November 2002

Recursive subtyping revealed

Vladimir Gapeyev
University of Pennsylvania

Michael Y. Levin
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania, bcpierce@cis.upenn.edu

Copyright Cambridge University Press. Reprinted from *Journal of Functional Programming*, Volume 12, Issue 6, November 2002, pages 511-548.

This article also appears as chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce [MIT Press, 2002].

Recursive subtyping revealed

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.

Comments
Copyright Cambridge University Press. Reprinted from *Journal of Functional Programming*, Volume 12, Issue 6, November 2002, pages 511-548.

This article also appears as chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce [MIT Press, 2002].

JFP **12** (6): 511–548, November 2002. © 2002 Cambridge University Press

DOI: 10.1017/S0956796802004318 Printed in the United Kingdom

*Recursive subtyping revealed**

VLADIMIR GAPEYEV, MICHAEL Y. LEVIN and BENJAMIN C. PIERCE

Department of Computer & Information Science, University of Pennsylvania, 200 South 33rd Street, Philadelphia, PA 19104-6389, USA

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.

Capsule Review

This paper provides a self-contained introduction to the theory of recursive subtyping, an area first studied by Amadio and Cardelli and later refined and reformulated by Brandt and Henglein, among others. The current paper aims at bringing together recent results on the subject, and presenting them, as well as the foundational work of Amadio and Cardelli, in the unifying setting of coinduction. As such, the paper does not provide any results of its own: its value lies in filling a pedagogical gap in an area which so far has lacked a comprehensive introduction.

However, this paper should not be judged solely on its contribution to the field of recursive subtyping. It can just as well be seen as an introductory text on coinduction in general, using the type system aspect as a running example. This dual purpose makes the article especially interesting as lecture material – the student of recursive subtyping benefits from a thorough survey of the semantic tools that he or she will need, while the reader primarily interested in the tools themselves will value the level of detail by which the coinductive framework is exemplified.

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

* This article also appears as chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002).

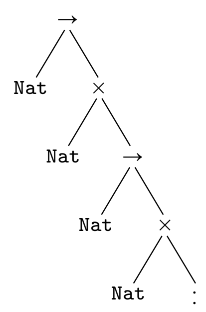
μX.Nat$\rightarrow$(Nat$\times$X). A variety of familiar recursive types such as lists and trees can be defined analogously.

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

$$\begin{array}{lll} \texttt{unfold} & \in & \mu\texttt{X.T} \rightarrow \{\texttt{X} \mapsto \mu\texttt{X.T}\}\texttt{T} \\ \texttt{fold} & \in & \{\texttt{X} \mapsto \mu\texttt{X.T}\}\texttt{T} \rightarrow \mu\texttt{X.T} \end{array}$$

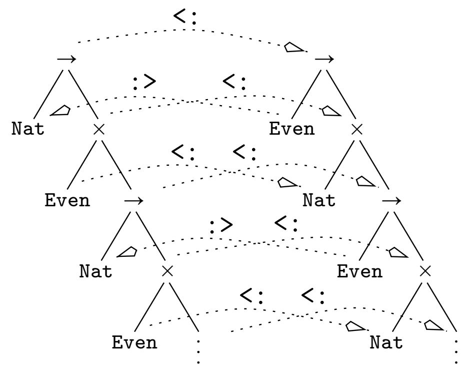
witnessing the isomorphism (as usual, $\{\texttt{X} \mapsto \texttt{S}\}\texttt{T}$ denotes the substitution of S for free occurrences of X in T).

In the *equi-recursive* formulation (our focus in this article), 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 μX.Nat$\rightarrow$(Nat$\times$X) as merely an abbreviation for the infinite tree obtained by unrolling the recursion 'out to infinity':



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 μX.Nat$\rightarrow$(Even$\times$X) and μX.Even$\rightarrow$(Nat$\times$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 process 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:



Can such arguments be made precise? Indeed they can. The basic ideas can be found in several places, going back to Amadio & Cardelli's (1993) comprehensive study, 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. Kozen *et al*., 1993; Brandt & Henglein, 1997; Jim & Palsberg, 1999). 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 subtype 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). At this point, we pause our discussion of types and subtyping and return to the general framework of induction and coinduction. Section 6 derives simple algorithms for checking membership in inductively and co-inductively defined sets; Section 7 considers more refined algorithms. In Section 8, we return to types and define a subtype relation for a special case of 'regular' infinite trees. The general algorithms of the previous two sections are then instantiated to decide regular tree subtyping. Section 9 introduces μ-types as a finite notation for representing tree types and establishes a theorem that the more complex (but finitely realizable) subtype relation on μ-types coincides with the ordinary coinductive definition of subtyping on representable trees. Section 10 brings together all the preceding material to derive a concrete subtyping algorithm

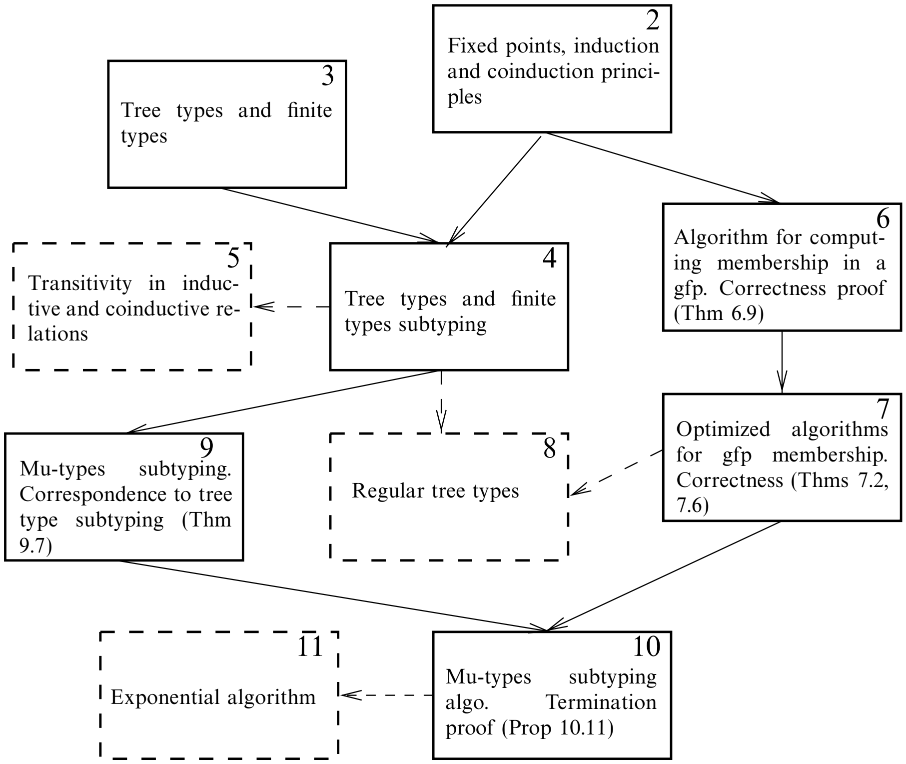


Fig. 1. Section dependencies.

for μ-types and proves its termination. Finally, Section 11 discusses a well-known variant of the algorithm and shows that it has exponential behavior. Several sections are accompanied by exercises for the reader; solutions to these can be found at the end of the paper.

To help the reader navigate, figure 1 presents a flow chart of section dependencies. Dashed boxes represent detours that are inessential for the overall flow of the article. The diagram shows several possible paths through the material. Sections 2, 6 and 7 address general principles of induction and coinduction, derivation of algorithms for testing membership in (co)inductively defined sets, and proofs of their correctness. Sections 2, 3, 4, and 9 can serve as an introduction to the coinductive definition of subtyping on infinite trees, μ-types as their finite representation, coinductive definition of subtyping on μ-types, and the proof of the correspondence between these two subtyping relations. To understand the complete picture, all the sections shown in solid boxes are needed.

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.

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 (see Ghelli, 1993), 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.

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', and 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. 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 $\mathcal{P}(\mathcal{U})$.

Definition 2.1
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 the following, we assume that F is some monotone function on $\mathcal{P}(\mathcal{U})$. We often refer to F as a *generating function*.

Definition 2.2
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 elements justified by F – it already contains all the conclusions that are justified by its members. An F-consistent set, on the other hand, 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 is a set that is both closed and consistent: it includes all the justifications required by its members, all the conclusions that follow from its members, and nothing else.

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

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

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

E_1 can be represented compactly by a collection of *inference rules*:

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

Each 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.

Theorem 2.4

1. The intersection of all F-closed sets is the least fixed point of F.
2. The union of all F-consistent sets is the greatest fixed point of F.

Proof

We consider only part (2); the proof of part (1) is symmetric. Let $C = \{X \mid X \subseteq F(X)\}$ be the collection of all F-consistent sets, and let P be the union of all these sets. Taking into account the fact that F is monotone and that, for any $X \in C$, we know both that X is F-consistent and that $X \subseteq P$, we obtain $X \subseteq F(X) \subseteq F(P)$. Consequently, $P = \bigcup_{X \in C} X \subseteq F(P)$, i.e. P is F-consistent. Moreover, by its definition, P is the largest F-consistent set. Using the monotonicity of F again, we obtain $F(P) \subseteq F(F(P))$. This means, by the definition of C, that $F(P) \in C$. Hence, as for any member of C, we have $F(P) \subseteq P$, i.e. P is F-closed. Now we have established both that P is the largest F-consistent set and that P is a fixed point of F, so P is the largest fixed point. □

Definition 2.5

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

Example 2.6

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

Exercise 2.7

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?

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). This observation gives us a pair of fundamental reasoning tools:

Corollary 2.8 (of Theorem 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 $\mathscr{U}$ for which the predicate is true; showing that property X holds of an element x is the same as showing that x is in the set X. Now, the induction principle says that any property whose characteristic set is closed under F (i.e. the property is preserved by F) is true of all the elements of the inductively defined set μF.

The coinduction principle, on the other hand, gives us a method for establishing that an element x is *in* the coinductively defined set νF. To show $x \in \nu F$, it suffices to find a set X such that $x \in X$ and X is F-consistent. Although it is a little less familiar than induction, the principle of coinduction is central to many areas of computer science; for example, it is the main proof technique in theories of concurrency based on *bisimulation*, and it lies at the heart of many *model checking* algorithms.

The principles of induction and coinduction are used heavily throughout the paper. We do not write out every inductive argument in terms of generating functions and predicates; instead, in the interest of brevity, we often rely on familiar abbreviations such as structural induction. Coinductive arguments are presented more explicitly.

Exercise 2.9
Show that the following familiar induction principles follow from the general principle of induction in Corollary 2.8.

- *Induction on natural numbers:* let $P \subseteq \mathbb{N}$ be a predicate on natural numbers. If $P(0)$ and $\forall i \in \mathbb{N}\,.\; P(i) \Rightarrow P(i+1)$, then $\forall n \in \mathbb{N}\,.\; P(n)$,
- *Lexicographic induction on pairs*: let $P \subseteq \mathbb{N} \times \mathbb{N}$ be a predicate on pairs of natural numbers. If $\forall (m,n) \in \mathbb{N} \times \mathbb{N}\,.\; [\forall (m',n') < (m,n).\; P(m',n')] \Rightarrow P(m,n)$, then $\forall (m,n) \in \mathbb{N} \times \mathbb{N}\,.\; P(m,n)$.

(Recall that the lexicographic order on pairs is defined by: $(m,n) < (m',n')$ iff either $m < m'$ or $m = m'$ and $n < n'$.)

3 Finite and infinite types

We are going to instantiate the general definitions of greatest fixed points and the coinductive proof method with the specifics of subtyping. Before we can do this, though, we need to show 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 represent 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 Courcelle (1983).

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.

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

- $\mathtt{T}(\bullet)$ is defined;
- if $\mathtt{T}(\pi \cdot \sigma)$ is defined then $\mathtt{T}(\pi)$ is defined;
- if $\mathtt{T}(\pi) = \to$ 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*(T) is finite. The set of all tree types is written $\mathscr{T}$; the subset of all finite tree types is written $\mathscr{T}_f$.

For notational convenience, we write Top for the tree T with $\mathtt{T}(\bullet) = \mathtt{Top}$. 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 $\mathtt{T}_1\to\mathtt{T}_2$ for the tree with $(\mathtt{T}_1\to\mathtt{T}_2)(\bullet) = \to$ and $(\mathtt{T}_1\to\mathtt{T}_2)(i\cdot\pi) = \mathtt{T}_i(\pi)$, for $i = 1, 2$. For example, the expression $(\mathtt{Top}\times\mathtt{Top})\to\mathtt{Top}$ denotes the finite tree type T defined by the function with $\mathtt{T}(\bullet) = \to$ and $\mathtt{T}(1) = \times$ and $\mathtt{T}(2) = \mathtt{T}(1 \cdot 1) = \mathtt{T}(1 \cdot 2) = \mathtt{Top}$. We use ellipses informally for describing non-finite tree types. For example, $\mathtt{Top}\to(\mathtt{Top}\to(\mathtt{Top}\to\ldots))$ corresponds to the type T defined by $\mathtt{T}(2^k) = \to$, for all $k \geqslant 0$, and $\mathtt{T}(2^k \cdot 1) = \mathtt{Top}$, for all $k \geqslant 0$. Figure 2 illustrates these conventions.

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

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

Formally, $\mathscr{T}_f$ is the least fixed point of the generating function described by the grammar. The universe of this generating function is the set of all finite and infinite trees labeled with Top, $\to$, and $\times$ (i.e. the set formed by generalizing Definition 3 by dropping its two last conditions). The whole set $\mathscr{T}$ can be derived from the same generating function by taking the greatest fixed point instead of the least.

Exercise 3.2
Following the ideas in the previous paragraph, suggest a universe $\mathscr{U}$ and a generating function $F \in \mathscr{P}(\mathscr{U}) \to \mathscr{P}(\mathscr{U})$ such that the set of finite tree types $\mathscr{T}_f$ is the least fixed point of F and the set of all tree types $\mathscr{T}$ is its greatest fixed point.

4 Subtyping

We define subtype 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 $\mathscr{T}_f \times \mathscr{T}_f$ of pairs of finite tree types; our generating function will map subsets of this universe – that is, relations on $\mathscr{T}_f$ – to other subsets, and their fixed points will also be relations on $\mathscr{T}_f$. For subtyping on arbitrary (finite or infinite) trees, the universe is $\mathscr{T} \times \mathscr{T}$.

[1] The locution 'tree type' is slightly awkward, but it will help to keep things straight when we discuss the alternative presentation of recursive types as finite expressions involving μ ('μ-types') in Section 9.

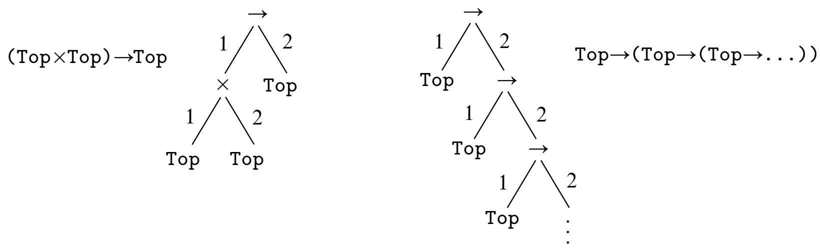


Fig. 2. Sample tree types.

Definition 4.1 [Finite subtyping]
Two finite tree types S and T are in the subtype relation ('S is a subtype of T') if $(\mathtt{S},\mathtt{T}) \in \mu S_f$, where the monotone function $S_f \in \mathscr{P}(\mathscr{T}_f \times \mathscr{T}_f) \rightarrow \mathscr{P}(\mathscr{T}_f \times \mathscr{T}_f)$ is defined by

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

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

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

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

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

The statement S <: T above the line in the second and third rules should be read as 'if the pair (S, T) is in the argument to S_f' and below the line as 'then (S, T) is in the result'.

Definition 4.2 [Infinite subtyping]
Two (finite or infinite) tree types S and T are in the subtype relation if $(\mathtt{S},\mathtt{T}) \in \nu S$, where $S \in \mathscr{P}(\mathscr{T} \times \mathscr{T}) \rightarrow \mathscr{P}(\mathscr{T} \times \mathscr{T})$ is defined by:

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

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.

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

Exercise 4.4
Is there a pair of types (S, T) that is related by νS, but not by μS? What about a pair of types (S, T) that is related by νS_f, but not by μS_f?

One fundamental property of the subtype relation on infinite tree types – the fact that it is transitive – should be verified right away. If the subtype relation were *not* transitive, the critical property of preservation of types under evaluation 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` could be typed, using the rule of subsumption once for each application, but this term reduces in one step to the ill-typed term `f s`.

Definition 4.5
A relation $R \subseteq \mathscr{U} \times \mathscr{U}$ is *transitive* if R is closed under the monotone function $TR(R) = \{(x, y) \mid \exists z \in \mathscr{U}. (x, z), (z, y) \in R\}$, i.e. if $TR(R) \subseteq R$.

Lemma 4.6
Let $F \in \mathscr{P}(\mathscr{U} \times \mathscr{U}) \to \mathscr{P}(\mathscr{U} \times \mathscr{U})$ be a monotone function. If $TR(F(R)) \subseteq F(TR(R))$ for any $R \subseteq \mathscr{U} \times \mathscr{U}$, then νF is transitive.

Proof
Since νF is a fixed point, $\nu F = F(\nu F)$, implying $TR(\nu F) = TR(F(\nu F))$. Therefore, by the lemma's assumption, $TR(\nu F) \subseteq F(TR(\nu F))$. In other words, $TR(\nu F)$ is F-consistent, so, by the principle of coinduction, $TR(\nu F) \subseteq \nu F$. Equivalently, νF is transitive by Definition 4.5. □

This lemma is reminiscent of the traditional technique for establishing redundancy of the transitivity rule in inference systems, often called 'cut-elimination proofs.' The condition $TR(F(R)) \subseteq F(TR(R))$ corresponds to the crucial step in this technique: given that a certain statement can be obtained by taking some statements from R, applying rules from F, and then applying the rule of transitivity TR, we argue that the statement can instead be obtained by reversing the steps – first applying the rule of transitivity, and then rules from F. We use the lemma to establish transitivity of the subtype relation.

Theorem 4.7
νS is transitive.

Proof
By Lemma 4.6, it suffices to show that $TR(S(R)) \subseteq S(TR(R))$ for any $R \subseteq \mathscr{T} \times \mathscr{T}$. Let $(\mathtt{S}, \mathtt{T}) \in TR(S(R))$. By the definition of TR, there exists some $\mathtt{U} \in \mathscr{T}$ such that $(\mathtt{S}, \mathtt{U}), (\mathtt{U}, \mathtt{T}) \in S(R)$. Our goal is to show that $(\mathtt{S}, \mathtt{T}) \in S(TR(R))$. Consider the possible shapes of `U`.

Case: $\mathtt{U} = \mathtt{Top}$

Since $(\mathtt{U}, \mathtt{T}) \in S(R)$, the definition of S implies that `T` must be `Top`. But $(\mathtt{A}, \mathtt{Top}) \in S(Q)$ for any `A` and Q; in particular, $(\mathtt{S}, \mathtt{T}) = (\mathtt{S}, \mathtt{Top}) \in S(TR(R))$.

Case: $\mathtt{U} = \mathtt{U}_1 \times \mathtt{U}_2$

If $\mathtt{T} = \mathtt{Top}$, then $(\mathtt{S},\mathtt{T}) \in S(TR(R))$ as in the previous case. Otherwise, $(\mathtt{U},\mathtt{T}) \in S(R)$ implies $\mathtt{T} = \mathtt{T}_1 \times \mathtt{T}_2$, with $(\mathtt{U}_1,\mathtt{T}_1)$, $(\mathtt{U}_2,\mathtt{T}_2) \in R$. Similarly, $(\mathtt{S},\mathtt{U}) \in S(R)$ implies $\mathtt{S} = \mathtt{S}_1 \times \mathtt{S}_2$, with $(\mathtt{S}_1,\mathtt{U}_1)$, $(\mathtt{S}_2,\mathtt{U}_2) \in R$. By the definition of TR, we have $(\mathtt{S}_1,\mathtt{T}_1)$, $(\mathtt{S}_2,\mathtt{T}_2) \in TR(R)$, from which $(\mathtt{S}_1 \times \mathtt{S}_2, \mathtt{T}_1 \times \mathtt{T}_2) \in S(TR(R))$ follows from the definition of S.

Case: $\mathtt{U} = \mathtt{U}_1 \to \mathtt{U}_2$

Similar. □

Exercise 4.8
Show that the subtype relation on infinite tree types is also reflexive.

The following section continues the discussion of transitivity by comparing its treatment in standard accounts of subtyping for finite types and in the present account of subtyping for infinite tree types. It can be skipped or skimmed on a first reading.

5 A digression on transitivity

Standard formulations of inductively defined subtype 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 presentations are fairly similar; in more complex systems, they can be quite different, and proving that they define the same relation on types can pose a significant challenge.

One of the most distinctive differences between declarative and algorithmic presentations is that declarative presentations include an explicit rule of transitivity – if `S<:U` and `U<:T` then `S<:T` – while algorithmic systems do not. 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 makes it obvious to the reader that the subtype relation is, indeed, transitive. Secondly, 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 one that takes width, depth, and permutation into account all at once.

Somewhat surprisingly, the possibility of giving a declarative presentation with the rule of transitivity turns out to be a consequence of a 'trick' that can be played with inductive, but not coinductive, definitions. To see why, observe that the property of transitivity is a *closure property* – it demands that the subtype relation be closed under the transitivity rule. Since the subtype relation for finite types is itself defined as the closure of a set of rules, we can achieve closure under transitivity simply by adding it to the other rules. This is a general property of inductive definitions and closure properties: the union of two sets of rules, when applied inductively, generates the least relation that is closed under both sets of rules separately. This fact can be formulated more abstractly in terms of generating functions:

Proposition 5.1
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.

Proof
First, we show that μH is closed under both F and G. By definition, $\mu H = H(\mu H) = F(\mu H) \cup G(\mu H)$, so $F(\mu H) \subseteq \mu H$ and $G(\mu H) \subseteq \mu H$. Secondly, we show that μH is the *least* set closed under both F and G. Suppose there is some set X such that $F(X) \subseteq X$ and $G(X) \subseteq X$. Then $H(X) = F(X) \cup G(X) \subseteq X$, that is, X is H-closed. Since μH is the least H-closed set (by the Knaster-Tarski theorem), we have $\mu H \subseteq X$. □

Unfortunately, this trick for achieving transitive closure does not work when we are dealing with coinductive definitions. As the following exercise shows, adding transitivity to the rules generating a coinductively defined relation always gives us a degenerate relation.

Exercise 5.2
Suppose F is a generating function on the universe $\mathcal{U}$. Show that the greatest fixed point νF^{TR} of the generating function

$$F^{TR}(R) = F(R) \cup TR(R)$$

is the *total* relation on $\mathcal{U} \times \mathcal{U}$.

In the coinductive setting, then, we drop declarative presentations and work just with algorithmic ones.

6 Membership checking

We now turn our attention to the central question of the paper: 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. Membership checking for least fixed points is addressed more briefly (in Exercise 6.13).

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 of the monotonicity of F, any superset of a generating set for x is also a generating set for x, so it makes sense to restrict our attention to minimal generating sets. Going one step further, we can focus on the class of 'invertible' generating functions, where each x has at most one minimal generating set.

Definition 6.1
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 member that is a subset of all the others. When F is invertible, 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 *support* 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 similar functions based on F that we define later).

Exercise 6.2
Verify that S_f and S, the generating functions for the subtyping relations from Definitions 4.1 and 4.2, are invertible, and give their support functions.

Our goal is to develop algorithms for checking membership in the least and greatest fixed points of a generating function F. The basic steps in these algorithms will involve 'running F backwards': to check membership for an element x, we need to ask how x could have been generated by F. The advantage of an invertible F is that there is at most one way to generate a given x. For a non-invertible F, elements can be generated in multiple ways, leading to a combinatorial explosion in the number of paths that the algorithm must explore. From now on, we restrict our attention to invertible generating functions.

Definition 6.3
An element x is *F-supported* if $support_F(x)\downarrow$; otherwise, x is *F-unsupported*. An F-supported element is called *F-ground* if $support_F(x) = \emptyset$.

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

An invertible function can be visualized as a *support graph*. For example, figure 3 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, the set $support_E(x)$ contains every 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, even though its support set includes an unsupported element.)

Exercise 6.4
Give 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.

Thinking about the graph in figure 3 suggests the idea that an element x is

[2] As usual, the symbol $\uparrow$ means 'undefined', and the notation $f(x)\uparrow$ says that the function f is undefined at x, while $f(x)\downarrow$ says that f is defined at x.

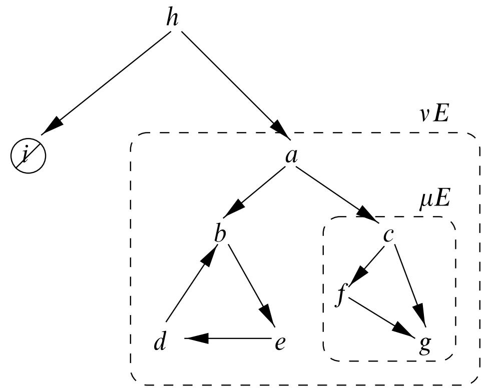


Fig. 3. A sample *support* function.

in the greatest fixed point iff no unsupported element is reachable from x in the support graph. 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 must take some precautions against falling into an infinite loop. We will pursue this idea for the remainder of this section.

Definition 6.5

Suppose F is an invertible generating function. Define the boolean-valued function gfp_F (or just gfp) as follows:[3]

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

Intuitively, gfp starts from X and keeps enriching it using *support* until either it becomes consistent or else an unsupported element is found. We extend gfp to individual elements by taking $gfp(x) = gfp(\{x\})$.

Exercise 6.6

Another observation that can be made from figure 3 is that an element x of νF is not a member of μF if x participates in a cycle in the support graph (or if there is a path from x to an element that participates in a cycle). Is the converse also true – that is, if x is a member of νF but not μF, is it necessarily the case that x leads to a cycle?

The remainder of the section is devoted to proving the correctness and termination

[3] We use here the standard notation for defining recursive functions, i.e. we intend that gfp is the *smallest* partial function satisfying the stated equation. Such definitions can themselves be viewed more formally as least fixed points of appropriate generating functions. Details can be found in any standard treatment of denotational semantics, e.g. the in texts of Gunter (1992), Winskel (1993) or Mitchell (1996).

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.

Lemma 6.7
$X \subseteq F(Y)$ iff $support_F(X)\downarrow$ and $support_F(X) \subseteq Y$.

Proof
It suffices to show that $x \in F(Y)$ iff $support(x)\downarrow$ and $support(x) \subseteq Y$. Suppose first that $x \in F(Y)$. Then $Y \in G_x = \{X \subseteq \mathcal{U} \mid x \in F(X)\}$ – that is, $G_x \neq \emptyset$. Therefore, since F is invertible, $support(x)$, the smallest set in G_x, exists and $support(x) \subseteq Y$. Conversely, if $support(x) \subseteq Y$, then $F(support(x)) \subseteq F(Y)$ by monotonicity. But $x \in F(support(x))$ by the definition of *support*, so $x \in F(Y)$. □

Lemma 6.8
Suppose P is a fixed point of F. Then $X \subseteq P$ iff $support_F(X)\downarrow$ and $support_F(X) \subseteq P$.

Proof
Recall that $P = F(P)$ and apply Lemma 6.7. □

Now we can prove partial correctness of *gfp*. (We are not concerned with total correctness yet, because some generating functions will make *gfp* diverge. We prove termination for a restricted class of generating functions later in the section.)

Theorem 6.9

1. If $gfp_F(X) = true$, then $X \subseteq \nu F$.
2. If $gfp_F(X) = false$, then $X \nsubseteq \nu F$.

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 6.7, 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. Again, there are two ways to get $gfp(X) = false$. Suppose first that $gfp(X) = false$ because $support(X)\uparrow$. Then $X \nsubseteq \nu F$ by Lemma 6.8. On the other hand, suppose $gfp(X) = false$ because $gfp(support(X) \cup X) = false$. By the induction hypothesis, $support(X) \cup X \nsubseteq \nu F$. Equivalently, $X \nsubseteq \nu F$ or $support(X) \nsubseteq \nu F$. Either way, $X \nsubseteq \nu F$ (using Lemma 6.8 in the second case). □

Next, we identify a sufficient termination condition for *gfp*, giving a class of generating functions for which the algorithm is guaranteed to terminate. To describe the class, we need some additional terminology.

Definition 6.10
Given an invertible generating function F and an element $x \in \mathcal{U}$, the set $pred_F(x)$ (or just $pred(x)$) of immediate predecessors of x is

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

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

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

The set $reachable_F(X)$ (or just $reachable(X)$) of all elements reachable from a set X via *support* is defined as

$$reachable(X) = \bigcup_{n \geqslant 0} pred^n(X).$$

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

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

An element $y \in \mathcal{U}$ is *reachable* from an element x if $y \in reachable(x)$.

Definition 6.11
An invertible generating function F is said to be *finite state* if $reachable(x)$ is finite for each $x \in \mathcal{U}$.

For a finite-state generating function, the search space explored by gfp is finite and gfp always terminates:

Theorem 6.12
If $reachable_F(X)$ is finite, then $gfp_F(X)$ is defined. Consequently, if F is finite state, then $gfp_F(X)$ terminates for any finite $X \subseteq \mathcal{U}$.

Proof
For each 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. □

Exercise 6.13
Suppose F is an invertible generating function. Define the function lfp_F (or just lfp) as follows:

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

Intuitively, lfp works by starting with a set X and using the *support* relation to reduce it until it becomes empty. Prove that this algorithm is partially correct, in the sense that

1. If $lfp_F(X) = true$, then $X \subseteq \mu F$.
2. If $lfp_F(X) = false$, then $X \nsubseteq \mu F$.

Can you find a class of generating functions for which lfp_F is guaranteed to terminate on all finite inputs?

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. For example, in the following trace of *gfp* on the function E from figure 3,

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

Note that $support(a)$ is recomputed 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.

Definition 7.1

Suppose F is an invertible generating function. Define the function gfp^a_F (or just gfp^a) as follows (the superscript 'a' is for 'assumptions'):

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

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

This algorithm (like the two following algorithms in this section) computes the support of each element at most once. A trace for the above example looks like this:

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

Naturally, the correctness statement for this algorithm is slightly more elaborate than the one we saw in the previous section.

Theorem 7.2

1. If $support_F(A) \subseteq A \cup X$ and $gfp^a_F(A,X) = true$, then $A \cup X \subseteq \nu F$.
2. If $gfp^a_F(A,X) = false$, then $X \nsubseteq \nu F$.

Proof
Similar to Theorem 6.9. □

The rest of this section examines two more variations on the *gfp* algorithm that correspond more closely to well-known subtyping algorithms for recursive types. First-time readers may want to skip to the beginning of the next section.

Definition 7.3
A small variation on gfp^a has the algorithm pick just one element at a time from X and expand its *support*. The new algorithm is called gfp^s_F (or just gfp^s, '*s*' being for 'single').

$$
\begin{array}{lll}
gfp^s(A,X) & = & \text{if } X = \emptyset, \text{ then } true \\
& & \text{else let } x \text{ be some element of } X \text{ in} \\
& & \quad \text{if } x \in A \text{ then } gfp^s(A,\, X \setminus \{x\}) \\
& & \quad \text{else if } support(x) \uparrow \text{ then } false \\
& & \quad \text{else } gfp^s(A \cup \{x\},\, (X \cup support(x)) \setminus (A \cup \{x\})).
\end{array}
$$

The correctness statement (i.e. the invariant of the recursive 'loop') for this algorithm is exactly the same as Theorem 7.2.

Unlike the above algorithm, many existing algorithms for recursive subtyping take just one candidate element, rather than a set, as an argument. Another small modification to our algorithm makes it more similar to these. The modified algorithm is no longer tail recursive,[4] since it uses the call stack to remember subgoals that have not yet been checked. Another change is that the algorithm both takes a set of assumptions A as an argument and returns a new set of assumptions as a result. This allows it to record the subtyping assumptions that have been generated during completed recursive calls and reuse them in later calls. In effect, the set of assumptions is 'threaded' through the recursive call graph – whence the name of the algorithm, gfp^t.

Definition 7.4
Given an invertible generating function F, define the function gfp^t_F (or just gfp^t) as follows:

[4] A *tail-recursive* call (or *tail call*) is a recursive call that is the last action of the calling function, i.e. such that the result returned from the recursive call will also be caller's result. Tail calls are interesting because most compilers for functional languages will implement a tail call as a simple branch, re-using the stack space of the caller instead of allocating a new stack frame for the recursive call. This means that a loop implemented as a tail-recursive function compiles into the same machine code as an equivalent `while` loop.

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

To check $x \in \nu F$, compute $gfp^t(\emptyset, x)$. If this call succeeds, then $x \in \nu F$; if it fails, then $x \notin \nu F$. We use the following convention for failure: if an expression B fails, then 'let $A = B$ in C' also fails. This avoids writing explicit 'exception handling' clauses for every recursive invocation of gfp^t.

The correctness statement for this algorithm must again be refined from what we had above, taking into account the non-tail-recursive nature of this formulation by positing an extra 'stack' X of elements whose supports remain to be checked.

Lemma 7.5

1. If $gfp^t_F(A,x) = A'$, then $A \cup \{x\} \subseteq A'$.
2. For all X, if $support_F(A) \subseteq A \cup X \cup \{x\}$ and $gfp^t_F(A,x) = A'$, then $support_F(A') \subseteq A' \cup X$.

Proof

Part (1) is a routine induction on the recursive structure of a run of the algorithm.

Part (2) also goes by induction on the recursive structure of a run of the algorithm. If $x \in A$, then $A' = A$ and the desired conclusion follows immediately from the assumption. On the other hand, suppose $A' \neq A$, and consider the special case where $support(x)$ contains two elements x_1 and x_2 – the general case (not shown here) is proved similarly, using an inner induction on the size of $support(x)$. The algorithm calculates A_0, A_1, and A_2 and returns A_2. We want to show, for an arbitrary X_0, that if $support(A) \subseteq A \cup \{x\} \cup X_0$, then $support(A_2) \subseteq A_2 \cup X_0$. Let $X_1 = X_0 \cup \{x_2\}$. Since

$$
\begin{array}{rcl}
support(A_0) & = & support(A) \cup support(x) \\
& = & support(A) \cup \{x_1, x_2\} \\
& \subseteq & A \cup \{x\} \cup X_0 \cup \{x_1, x_2\} \\
& = & A_0 \cup X_0 \cup \{x_1, x_2\} \\
& = & A_0 \cup X_1 \cup \{x_1\},
\end{array}
$$

we can apply the induction hypothesis to the first recursive call by instantiating the universally quantified X with X_1. This yields $support(A_1) \subseteq A_1 \cup X_1 = A_1 \cup \{x_2\} \cup X_0$. Now, we can apply the induction hypothesis to the second recursive call by instantiating the universally quantified X with X_0 to obtain the desired result: $support(A_2) \subseteq A_2 \cup X_0$. □

Theorem 7.6

1. If $gfp^t_F(\emptyset, x) = A'$, then $x \in \nu F$.
2. If $gfp^t_F(\emptyset, x) = fail$, then $x \notin \nu F$.

Proof

For part (1), observe that, by Lemma 7.5(1), $x \in A'$. Instantiating part (2) of the lemma with $X = \emptyset$, we obtain $support(A') \subseteq A'$, that is, A' is F-consistent by Lemma 6.7, and so $A' \subseteq \nu F$ by coinduction. For part (2), we argue (by an easy induction on the depth of a run of the gfp^t_F algorithm, using Lemma 6.8) that if, for some A, we have $gfp^t_F(A, x) = fail$, then $x \notin \nu F$. □

Since all of the algorithms in this section examine the reachable set, a sufficient 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 set defined as the greatest fixed point of a generating function F, assuming that F is invertible and finite state; separately, we have 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 our 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, so-called *regular types,* then the sets of reachable states will be guaranteed to remain finite and the subtype checking algorithm will always terminate.

Definition 8.1

A tree type `S` is a *subtree* of a tree type `T` if $\mathtt{S} = \lambda\sigma.\ \mathtt{T}(\pi \cdot \sigma)$ for some π, that is, if the function `S` from paths to symbols can be obtained from the function `T` by adding some constant prefix π to the argument paths we give to `T`; the prefix π corresponds to the path from the root of `T` to the root of `S`. We write *subtrees*(`T`) for the set of all subtrees of `T`.

Definition 8.2

A tree type $\mathtt{T} \in \mathscr{T}$ is **regular** if *subtrees*(`T`) is finite, i.e. if `T` has finitely many distinct subtrees. The set of regular tree types is written $\mathscr{T}_r$.

Examples

1. Every finite tree type is regular; the number of distinct subtrees is at most the number of nodes. The number of distinct subtrees of a tree type can be strictly less than the number of nodes. For example, $\mathtt{T} = \mathtt{Top}{\rightarrow}(\mathtt{Top}{\times}\mathtt{Top})$ has five nodes but only three distinct subtrees (`T` itself, $\mathtt{Top}{\times}\mathtt{Top}$, and `Top`).
2. Some infinite tree types 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`).

3. The tree 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. Because T is irregular, the set $\mathit{reachable}_S(\mathtt{T},\mathtt{T})$ containing all the subtyping pairs needed to justify the statement T<:T is infinite.

Proposition 8.4
The restriction S_r of the generating function S to regular tree types is finite state.

Proof
We need to show that for any pair (S,T) of regular tree types, the set $\mathit{reachable}_{S_r}(\mathtt{S},\mathtt{T})$ is finite. Observe that $\mathit{reachable}_{S_r}(\mathtt{S},\mathtt{T}) \subseteq \mathit{subtrees}(\mathtt{S}) \times \mathit{subtrees}(\mathtt{T})$; the latter is finite, since both $\mathit{subtrees}(\mathtt{S})$ and $\mathit{subtrees}(\mathtt{T})$ are. □

This means that we can obtain a decision procedure for the subtype relation on regular tree types by instantiating one of the membership algorithms with S. Naturally, for this to work in a practical implementation, regular trees must be represented by some finite structures. One such representation, μ-notation, is discussed in the next section.

9 μ-Types

This section develops the finite μ-notation, defines subtyping on μ-expressions, and establishes the correspondence between this notion of subtyping and the subtyping on tree types.

Definition 9.1
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 as usual.

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 subexpressions of the form $\mu\mathtt{X}.\mu\mathtt{X}_1\ldots\mu\mathtt{X}_n.\mathtt{X}$, where the variables $\mathtt{X}_1$ 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.

Definition 9.2
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. Equivalently, a raw μ-type is contractive if every occurrence of a μ-bound variable in the body is separated from its binder by at least one $\rightarrow$ or $\times$.

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

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.

Definition 9.3
The function *treeof*, mapping closed μ-types to tree types, is defined inductively as follows:

$$\begin{array}{lcl}
\mathit{treeof}(\mathtt{Top})(\bullet) & = & \mathtt{Top} \\[4pt]
\mathit{treeof}(\mathtt{T}_1{\rightarrow}\mathtt{T}_2)(\bullet) & = & \rightarrow \\
\mathit{treeof}(\mathtt{T}_1{\rightarrow}\mathtt{T}_2)(i\cdot\pi) & = & \mathit{treeof}(\mathtt{T}_i)(\pi) \\[4pt]
\mathit{treeof}(\mathtt{T}_1{\times}\mathtt{T}_2)(\bullet) & = & \times \\
\mathit{treeof}(\mathtt{T}_1{\times}\mathtt{T}_2)(i\cdot\pi) & = & \mathit{treeof}(\mathtt{T}_i)(\pi) \\[4pt]
\mathit{treeof}(\mu\mathtt{X}.\mathtt{T})(\pi) & = & \mathit{treeof}(\{\mathtt{X}\mapsto\mu\mathtt{X}.\mathtt{T}\}\mathtt{T})(\pi)
\end{array}$$

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

1. Every recursive use of *treeof* on the right-hand side reduces the lexicographic size of the pair $(|\pi|, \mu\text{-}\mathit{height}(\mathtt{T}))$: the cases for $\mathtt{S}{\rightarrow}\mathtt{T}$ and $\mathtt{S}{\times}\mathtt{T}$ reduce $|\pi|$ and the case for $\mu\mathtt{X}.\mathtt{T}$ preserves $|\pi|$ but reduces μ-*height*(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 unfolding step in the definition of $\mathit{treeof}(\mu\mathtt{X}.\mathtt{T})$.

The *treeof* function is lifted to pairs of types by defining $\mathit{treeof}(\mathtt{S},\mathtt{T}) = (\mathit{treeof}(\mathtt{S}), \mathit{treeof}(\mathtt{T}))$.

A sample application of *treeof* to a μ-type is shown in figure 4.

The subtype relation for tree types was defined in Section 4 as the greatest fixed point of the generating function S. In the present section, we extended the syntax of types with μ-types, whose behavior is intuitively described by the rules of (right and left, correspondingly) μ-*folding*:

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

Formally, we define subtyping for μ-types by giving a generating function S_m, with *treeof*(μ`X.((X×Top)→X)`) =

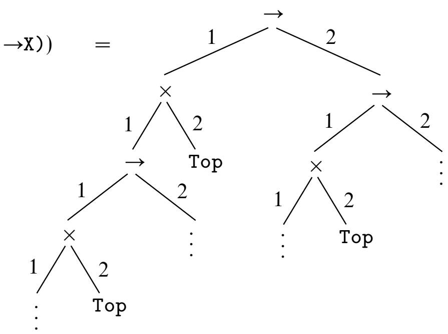


Fig. 4. Sample *treeof* application.

three clauses identical to the definition of S and two additional clauses corresponding to the μ-folding rules.

Definition 9.4

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

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

Note that this definition does not embody precisely the μ-folding rules above: we have introduced an asymmetry between its final and penultimate clauses to make it invertible (otherwise, the clauses would overlap). However, as the next exercise shows, S_m generates the same subtype relation as the more natural generating function[5] S_d whose clauses exactly correspond to the inference rules.

Exercise 9.5

Write down the function S_d mentioned above, and demonstrate that it is not invertible. Prove that $\nu S_d = \nu S_m$.

The generating function S_m is invertible because the corresponding support func-

[5] The 'd' in S_d is a reminder that the function is based on the 'declarative' inference rules for μ-folding, in contrast to the 'algorithmic' versions used in S_m.

tion is well-defined:

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

The subtype relation on μ-types so far has been introduced independently of the previously defined subtyping on tree types. Since we think of μ-types as just a way of representing tree types in a finite form, it is necessary to ensure that the two notions of subtyping correspond to each other. Theorem 9.7 establishes this correspondence. But first, we need a technical lemma.

Lemma 9.6

Suppose that $R \subseteq \mathcal{T}_m \times \mathcal{T}_m$ is S_m-consistent. For any $(\mathtt{S},\mathtt{T}) \in R$, there is some $(\mathtt{S}',\mathtt{T}') \in R$ such that $\mathit{treeof}(\mathtt{S}',\mathtt{T}') = \mathit{treeof}(\mathtt{S},\mathtt{T})$ and neither $\mathtt{S}'$ nor $\mathtt{T}'$ starts with μ.

Proof

By induction on the total number of μs at the front of $\mathtt{S}$ and $\mathtt{T}$. If neither $\mathtt{S}$ nor $\mathtt{T}$ starts with μ, then we can take $(\mathtt{S}',\mathtt{T}') = (\mathtt{S},\mathtt{T})$. On the other hand, if $(\mathtt{S},\mathtt{T}) = (\mathtt{S},\mu\mathtt{X}.\mathtt{T}_1)$, then by the S_m-consistency of R, we have $(\mathtt{S},\mathtt{T}) \in S_m(R)$, so $(\mathtt{S}'',\mathtt{T}'') = (\mathtt{S},\{\mathtt{X}\mapsto \mu\mathtt{X}.\mathtt{T}_1\}\mathtt{T}_1) \in R$. Since $\mathtt{T}$ is contractive, the result $\mathtt{T}''$ of unfolding $\mathtt{T}$ has one fewer μ at the front than $\mathtt{T}$ does. By the induction hypothesis, there is some $(\mathtt{S}',\mathtt{T}') \in R$ such that neither $\mathtt{S}'$ nor $\mathtt{T}'$ starts with μ and such that $\mathit{treeof}(\mathtt{S}'',\mathtt{T}'') = (\mathtt{S}',\mathtt{T}')$. Since, by the definition of *treeof*, $\mathit{treeof}(\mathtt{S},\mathtt{T}) = \mathit{treeof}(\mathtt{S}'',\mathtt{T}'')$, the pair $(\mathtt{S}',\mathtt{T}')$ is the one we need. The case where $(\mathtt{S},\mathtt{T}) = (\mu\mathtt{X}.\mathtt{S}_1,\mathtt{T})$ is similar. □

Theorem 9.7

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

Proof

First, let us consider the 'only if' direction – that $(\mathtt{S},\mathtt{T}) \in \nu S_m$ implies $\mathit{treeof}(\mathtt{S},\mathtt{T}) \in \nu S$. Let $(\mathtt{A},\mathtt{B}) = \mathit{treeof}(\mathtt{S},\mathtt{T}) \in \mathcal{T}\times\mathcal{T}$. By the coinduction principle, the result will follow if we can exhibit an S-consistent set $Q \in \mathcal{T}\times\mathcal{T}$ such that $(\mathtt{A},\mathtt{B}) \in Q$. Our claim is that $Q = \mathit{treeof}(\nu S_m)$ is such a set. To verify this, we must show that $(\mathtt{A}',\mathtt{B}') \in S(Q)$ for every $(\mathtt{A}',\mathtt{B}') \in Q$.

Let $(\mathtt{S}',\mathtt{T}') \in \nu S_m$ be a pair of μ-types such that $\mathit{treeof}(\mathtt{S}',\mathtt{T}') = (\mathtt{A}',\mathtt{B}')$. By Lemma 9.6, we may assume that neither $\mathtt{S}'$ nor $\mathtt{T}'$ starts with μ. Since νS_m is S_m-consistent, $(\mathtt{S}',\mathtt{T}')$ must be supported by one of the clauses in the definition of S_m, i.e. it must have one of the following shapes.

Case: $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}',\mathtt{Top})$

Then $\mathtt{B}' = \mathtt{Top}$, and $(\mathtt{A}',\mathtt{B}') \in S(Q)$ by the definition of S.

Case: $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}_1\times\mathtt{S}_2, \mathtt{T}_1\times\mathtt{T}_2)$ with $(\mathtt{S}_1,\mathtt{T}_1),(\mathtt{S}_2,\mathtt{T}_2) \in \nu S_m$

By the definition of *treeof*, we have $\mathtt{B}' = \mathit{treeof}(\mathtt{T}') = \mathtt{B}_1\times\mathtt{B}_2$, where each $\mathtt{B}_i = \mathit{treeof}(\mathtt{T}_i)$. Similarly, $\mathtt{A}' = \mathtt{A}_1\times\mathtt{A}_2$, where $\mathtt{A}_i = \mathit{treeof}(\mathtt{S}_i)$. Applying *treeof* to these pairs gives $(\mathtt{A}_1,\mathtt{B}_1),(\mathtt{A}_2,\mathtt{B}_2) \in Q$. But then, by the definition of S, we have $(\mathtt{A},\mathtt{B}) = (\mathtt{A}_1\times\mathtt{A}_2, \mathtt{B}_1\times\mathtt{B}_2) \in S(Q)$.

Case: $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}_1\to\mathtt{S}_2, \mathtt{T}_1\to\mathtt{T}_2)$ with $(\mathtt{T}_1,\mathtt{S}_1),(\mathtt{S}_2,\mathtt{T}_2) \in \nu S_m$

Similar.

Next, let us check the 'if' direction of the theorem – that $\mathit{treeof}(\mathtt{S},\mathtt{T}) \in \nu S$ implies $(\mathtt{S},\mathtt{T}) \in \nu S_m$. By the coinduction principle, it suffices to exhibit an S_m-consistent set $R \in \mathscr{T}_m \times \mathscr{T}_m$ with $(\mathtt{S},\mathtt{T}) \in R$. We claim that $R = \{(\mathtt{S}',\mathtt{T}') \in \mathscr{T}_m \times \mathscr{T}_m \mid \mathit{treeof}(\mathtt{S}',\mathtt{T}') \in \nu S\}$ is such a set. Clearly, $(\mathtt{S},\mathtt{T}) \in R$. To finish the proof, we must now show that $(\mathtt{S}',\mathtt{T}') \in R$ implies $(\mathtt{S}',\mathtt{T}') \in S_m(R)$.

Note that, since νS is S-consistent, any pair $(\mathtt{A}',\mathtt{B}') \in \nu S$ must have one of the forms $(\mathtt{A}',\mathtt{Top})$, $(\mathtt{A}_1\times\mathtt{A}_2, \mathtt{B}_1\times\mathtt{B}_2)$, or $(\mathtt{A}_1\to\mathtt{A}_2, \mathtt{B}_1\to\mathtt{B}_2)$. From this and the definition of *treeof*, we see that any pair $(\mathtt{S}',\mathtt{T}') \in R$ must have one of the forms $(\mathtt{S}',\mathtt{Top})$, $(\mathtt{S}_1\times\mathtt{S}_2, \mathtt{T}_1\times\mathtt{T}_2)$, $(\mathtt{S}_1\to\mathtt{S}_2, \mathtt{T}_1\to\mathtt{T}_2)$, $(\mathtt{S}', \mu\mathtt{X}.\mathtt{T}_1)$, or $(\mu\mathtt{X}.\mathtt{S}_1, \mathtt{T}')$. We consider each of these cases in turn.

Case: $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}',\mathtt{Top})$

Then $(\mathtt{S}',\mathtt{T}') \in S_m(R)$ immediately, by the definition of S_m.

Case: $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}_1\times\mathtt{S}_2, \mathtt{T}_1\times\mathtt{T}_2)$

Let $(\mathtt{A}',\mathtt{B}') = \mathit{treeof}(\mathtt{S}',\mathtt{T}')$. Then $(\mathtt{A}',\mathtt{B}') = (\mathtt{A}_1\times\mathtt{A}_2, \mathtt{B}_1\times\mathtt{B}_2)$, with $\mathtt{A}_i = \mathit{treeof}(\mathtt{S}_i)$ and $\mathtt{B}_i = \mathit{treeof}(\mathtt{T}_i)$. Since $(\mathtt{A}',\mathtt{B}') \in \nu S$, the S-consistency of νS implies that $(\mathtt{A}_i,\mathtt{B}_i) \in \nu S$, which in turn yields $(\mathtt{S}_i,\mathtt{T}_i) \in R$, by the definition of R. The definition of S_m yields $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}_1\times\mathtt{S}_2, \mathtt{T}_1\times\mathtt{T}_2) \in S_m(R)$.

Case: $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}_1\to\mathtt{S}_2, \mathtt{T}_1\to\mathtt{T}_2)$

Similar.

Case: $(\mathtt{S}',\mathtt{T}') = (\mathtt{S}', \mu\mathtt{X}.\mathtt{T}_1)$

Let $\mathtt{T}'' = \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}_1\}\mathtt{T}_1$. By definition, $\mathit{treeof}(\mathtt{T}'') = \mathit{treeof}(\mathtt{T}')$. Therefore, by the definition of R, we have $(\mathtt{S}',\mathtt{T}'') \in R$, and so $(\mathtt{S}',\mathtt{T}') \in S_m(R)$, by the definition of S_m.

Case: $(\mathtt{S}',\mathtt{T}') = (\mu\mathtt{X}.\mathtt{S}_1, \mathtt{T}')$

If $\mathtt{T}' = \mathtt{Top}$ or $\mathtt{T}'$ starts with μ, then one of the cases above applies; otherwise, the argument is similar to the previous one. □

The correspondence established by the theorem is a statement of soundness and completeness of subtyping between μ-types, as defined in this section, with respect to the ordinary subtype relation between infinite tree types, restricted to those tree types that can be represented by finite μ-expressions.

10 Counting subexpressions

Instantiating the generic algorithm gfp^t (7.4) with the specific support function $\mathit{support}_{S_m}$ for the subtype relation on μ-types (9.4) yields the subtyping algorithm

$$
\begin{array}{lll}
subtype(A, \mathtt{S}, \mathtt{T}) & = & \text{if } (\mathtt{S}, \mathtt{T}) \in A, \text{ then} \\
& & \quad A \\
& & \text{else let } A_0 = A \cup \{(\mathtt{S}, \mathtt{T})\} \text{ in} \\
& & \quad \text{if } \mathtt{T} = \mathtt{Top}, \text{ then} \\
& & \quad\quad A_0 \\
& & \quad \text{else if } \mathtt{S} = \mathtt{S}_1 \times \mathtt{S}_2 \text{ and } \mathtt{T} = \mathtt{T}_1 \times \mathtt{T}_2, \text{ then} \\
& & \quad\quad \text{let } A_1 = subtype(A_0, \mathtt{S}_1, \mathtt{T}_1) \text{ in} \\
& & \quad\quad subtype(A_1, \mathtt{S}_2, \mathtt{T}_2) \\
& & \quad \text{else if } \mathtt{S} = \mathtt{S}_1 \to \mathtt{S}_2 \text{ and } \mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2, \text{ then} \\
& & \quad\quad \text{let } A_1 = subtype(A_0, \mathtt{T}_1, \mathtt{S}_1) \text{ in} \\
& & \quad\quad subtype(A_1, \mathtt{S}_2, \mathtt{T}_2) \\
& & \quad \text{else if } \mathtt{T} = \mu\mathtt{X}.\mathtt{T}_1, \text{ then} \\
& & \quad\quad subtype(A_0, \mathtt{S}, \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}_1\}\mathtt{T}_1) \\
& & \quad \text{else if } \mathtt{S} = \mu\mathtt{X}.\mathtt{S}_1, \text{ then} \\
& & \quad\quad subtype(A_0, \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{S}_1\}\mathtt{S}_1, \mathtt{T}) \\
& & \quad \text{else} \\
& & \quad\quad fail
\end{array}
$$

Fig. 5. Concrete subtyping algorithm for μ-types.

shown in figure 5. The argument in Section 7 shows that the termination of this algorithm can be guaranteed if $reachable_{S_m}(\mathtt{S}, \mathtt{T})$ is finite for any pair of μ-types $(\mathtt{S}, \mathtt{T})$. The present section is devoted to proving that this is the case (Proposition 10.11).

At first glance, the property seems almost obvious, but proving it rigorously requires a surprising amount of work. The difficulty is that there are two possible ways of defining the set of 'closed subexpressions' of a μ-type. One, which we call *top-down* subexpressions, directly corresponds to the subexpressions generated by $support_{S_m}$. The other, called *bottom-up* subexpressions, supports a straightforward proof 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 (Proposition 10.10). The development here is based on Brandt & Henglein's (1997).

Definition 10.1
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:

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

Exercise 10.2
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 $\mathtt{T}$, all the pairs contained in $support_{S_m}(\mathtt{S},\mathtt{T})$ are formed from top-down subexpressions of $\mathtt{S}$ and $\mathtt{T}$:

Lemma 10.3
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 $\mathtt{T}' \sqsubseteq \mathtt{T}$.

Proof
Straightforward inspection of the definition of $support_{S_m}$. □

Also, the top-down subexpression relation is transitive:

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

Proof
The statement of the lemma is equivalent to $\forall \mathtt{U},\mathtt{T}.\ \mathtt{U} \sqsubseteq \mathtt{T} \Rightarrow (\forall \mathtt{S}.\ \mathtt{S} \sqsubseteq \mathtt{U} \Rightarrow \mathtt{S} \sqsubseteq \mathtt{T})$. In other words, we must show that $\mu(TD) \subseteq R$, where $R = \{(\mathtt{U},\mathtt{T}) \mid \forall \mathtt{S}.\ \mathtt{S} \sqsubseteq \mathtt{U} \Rightarrow \mathtt{S} \sqsubseteq \mathtt{T}\}$. By the induction principle, it suffices to show that R is TD-closed, that is, that $TD(R) \subseteq R$. So suppose $(\mathtt{U},\mathtt{T}) \in TD(R)$. Proceed by cases on the clauses in the definition of TD.

Case: $(\mathtt{U},\mathtt{T}) = (\mathtt{T},\mathtt{T})$

Clearly, $(\mathtt{T},\mathtt{T}) \in R$.

Case: $(\mathtt{U},\mathtt{T}) = (\mathtt{U},\mathtt{T}_1{\times}\mathtt{T}_2)$ and $(\mathtt{U},\mathtt{T}_1) \in R$

Since $(\mathtt{U},\mathtt{T}_1) \in R$, it must be the case that $\mathtt{S} \sqsubseteq \mathtt{U} \Rightarrow \mathtt{S} \sqsubseteq \mathtt{T}_1$ for all $\mathtt{S}$. By the definition of $\sqsubseteq$, it must also be the case that $\mathtt{S} \sqsubseteq \mathtt{U} \Rightarrow \mathtt{S} \sqsubseteq \mathtt{T}_1{\times}\mathtt{T}_2$ for all $\mathtt{S}$. Thus, $(\mathtt{U},\mathtt{T}) = (\mathtt{U},\mathtt{T}_1{\times}\mathtt{T}_2) \in R$, by the definition of R.

Other cases:

Similar. □

Combining the two previous lemmas gives us the proposition that motivates the introduction of top-down subexpressions:

Proposition 10.5
If $(\mathtt{S}',\mathtt{T}') \in reachable_{S_m}(\mathtt{S},\mathtt{T})$, then $\mathtt{S}' \sqsubseteq \mathtt{S}$ or $\mathtt{S}' \sqsubseteq \mathtt{T}$, and $\mathtt{T}' \sqsubseteq \mathtt{S}$ or $\mathtt{T}' \sqsubseteq \mathtt{T}$.

Proof
By induction on the definition of $reachable_{S_m}$, using transitivity of $\sqsubseteq$. □

The finiteness of $reachable_{S_m}(\mathtt{S},\mathtt{T})$ will follow (in Proposition 10.11) 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 the definition of TD does not work because the last clause of TD breaks the induction: to construct the subexpressions of $\mathtt{U} = \mu\mathtt{X}.\mathtt{T}$, it refers to a potentially *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 change will lead to a simple proof of finiteness.

Definition 10.6
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:

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

This new definition of subexpressions differs from the old one 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.

Exercise 10.7
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.

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

Proof
Straightforward structural induction on $\mathtt{T}$, using the following observations, which follow from the definition of BU and $\preceq$:

- if $\mathtt{T} = \mathtt{Top}$ or $\mathtt{T} = \mathtt{X}$ then $\{\mathtt{S} \mid \mathtt{S} \preceq \mathtt{T}\} = \{\mathtt{T}\}$;
- if $\mathtt{T} = \mathtt{T}_1{\times}\mathtt{T}_2$ or $\mathtt{T} = \mathtt{T}_1{\rightarrow}\mathtt{T}_2$ then $\{\mathtt{S} \mid \mathtt{S} \preceq \mathtt{T}\} = \{\mathtt{T}\} \cup \{\mathtt{S} \mid \mathtt{S} \preceq \mathtt{T}_1\} \cup \{\mathtt{S} \mid \mathtt{S} \preceq \mathtt{T}_2\}$;
- if $\mathtt{T} = \mu\mathtt{X}.\mathtt{T}'$ then $\{\mathtt{S} \mid \mathtt{S} \preceq \mathtt{T}\} = \{\mathtt{T}\} \cup \{\{\mathtt{X} \mapsto \mathtt{T}\}\mathtt{S} \mid \mathtt{S} \preceq \mathtt{T}'\}$. □

To prove that the bottom-up subexpressions of a type include its top-down subexpressions, we will need the following lemma relating bottom-up subexpressions and substitution.

Lemma 10.9
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 $\mathtt{S}' \preceq \mathtt{T}$.

Proof
By structural induction on $\mathtt{T}$.

Case: $\mathtt{T} = \mathtt{Top}$

Only the reflexivity clause of BU allows $\mathtt{Top}$ as the right-hand element of the pair, so we must have $\mathtt{S} = \mathtt{Top}$. Taking $\mathtt{S}' = \mathtt{Top}$ yields the desired result.

Case: $\mathtt{T} = \mathtt{Y}$

If $\mathtt{Y} = \mathtt{X}$, we have $\mathtt{S} \preceq \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T} = \mathtt{Q}$, and the desired result holds by assumption. If $\mathtt{Y} \neq \mathtt{X}$, we have $\mathtt{S} = \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T} = \mathtt{Y}$. Only the reflexivity clause of *BU* can justify this pair, so we must have $\mathtt{S} = \mathtt{Y}$. Take $\mathtt{S}' = \mathtt{Y}$ to get the desired result.

Case: $\mathtt{T} = \mathtt{T}_1 \times \mathtt{T}_2$

We have $\mathtt{S} \preceq \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T} = \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}_1 \times \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}_2$. According to the definition of *BU*, there are three ways in which $\mathtt{S}$ can be a bottom-up subexpression of this product type. We consider each in turn.

Subcase: $\mathtt{S} = \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}$

Then take $\mathtt{S}' = \mathtt{T}$.

Subcase: $\mathtt{S} \preceq \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}_1$

By the induction hypothesis, either $\mathtt{S} \preceq \mathtt{Q}$ (in which case we are done) or else $\mathtt{S} = \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{S}'$ for some $\mathtt{S}' \preceq \mathtt{T}_1$. The latter alternative implies the desired result $\mathtt{S}' \preceq \mathtt{T}_1 \times \mathtt{T}_2$ by the definition of *BU*.

Subcase: $\mathtt{S} \preceq \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}_2$

Similar.

Case: $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$

Similar to the product case.

Case: $\mathtt{T} = \mu\mathtt{Y}.\mathtt{T}'$

We have $\mathtt{S} \preceq \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T} = \mu\mathtt{Y}.\{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}'$. There are two ways in which $\mathtt{S}$ can be a bottom-up subexpression of this μ-type.

Subcase: $\mathtt{S} = \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}$

Take $\mathtt{S}' = \mathtt{T}$

Subcase: $\mathtt{S} = \{\mathtt{Y} \mapsto \mu\mathtt{Y}.\{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}'\}\mathtt{S}_1$
with $\mathtt{S}_1 \preceq \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}'$

Applying the induction hypothesis gives us two possible alternatives:

- $\mathtt{S}_1 \preceq \mathtt{Q}$. By our conventions on bound variable names, we know that $\mathtt{Y} \notin FV(\mathtt{Q})$, so it must be that $\mathtt{Y} \notin FV(\mathtt{S}_1)$. But then $\mathtt{S} = \{\mathtt{Y} \mapsto \mu\mathtt{Y}.\{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}'\}\mathtt{S}_1 = \mathtt{S}_1$, so $\mathtt{S} \preceq \mathtt{Q}$.
- $\mathtt{S}_1 = \{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{S}_2$ for some $\mathtt{S}_2$ such that $\mathtt{S}_2 \preceq \mathtt{T}'$. In this case, $\mathtt{S} = \{\mathtt{Y} \mapsto \mu\mathtt{Y}.\{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}'\}\mathtt{S}_1 = \{\mathtt{Y} \mapsto \mu\mathtt{Y}.\{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{T}'\}\{\mathtt{X} \mapsto \mathtt{Q}\}\mathtt{S}_2 = \{\mathtt{X} \mapsto \mathtt{Q}\}\{\mathtt{Y} \mapsto \mu\mathtt{Y}.\mathtt{T}'\}\mathtt{S}_2$. Take $\mathtt{S}' = \{\mathtt{Y} \mapsto \mu\mathtt{Y}.\mathtt{S}'\}\mathtt{S}_2$ to obtain the desired result. □

The final piece of the proof establishes that every top-down subexpression of a μ-type can be found among its bottom-up subexpressions.

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

Proof
We want to show that $\mu TD \subseteq \mu BU$. By the principle of induction, this will follow if we can show that μBU is *TD*-closed, that is, $TD(\mu BU) \subseteq \mu BU$. In other words, we

want to show that $(\mathtt{A},\mathtt{B}) \in TD(\mu BU)$ implies $(\mathtt{A},\mathtt{B}) \in \mu BU = BU(\mu BU)$. The latter will be true if every clause of TD that could have generated $(\mathtt{A},\mathtt{B})$ from μBU is matched by a clause of BU that also generates $(\mathtt{A},\mathtt{B})$ from μBU. This is trivially true for all the clauses of TD except the last, since they are exactly the same as the corresponding clauses of BU. In the last clause, $(\mathtt{A},\mathtt{B}) = (\mathtt{S},\mu\mathtt{X}.\mathtt{T}) \in TD(\mu BU)$ and $(\mathtt{S},\{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{T}) \in \mu BU$ or, equivalently, $\mathtt{S} \preceq \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{T}$. By Lemma 10.9, either $\mathtt{S} \preceq \mu\mathtt{X}.\mathtt{T}$, which is $(\mathtt{S},\mu\mathtt{X}.\mathtt{T}) \in \mu BU$, what is needed, or $\mathtt{S} = \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{S}'$ for some $\mathtt{S}'$ with $(\mathtt{S}',\mathtt{T}) \in \mu BU$. The latter implies $(\mathtt{S},\mu\mathtt{X}.\mathtt{T}) \in BU(\mu BU) = \mu BU$, by the last clause of BU. □

Combining the facts established in this section gives us the final result.

Proposition 10.11
For any μ-types $\mathtt{S}$ and $\mathtt{T}$, the set $reachable_{S_m}(\mathtt{S},\mathtt{T})$ is finite.

Proof
For $\mathtt{S}$ and $\mathtt{T}$, let Td be the set of all their top-down subexpressions, and Bu be the set of all their bottom-up subexpressions. According to Proposition 10.5, $reachable_{S_m}(\mathtt{S},\mathtt{T}) \subseteq Td \times Td$. By Proposition 10.10, $Td \times Td \subseteq Bu \times Bu$. By Lemma 10.8, the latter set is finite. Therefore, $reachable_{S_m}(\mathtt{S},\mathtt{T})$ is finite. □

11 Digression: an exponential algorithm

The algorithm *subtype* presented at the beginning of Section 10 (figure 5) can be simplified a bit more by making it return just a boolean value rather than a new set of assumptions (see figure 6). The resulting procedure, $subtype^{ac}$, corresponds to Amadio & Cardelli's (1993) algorithm for checking subtyping. 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 in the $\rightarrow$ 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 square of the total number of subexpressions in the two argument types (as can be seen by inspecting the proofs of Lemma 10.8 and Proposition 10.11), in the case of $subtype^{ac}$ it is exponential.

The exponential behavior of $subtype^{ac}$ can be seen clearly in the following example. Define families of types $\mathtt{S}_n$ and $\mathtt{T}_n$ inductively as follows:

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

Since $\mathtt{S}_n$ and $\mathtt{T}_n$ each contain just one occurrence of $\mathtt{S}_{n-1}$ and $\mathtt{T}_{n-1}$, respectively, their size (after expanding abbreviations) will be linear in n. Checking $\mathtt{S}_n$ <: $\mathtt{T}_n$ generates an exponential derivation, however, as can be seen by the following sequence of

$$
\begin{array}{lcl}
\textit{subtype}^{ac}(A, \mathtt{S}, \mathtt{T}) & = & \text{if } (\mathtt{S}, \mathtt{T}) \in A, \text{ then } \textit{true} \\
& & \text{else let } A_0 = A \cup (\mathtt{S}, \mathtt{T}) \text{ in} \\
& & \quad \text{if } \mathtt{T} = \mathtt{Top}, \text{ then } \textit{true} \\
& & \quad \text{else if } \mathtt{S} = \mathtt{S}_1 \times \mathtt{S}_2 \text{ and } \mathtt{T} = \mathtt{T}_1 \times \mathtt{T}_2, \text{ then} \\
& & \qquad \textit{subtype}^{ac}(A_0, \mathtt{S}_1, \mathtt{T}_1) \text{ and} \\
& & \qquad \textit{subtype}^{ac}(A_0, \mathtt{S}_2, \mathtt{T}_2) \\
& & \quad \text{else if } \mathtt{S} = \mathtt{S}_1 \to \mathtt{S}_2 \text{ and } \mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2, \text{ then} \\
& & \qquad \textit{subtype}^{ac}(A_0, \mathtt{T}_1, \mathtt{S}_1) \text{ and} \\
& & \qquad \textit{subtype}^{ac}(A_0, \mathtt{S}_2, \mathtt{T}_2) \\
& & \quad \text{else if } \mathtt{S} = \mu \mathtt{X}.\mathtt{S}_1, \text{ then} \\
& & \qquad \textit{subtype}^{ac}(A_0, \{\mathtt{X} \mapsto \mu \mathtt{X}.\mathtt{S}_1\}\mathtt{S}_1, \mathtt{T}) \\
& & \quad \text{else if } \mathtt{T} = \mu \mathtt{X}.\mathtt{T}_1, \text{ then} \\
& & \qquad \textit{subtype}^{ac}(A_0, \mathtt{S}, \{\mathtt{X} \mapsto \mu \mathtt{X}.\mathtt{T}_1\}\mathtt{T}_1) \\
& & \quad \text{else } \textit{false}.
\end{array}
$$

Fig. 6. Amadio and Cardelli's subtyping algorithm.

recursive calls:

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

where

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

Notice that the initial call $\textit{subtype}^{ac}(\emptyset, \mathtt{S}_n, \mathtt{T}_n)$ 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 is thus proportional to 2^n.

12 Notes

Background on coinduction can be found in Barwise and Moss's *Vicious Circles* (1996), Gordon's tutorial on coinduction and functional programming (1995), and Milner and Tofte's expository article on coinduction in programming language semantics (1991). For basic information on monotone functions and fixed points see Aczel (1977) and Davey & Priestley (1990).

The use of coinductive proof methods in computer science dates from the 1970s, for example in the work of Milner (1980) and Park (1981) on concurrency; also see Arbib and Manes's categorical discussion of duality in automata theory (1975). But the use of induction in its dual 'co-' form was familiar to mathematicians considerably earlier and is developed explicitly in, for example, universal algebra and category theory. Aczel's seminal book (1988) on non-well-founded sets includes a brief historical survey.

Recursive types in computer science go back at least to Morris (1968). Basic syntactic and semantic properties (without subtyping) are collected in Cardone & Coppo (1991). Properties of infinite and regular trees are surveyed by Courcelle (1983). Basic syntactic and semantic properties of recursive types without subtyping were established in early papers by Huet (1976) and MacQueen, Plotkin & Sethi (1986). The relation between iso- and equi-recursive systems was explored by Abadi & Fiore (1996).

Amadio & Cardelli (1993) 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 subtype 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 model construction based on partial equivalence relations. 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 & Henglein (1997) laid bare the underlying coinductive nature of Amadio and Cardelli's system, giving a new inductive axiomatization of the subtype 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 article closely follows the latter proof. Brandt and Henglein establish that the complexity of their algorithm is $O(n^2)$.

Kozen, Palsberg & Schwartzbach (1993) obtain an elegant quadratic subtyping algorithm by observing that a regular recursive type corresponds to an automaton with labeled states. They define a 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 emptiness 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.

Hosoya, Vouillon & Pierce (2000) 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 (1999) address type *reconstruction* for languages with subtyping and recursive types. As we have done in this article, 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.

The two alternative formulations of recursive types have been known since early times, but the pleasantly mnemonic terms *iso-recursive* and *equi-recursive* are a relatively new coinage by Crary, Harper & Puri (1999).

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 and JFP reviewers. This work was supported by the University of Pennsylvania and by NSF Career grant CCR-9701826, *Principled Foundations for Programming with Objects*.

Appendix: Solutions to exercises

Solution to Exercise 2.7

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

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

Solution to Exercise 2.9

To prove the principle of ordinary induction on natural numbers, we proceed as follows. Define the generating function $F \in \mathscr{P}(\mathbb{N}) \to \mathscr{P}(\mathbb{N})$ by

$$F(X) \quad = \quad \{0\} \cup \{i+1 \mid i \in X\}.$$

Now, suppose we have a predicate (i.e. a set of numbers) P such that $P(0)$ and such that $P(i)$ implies $P(i+1)$. Then, from the definition of F, it is easy to see that $F(P) \subseteq P$, i.e. P is F-closed. By the induction principle, $\mu F \subseteq P$. But μF is the whole set of natural numbers (indeed, this can be taken as the *definition* of the set of natural numbers), so $P(n)$ holds for all $n \in \mathbb{N}$.

For lexicographic induction, define $F \in \mathscr{P}(\mathbb{N} \times \mathbb{N}) \to \mathscr{P}(\mathbb{N} \times \mathbb{N})$ to be

$$F(X) \quad = \quad \{(m,n) \mid \forall (m',n') < (m,n),\ (m',n') \in X\}.$$

Now, suppose we have a predicate (i.e. a set of pairs of numbers) P such that, whenever $P(m',n')$ for all $(m',n') < (m,n)$, we also have $P(m,n)$. As before, from the definition of F, it is easy to see that $F(P) \subseteq P$, i.e. P is F-closed. By the induction principle, $\mu F \subseteq P$. To finish, we must check that μF is indeed the set of all pairs of numbers (this is the only subtle bit of the argument). This can be argued in two steps. First, we remark that $\mathbb{N} \times \mathbb{N}$ is F-closed (this is immediate from the definition of F). Secondly, we show that no proper subset of $\mathbb{N} \times \mathbb{N}$ is F-closed, i.e. $\mathbb{N} \times \mathbb{N}$ is the smallest F-closed set. To see this, suppose there were a smaller F-closed set Y, and let (m, n) be the smallest pair that does *not* belong to Y; by the definition of F, we see that $F(Y) \nsubseteq Y$, i.e. Y is not closed – a contradiction.

Solution to Exercise 3.2

Define a *tree* to be a partial function $T \in \{1,2\}^* \rightharpoonup \{\to, \times, \mathtt{Top}\}$ satisfying the following constraints:

- $T(\bullet)$ is defined;
- if $T(\pi \cdot \sigma)$ is defined then $T(\pi)$ is defined.

Note that occurrences of the symbols $\to,\times,\mathtt{Top}$ in the nodes of a tree are completely unconstrained, e.g. a node with Top can have non-trivial children, etc. As in Section 3, we overload the symbols $\to$, $\times$ and Top to be also operators on trees.

The set of all trees is taken as the universe $\mathcal{U}$. The generating function F is based on the familiar grammar for types:

$$\begin{array}{rcl} F(X) & = & \{\mathtt{Top}\} \\ & \cup & \{T_1 \times T_2 \mid T_1, T_2 \in X\} \\ & \cup & \{T_1 \to T_2 \mid T_1, T_2 \in X\}. \end{array}$$

It can be seen from the definitions of $\mathcal{T}$ and $\mathcal{U}$ that $\mathcal{T} \subseteq \mathcal{U}$, so it makes sense to compare the sets in the equations of interest, $\mathcal{T} = \nu F$ and $\mathcal{T}_f = \mu F$. It remains to check that the equations are true.

$\mathcal{T} \subseteq \nu F$ follows by the principle of coinduction from the fact that $\mathcal{T}$ is F-consistent. To obtain $\nu F \subseteq \mathcal{T}$, we need to check, for any $T \in \nu F$, the two last conditions from Definition 3.1. This can be done by induction on the length of π.

$\mu F \subseteq \mathcal{T}_f$ follows by the principle of induction from the fact that $\mathcal{T}_f$ is F-closed. To obtain $\mathcal{T}_f \subseteq \mu F$, we argue, by induction on the size of $\mathtt{T}$, that $\mathtt{T} \in \mathcal{T}_f$ implies $\mathtt{T} \in \mu F$. (The size of $\mathtt{T} \in \mathcal{T}_f$ can be defined as the length of the longest sequence $\pi \in \{1,2\}^*$ such that $\mathtt{T}(\pi)$ is defined.)

Solution to Exercise 4.3

The pair (Top, Top$\times$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 Exercise 4.4

For an example of a pair of tree types that are related by νS but not by μS, we can take the pair $(\mathtt{T},\mathtt{T})$ for any infinite type $\mathtt{T}$. Consider the set pairs $R = \{(\mathtt{T}(\pi), \mathtt{T}(\pi)) \mid \pi \in \{1,2\}^*\}$. An examination of the definition of S easily gives $R \subseteq S(R)$, and applying the principle of coinduction gives $R \subseteq \nu S$. Then $(\mathtt{T},\mathtt{T}) \in \nu S$ because $(\mathtt{T},\mathtt{T}) \in R$. On the other hand, $(\mathtt{T},\mathtt{T}) \notin \mu S$ because μS relates only finite types – this can be established by taking R' to be the set of all pairs of finite types and obtaining $\mu S \subseteq R'$ by the principle of induction.

There are no pairs $(\mathtt{S},\mathtt{T})$ of finite types that are related by νS_f, but not by μS_f, because the two fixed points coincide. This follows from the fact that, for any $\mathtt{S},\mathtt{T} \in \mathcal{T}_f$, $(\mathtt{S},\mathtt{T}) \in \nu S_f$ implies $(\mathtt{S},\mathtt{T}) \in \mu S_f$. (Since $\mathtt{T}$ is a finite tree, the latter statement follows, in turn, be obtained by induction on $\mathtt{T}$. One needs to consider the cases of $\mathtt{T}$ being $\mathtt{Top}$, $\mathtt{T}_1\times\mathtt{T}_2$, $\mathtt{T}_1\to\mathtt{T}_2$, inspect the definition of S_f, and use the equalities $S_f(\nu S_f) = \nu S_f$ and $S_f(\mu S_f) = \mu S_f$.)

Solution to Exercise 4.8

Begin by defining the identity relation on tree types: $I = \{(\mathtt{T},\mathtt{T}) \mid \mathtt{T} \in \mathscr{T}\}$. If we can show that I is S-consistent, then the coinduction principle will tell us that $I \subseteq \nu S$, that is, νS is reflexive. To show the S-consistency of I, consider an element $(\mathtt{T},\mathtt{T}) \in I$, and proceed by cases on the form of $\mathtt{T}$. First, suppose $\mathtt{T} = \mathtt{Top}$. Then $(\mathtt{T},\mathtt{T}) = (\mathtt{Top},\mathtt{Top})$, which is in $S(I)$ by definition. Suppose, next, that $\mathtt{T} = \mathtt{T}_1\times\mathtt{T}_2$. Then, since $(\mathtt{T}_1,\mathtt{T}_1),(\mathtt{T}_2,\mathtt{T}_2) \in I$, the definition of S gives $(\mathtt{T}_1\times\mathtt{T}_2,\mathtt{T}_1\times\mathtt{T}_2) \in S(I)$. Similarly for $\mathtt{T} = \mathtt{T}_1\to\mathtt{T}_2$.

Solution to Exercise 5.2

By the coinduction principle, it is enough to show that $\mathscr{U} \times \mathscr{U}$ is F^{TR}-consistent, i.e. $\mathscr{U} \times \mathscr{U} \subseteq F^{TR}(\mathscr{U} \times \mathscr{U})$. Suppose $(x,y) \in \mathscr{U} \times \mathscr{U}$. Pick any $z \in \mathscr{U}$. Then $(x,z),(z,y) \in \mathscr{U} \times \mathscr{U}$, and so, by the definition of F^{TR}, also $(x,y) \in F^{TR}(\mathscr{U} \times \mathscr{U})$.

Solution to Exercise 6.2

To check invertability, we just inspect the definitions of S_f and S and make sure that each set $G_{(\mathtt{S},\mathtt{T})}$ contains at most one element.

In the definitions of S_f and S each clause explicitly specifies the form of a supportable element and the contents of its support set, so writing down $support_{S_f}$ and $support_S$ is easy. (Compare with the support function for S_m in Definition 9.4.)

Solution to Exercise 6.4

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

Solution to Exercise 6.6

No, an $x \in \nu F \setminus \mu F$ does not have to lead to a cycle in the support graph: it can also lead to an infinite chain. For example, consider $F \in \mathscr{P}(\mathbb{N}) \to \mathscr{P}(\mathbb{N})$ defined by $F(X) = \{0\} \cup \{n \mid n+1 \in X\}$. Then $\mu F = \{0\}$ and $\nu F = \mathbb{N}$. Also, for any $n \in \nu F \setminus \mu F$, that is for any $n > 0$, $support(n) = \{n+1\}$, generating an infinite chain.

Solution to Exercise 6.13

First, consider partial correctness. The proof for each part proceeds by induction on the recursive structure of a run of the algorithm:

1. From the definition of *lfp*, it is easy to see that there are two cases where $lfp(X)$ can return *true*. If $lfp(X) = true$ because $X = \emptyset$, we have $X \subseteq \mu F$ trivially. On the other hand, if $lfp(X) = true$ because $lfp(support(X)) = true$, then, by the induction hypothesis, $support(X) \subseteq \mu F$, from which Lemma 6.8 yields $X \subseteq \mu F$.
2. If $lfp(X) = false$ because $support(X)\uparrow$, then $X \nsubseteq \mu F$ by Lemma 6.8. Otherwise, $lfp(X) = false$ because $lfp(support(X)) = false$, and, by the induction hypothesis, $support(X) \nsubseteq \mu F$. By Lemma 6.8, $X \nsubseteq \mu F$.

Next, we want to characterize the generating functions F for which *lfp* is guaranteed to terminate on all finite inputs. For this, some new terminology is helpful.

Given a finite-state generating function $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$, the partial function $height_F \in \mathcal{U} \rightharpoonup \mathbb{N}$ (or just *height*) is the least partial function satisfying the following condition:[6]

$$height(x) = \begin{cases} 0 & \text{if } support(x) = \emptyset \\ 0 & \text{if } support(x) \uparrow \\ 1 + max\{height(y) \mid y \in support(x)\} & \text{if } support(x) \neq \emptyset \end{cases}$$

(Note that $height(x)$ is undefined if x either participates in a reachability cycle itself or depends on an element from a cycle.) A generating function F is said to be *finite height* if $height_F$ is a total function. It is easy to check that, if $y \in support(x)$ and both $height(x)$ and $height(y)$ are defined, then $height(y) < height(x)$.

Now, if F is finite state and finite height, then $lfp(X)$ terminates for any finite input set $X \subseteq \mathcal{U}$. To see this, observe that, since F is finite state, for every recursive call $lfp(Y)$ descended from the original call $lfp(X)$, the set Y is finite. Since F is finite height, $h(Y) = max\{height(y) \mid y \in Y\}$ is well defined. Since $h(Y)$ decreases with each recursive call and is always non-negative, it serves as a termination measure for *lfp*.

Solution to Exercise 9.5

The definition of S_d is the same as that of S_m, except that the last clause does not contain the conditions $\mathtt{T} \neq \mu\mathtt{X}.\mathtt{T}_1$ and $\mathtt{T} \neq \mathtt{Top}$. To see that S_d is not invertible, observe that the set $G_{(\mu\mathtt{X}.\mathtt{Top},\mu\mathtt{Y}.\mathtt{Top})}$ contains two generating sets, $\{(\mathtt{Top}, \mu\mathtt{Y}.\mathtt{Top})\}$ and $\{(\mu\mathtt{X}.\mathtt{Top}, \mathtt{Top})\}$ (compare the contents of this set for the function S_m).

Because all the clauses of S_d and S_m are the same, except the last, and the last clause of S_m is a restriction of the last clause of S_d, the inclusion $\nu S_m \subseteq \nu S_d$ is obvious. The other inclusion, $\nu S_d \subseteq \nu S_m$, can be proved using the principle of coinduction together with the following lemma, which establishes that νS_d is S_m-consistent.

Lemma

For any two μ-types $\mathtt{S}$, $\mathtt{T}$, if $(\mathtt{S},\mathtt{T}) \in \nu S_d$, then $(\mathtt{S},\mathtt{T}) \in S_m(\nu S_d)$.

The lemma is proved by lexicographic induction on (n, k), where $k = \mu\text{-}height(\mathtt{S})$ and $n = \mu\text{-}height(\mathtt{T})$. This induction verifies the informal idea that any derivation of $(\mathtt{S},\mathtt{T}) \in \nu S_d$ can be transformed into another derivation of the same fact, that also happens to be a derivation of $(\mathtt{S},\mathtt{T}) \in \nu S_m$. The restrictions in the rule of left μ-folding dictate that the transformed derivation has the property that every sequence of applications of μ-folding rules starts with a sequence of left μ-foldings, which are then followed by a sequence of right μ-foldings.

Solution to Exercise 10.2

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

$$\frac{\mathtt{S} \sqsubseteq \mathtt{T}_1}{\mathtt{S} \sqsubseteq \mathtt{T}_1 \to \mathtt{T}_2} \qquad \frac{\mathtt{S} \sqsubseteq \mathtt{T}_2}{\mathtt{S} \sqsubseteq \mathtt{T}_1 \to \mathtt{T}_2} \qquad \frac{\mathtt{S} \sqsubseteq \{\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}\}\mathtt{T}}{\mathtt{S} \sqsubseteq \mu\mathtt{X}.\mathtt{T}}$$

[6] Observe that this way of phrasing the definition of *height* can easily be rephrased as the least fixed point of a monotone function on relations representing partial functions.

(Note, as a point of interest, that the generating function *TD* differs from the generating functions we have considered throughout this article: it is not invertible. For example, $\mathtt{B} \sqsubseteq \mathtt{A}\times\mathtt{B}\rightarrow\mathtt{B}\times\mathtt{C}$ is supported by the two sets $\{\mathtt{B} \sqsubseteq \mathtt{A}\times\mathtt{B}\}$ and $\{\mathtt{B} \sqsubseteq \mathtt{B}\times\mathtt{C}\}$, neither of which is a subset of the other.)

Solution to Exercise 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 types starting with a μ binder:

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

References

Abadi, M. and Fiore, M. P. (1996) Syntactic considerations on recursive types. *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science, LICS '96*, pp. 242–252. IEEE Press.

Aczel, P. (1977) An introduction to inductive definitions. In: Barwise, J. (ed.), *Handbook of Mathematical Logic*, pp. 739–782. Studies in Logic and the Foundations of Mathematics, no. 90. North Holland.

Aczel, P. (1988) *Non-well-founded sets.* Stanford Center for the Study of Language and Information. CSLI Lecture Notes number 14.

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

Arbib, M. and Manes, E. (1975) *Arrows, Structures, and Functors: The categorical imperative.* Academic Press.

Barwise, J. and Moss, L. (1996) *Vicious Circles: On the mathematics of non-wellfounded phenomena.* Cambridge University Press.

Brandt, M. and Henglein, F. (1997) Coinductive axiomatization of recursive type equality and subtyping. In: Hindley, R. (ed.), *Proc. 3rd Int. Conf. on Typed Lambda Calculi and Applications (TLCA): Lecture Notes in Computer Science 1210*, pp. 63–81. Nancy, France. Springer-Verlag.

Cardone, F. and Coppo, M. (1991) Type inference with recursive types: Syntax and semantics. *Information & Computation*, **92**(1), 48–80.

Courcelle, B. (1983) Fundamental properties of infinite trees. *Theor. Comput. Sci.*, **25**, 95–169.

Crary, K., Harper, R. and Puri, S. (1999) What is a recursive module? *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 50–63.

Davey, B. A. and Priestley, H. A. (1990) *Introduction to Lattices and Order.* Cambridge University Press.

Ghelli, G. (1993) Recursive types are not conservative over $F_{\leqslant}$. In: Bezen, M. and Groote, J.F. (eds.), *Proceedings International Conference on Typed Lambda Calculi and Applications (TLCA): Lecture Notes in Computer Science 664*, pp. 146–162. Utrecht, The Netherlands. Springer-Verlag.

Gordon, A. (1995) A tutorial on co-induction and functional programming. *Functional Programming*, pp. 78–95. Glasgow, Scotland. Springer-Verlag.

Gunter, C. A. (1992) *Semantics of Programming Languages: Structures and techniques.* MIT Press.

Hosoya, H., Vouillon, J. and Pierce, B. C. (2000) Regular expression types for XML. *Proceedings International Conference on Functional Programming (ICFP)*.

Huet, G. (1976) *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France).

Jim, T. and Palsberg, J. (1999) *Type inference in systems of recursive types with subtyping*. Manuscript.

Kozen, D., Palsberg, J. and Schwartzbach, M. I. (1993) Efficient recursive subtyping. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 419–428.

MacQueen, D., Plotkin, G. and Sethi, R. (1986) An ideal model for recursive polymorphic types. *Information & Control*, **71**, 95–130.

Milner, R. (1980) *A Calculus of Communicating Systems: Lecture Notes in Computer Science 92*. Springer-Verlag.

Milner, R. and Tofte, M. (1991) Co-induction in relational semantics. *Theor. Comput. Sci.*, **87**, 209–220.

Mitchell, J. C. (1996) *Foundations for Programming Languages*. MIT Press.

Morris, J. H. (1968) *Lambda calculus models of programming languages*. Technical Report MIT-LCS//MIT/LCS/TR-57, MIT Laboratory for Computer Science.

Park, D. (1981) Concurrency and automata on infinite sequences. In: Deussen, P. (ed.), *Proceedings 5th GI-Conference on Theoretical Computer Science: Lecture Notes in Computer Science 104*, pp. 167–183. Springer-Verlag.

Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Mathematics*, **5**, 285–309.

Winskel, G. (1993) *The Formal Semantics of Programming Languages: An introduction*. MIT Press.