \mu\mathtt{X}.\mathtt{T}}$$