Recursive Types

Marco Kuhlmann*

Infinite but regular data structures are indispensable tools in all modern programming languages. *Recursive types* are finite representations of such infinite structures, yielding a natural extension of the simply typed Lambda-Calculus. This paper presents the formal tools needed to reason about recursive types, discusses a generic type-checking algorithm, and demonstrates how this algorithm can be generalised to also handle subtyping.

1 Introduction

Recursion is ubiquitious in all modern programming languages: Lists and trees are just two large classes of examples for data structures whose values may be arbitrarily large, but follow a simple, regular structure. In a typed language, the structure of a value is captured by its type, and lists and trees are instances of *recursive types.*

The running example for a recursive data structure that will be used throughout this paper is the type of lists of natural numbers, NatList. Each such list is either empty (**nil**) or the concatenation (**cons**) of a natural number (type Nat) and a possibly empty rest list. Using an ML-like type constructor notation, the structure of the NatList type may be written as

$$\mathsf{NatList} = \mathbf{nil} : \mathsf{Unit} \mid \mathbf{cons} : \mathsf{Nat} \times \mathsf{NatList}.$$

Note that NatList occurs both to the left and to the right of the equals sign—the definition is recursive. The Unit type used in the **nil** case is a dummy type with just one value, also written Unit.

* This paper is the extended abstract of a talk given in a seminar on Types and Programming Languages at Saarland University on March 5, 2003. All material, including references, was taken from Benjamin Pierce's book *Types and Programming Languages,* MIT Press 2002.

The author of this summary may be contacted via email at `kuhlmann@ps.uni-sb.de`.

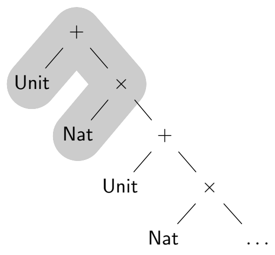

Figure 1: Representation of the NatList type $\mu T.\ \mathsf{Unit} + \mathsf{Nat} \times T$ as an infinite tree. The shaded area corresponds to a one-level unfolding of the type term.

A more abstract view on the NatList type will omit concrete names for the alternatives of the sum type and make the recursion in the **cons** alternative explicit. The NatList type may then be written as

$$\mathsf{NatList} = \mu T.\ \mathsf{Unit} + \mathsf{Nat} \times T.$$

This notation is the *μ-notation* that will be formally developed in section 3.1: μ is an abstraction operator for types, just as λ is an abstraction operator for values.

The μ-notation can be taken as a compact representation of infinite regular trees. For example, the NatList type corresponds to the tree depicted in figure 1. Section 3 will elaborate this view.

Expressiveness of recursive types

Recursive types are an extremely expressive extension of the simply-typed Lambda-Calculus $\lambda_{\to}$. To see this, recall that in the untyped Lambda-Calculus, recursive functions can be defined with the help of the fixed point combinator

$$\mathsf{fix} = \lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx))\,.$$

This term cannot be typed in $\lambda_{\to}$, as in order to do so, the sub-term x would need to have an arrow type whose domain is the type of x itself—something that does not exist in $\lambda_{\to}$. In a type system with recursive types, however, the required property is easily satisfied by the type $\mu S.\ S \to T$, and for every type T, a fixed point combinator can be typed as follows:

$$\begin{aligned} \mathsf{fix}_T &= \lambda f : T \to T.\ (\lambda x : (\mu S.\ S \to T).\ f(xx))\ (\lambda x : (\mu S.\ S \to T).\ f(xx)) \\ \mathsf{fix}_T &: (T \to T) \to T \end{aligned}$$

This feature was first noticed by James Morris in 1968 (Morris 1968), one of the earliest authors to deal with recursive types in computer science.

A corollary of the presence of a well-typed fixed point combinator for every type T is, that systems with recursive types do not have the strong normalisation property according to which every well-typed term should have a normal form. More specifically, they allow for infinitely many well-typed diverging functions: For every type T, a function $\mathsf{diverge}_T$ can be defined and typed that diverges when applied to Unit:

$$\begin{aligned} \mathsf{diverge}_T &= \lambda\,\_ : \mathsf{Unit}.\ \mathsf{fix}_T(\lambda\, x : T.\ x) \\ \mathsf{diverge}_T &: \mathsf{Unit} \to T \end{aligned}$$

The existance of such functions also renders recursive type systems useless as logics in the Curry–Howard sense: The fact that every type T is inhabited translates to the statement that every proposition is provable.

The structure of this paper

The next section will present two ways to formalise recursive types. Section 3 introduces two formal tools needed to reason about recursive types, the μ-notation and the concept of co-induction. In section 4, a generic algorithm for membership tests for recursive types is developed. Finally, in section 5, this algorithm is instantiated with a typability relation for subtyping on recursive types.

2 Formalising recursive types

The literature on type systems offers two principal views on recursive types, which may be distinguished by their answer to a simple question: What is the relation between the recursive type $\mu X.\ T$ and its one-step unfolding, $T[X/\mu X.\ T]$? In the $\mathsf{NatList}$ example, this means asking for the relation between

$$\mu T.\ \mathsf{Unit} + \mathsf{Nat} \times T \qquad \text{and} \qquad \mathsf{Unit} + \mathsf{Nat} \times (\mu T.\ \mathsf{Unit} + \mathsf{Nat} \times T).$$

In the *equi-recursive approach,* a recursive type and its unfoldings are considered to be interchangeable in all contexts—both denote the *same* infinite type. In the *iso-recursive approach,* recursive types and their unfoldings are taken to be different, though isomorphic types. Either of the two approaches has its benefits and drawbacks.

The major benefit of the *equi-recursive approach* is that it is conceptually clean and can be integrated into an existing type system in a straightforward fashion—the only

$$\frac{U = \mu X.\, T_1 \qquad \Gamma \vdash t_1 : U}{\Gamma \vdash \mathsf{unfold}_U(t_1) : T_1[X/U]}\ \textsf{TUnfold} \qquad \frac{U = \mu X.\, T_1 \qquad \Gamma \vdash t_1 : T_1[X/U]}{\Gamma \vdash \mathsf{fold}_U(t_1) : U}\ \textsf{TFold}$$

$$\mathsf{unfold}_S(\mathsf{fold}_T(v_1)) \to v_1$$

Figure 2: Typing rules for folding and unfolding

thing that needs to be changed is to allow type expressions to be infinite. The implementation of type-checkers for equi-recursive types is *not* straightforward, however; one basic idea for such an implementation will be outlined in section 4. Another unpleasant thing about the equi-recursive approach is that it does not smoothly integrate with other advanced typing features, such as bounded quantification.

Implementability is the strong side of the *iso-recursive approach:* While on the formal level, the need for an explicit isomorphism between a recursive type and its unfoldings makes a type system more complicated, this isomorphism can often be hidden away completely behind other language constructs in the implementation. The rest of this section briefly discusses such “piggybacking” techniques.

Although the distinction between the equi-recursive and the iso-recursive approach towards has been present in the literature on recursive types from the beginning, the names for the two approaches were coined in a paper by Karl Crary, Robert Harper, and Sidd Puri from as recent as 1999 (Crary et. al. 1999).

Folding, unfolding, and piggybacking

In the iso-recursive approach, the isomorphism between a recursive type and its unfoldings is established by two functions unfold and fold:

$$\begin{aligned} \mathsf{unfold}_{\mu X.\,T} &: \mu X.\, T \to T[X/\mu X.\, T] \\ \mathsf{fold}_{\mu X.\,T} &: T[X/\mu X.\, T] \to \mu X.\, T \end{aligned}$$

These functions, together with a new evaluation rule that annihilates a fold when it meets a corresponding unfold, can then be used in the type system to equalise two type annotations whenever their recursive types are isomorphic (figure 2).

For an iso-recursively defined NatList type, the two functions would have the types

$$\begin{aligned} \mathsf{unfold}_{\mathsf{NatList}} &: \mathsf{NatList} \to \mathsf{NatListBody} \\ \mathsf{fold}_{\mathsf{NatList}} &: \mathsf{NatListBody} \to \mathsf{NatList}, \end{aligned}$$

where $\mathsf{NatListBody} := \mathsf{Unit} + \mathsf{Nat} \times \mathsf{NatList}$. Given these functions, the two type constructors for the $\mathsf{NatList}$ type, **nil** and **cons** could be written as

$$\begin{aligned} \mathbf{nil} &= \mathsf{fold}_{\mathsf{NatList}}(\langle 1, \mathsf{Unit}\rangle) \\ \mathbf{cons} &= \lambda\, n : \mathsf{Nat}.\ \lambda\, l : \mathsf{NatList}.\ \mathsf{fold}_{\mathsf{NatList}}(\langle 2, (n, l)\rangle). \end{aligned}$$

Folding can thus be hidden away behind the type constructors of an SML datatype: Each use of a constructor implicitly includes a fold. Similarly, the unfolding of an iso-recursive type can be "piggybacked" onto pattern matching: Each type constructor implicitly forces an unfold operation. Variations of this technique are used in many implementations of SML.

3 Reasoning about recursive types

This section will introduce the formal framework needed to reason about recursive types. The basic idea is that recursive types can be represented as infinite but regular labelled trees over a signature containing type variables and type constructors like $\times$ (for product types), $+$ (for sum types) and $\to$ (for arrow types).

Arbitrary labelled trees can be defined like this:

Definition 1 (Labelled tree) Let $(\Sigma, \mathsf{ar} : \Sigma \to \mathbb{N})$ be a signature. A *labelled tree* is a partial function $\mathsf{T} \in \mathbb{N}^* \to \Sigma$ where (a) $\mathsf{T}(\varepsilon)$ is defined, (b) $\mathsf{T}(\pi)$ is defined whenever $\mathsf{T}(\pi\sigma)$ is defined, and (c) if $(\mathsf{ar} \circ \mathsf{T})(\pi) = k$, then $\mathsf{T}(\pi i)$ is defined for $1 \leq i \leq k$.

The domain of T is called the set of the *nodes* of the tree; the special node ε required by condition (a) is the *root node.* The codomain of T is a set of *labels.* Condition (b) induces an *ancestor relation* on the tree: A node π_1 is an ancestor of a node π_2 if and only if π_1, taken as a word in $\mathbb{N}^*$, is a prefix of π_2.

In the context of a type system, we are only interested in trees that can be finitely represented. These trees are called *regular trees:*

Definition 2 (Subtree; regular trees) Let T be a tree. A tree S is a *subtree* of T if $S = \lambda\, \sigma.\ T(\pi\sigma)$ for some π. T is called *regular,* if the set of its subtrees is finite.

In the following, the μ-notation will be developed as a compact representation of regular trees.

$$\begin{array}{lcl}
\mathsf{treeof}(T_1 \times T_2)(\varepsilon) & = & \times \\
\mathsf{treeof}(T_1 \times T_2)(i\pi) & = & \mathsf{treeof}(T_i)(\pi) \\
\mathsf{treeof}(T_1 + T_2)(\varepsilon) & = & + \\
\mathsf{treeof}(T_1 + T_2)(i\pi) & = & \mathsf{treeof}(T_i)(\pi) \\
\mathsf{treeof}(T_1 \to T_2)(\varepsilon) & = & \to \\
\mathsf{treeof}(T_1 \to T_2)(i\pi) & = & \mathsf{treeof}(T_i)(\pi) \\
\mathsf{treeof}(\mu X.\ T)(\pi) & = & \mathsf{treeof}(T[X/\mu X.\ T])(\pi)
\end{array}$$

Figure 3: Mapping μ-types to trees

3.1 Regular trees and μ-types

As already mentioned in the introduction, regular trees can be described by μ-terms (or μ-types—the two names are used synonymously in the following paragraphs): Each distinct subtree is represented by a different subterm of a μ-term, and (regular) infinity is introduced by the the μ-binder.

Definition 3 (Raw μ-terms) Let $\mathcal{V}$ be a set of type variables. The set of *raw μ-terms over $\mathcal{V}$* then is defined by the following abstract grammar:

$$T, T_1, T_2, T' \ ::= \ X \in \mathcal{V} \mid T_1 \times T_2 \mid T_1 \to T_2 \mid \mu X.\ T'.$$

Raw μ-terms are not yet exactly what we would like to have: For example, a term like $\mu X.\ X$ cannot sensibly be interpreted as a regular tree. Therefore, we define μ-types to be raw μ-terms with the additional property of being *contractive:*

Definition 4 (Contractive μ-terms) Let T be a raw μ-term. T is called *contractive,* if it does not have the form $\mu X.\ \mu X_1 \ldots \mu X_n.\ X$.

Each regular tree uniquely corresponds to a (contractive) μ-term. (The proof of this proposition is omitted due to reasons of space.) In particular, there is a function treeof that maps μ-terms to trees (figure 3). The rest of this section will present co-induction as a proof technique for the reasoning about μ-types.

3.2 Co-inductive definitions

The central proof technique used in the reasoning about the simply-typed Lambda-Calculus is induction. Inductively defined sets can be infinitely large, but contain only finite objects. For sets of *infinite* objects, such as a μ-type as a certain set of infinite regular trees, induction as a proof technique does not suffice.

Every inductively defined set X can be identified with a pair of a universe of values, $\mathcal{U}$, and a generating function, F, that eventually partitions $\mathcal{U}$ into the elements of X and the non-elements. The set X then is the least fixed point of F, μF:

$$\mathcal{U} = X \uplus \overline{X} = \mu F \uplus \overline{\mu F}.$$

Dually, a co-inductively defined set is the *greatest* fixed point of the generating function F, νF. For example, if F is a generating function for the NatList type, then μF will be the (infitely large) set of finite NatLists, while νF will contain all objects of the NatList type, even the infinite ones.

Theorem 1, which was first published by Alfred Tarski in 1955 (Tarski 1955), characterises the least and the greatest fixed point of a generating function F on a universe $\mathcal{U}$ in terms of set operations.

Definition 5 Let X be a subset of $\mathcal{U}$.

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

Theorem 1 (Knaster-Tarski) Let $F \in \mathfrak{P}(U) \to \mathfrak{P}(U)$ be monotone.

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: Due to lack of space, I only give the proof of the second part of the theorem.

Let C be the collection of all F-consistent sets, and let P be the union of all these sets. Take any $X \in C$. As X is F-consistent, $X \subseteq F(X)$. From the definition of P, we have $X \subseteq P$; together with the monotonicity of F, this yields $F(X) \subseteq F(P)$. Thus, by transitivity, $X \subseteq F(P)$. Because X was arbitrary,

$$P = \bigcup_{X \in C} X \ \subseteq \ F(P).$$

That is, P is an F-consistent set. As F is a monotonic function, $F(P) \subseteq F(F(P))$, so $F(P)$ is F-consistent as well, thus $F(P) \in C$. Remember that for any $X \in C$, $X \subseteq P$, so in particular $F(P) \subseteq P$—that is, P is F-closed. Because of $P \subseteq F(P)$ (P is F-consistent) and $F(P) \subseteq P$ (P is F-closed), P is a fixed point of F. It is also the largest fixed point, as every other fixed point $P' \supseteq P$ would need to be F-consistent, whereas by its definition, P already is the largest F-consistent set. □

The Knaster-Tarski theorem justifies induction and co-induction as two proof principles for sets defineable by generating functions:

- *Induction:* To show that $\mu F \subseteq P$, show that P is F-closed.

  EXAMPLE: Let P be any property on (that is, subset of) the natural numbers, which are taken to be defined by the generating function

  $$F(N) = \{0\} \cup \{\, \mathsf{succ}(n) \mid n \in N \,\}.$$

  In order to show that all $n \in N$ satisfy the property P ($N \subseteq P$), one has to show that P is F-closed—or, more specifically, that $\{0\} \subseteq P$ and that

  $$\{\, \mathsf{succ}(p) \mid p \in P \,\} \subseteq P.$$

- *Co-induction:* To show that $P \subseteq \nu F$, show that P is F-consistent.

  (TRIVIAL) EXAMPLE: Let $\rightsquigarrow$ be the reduction relation on functional programs, and let the set of diverging programs be defined by the generating function

  $$F(\uparrow) = \{\, a \mid \exists b\colon\ (a \rightsquigarrow b \wedge b \in \uparrow) \,\}.$$

  Consider an expression Ω that reduces to itself ($\Omega \rightsquigarrow \Omega$), and let $P = \{\Omega\}$. P is F-consistent, as $\{\Omega\} = P \subseteq F(P)$. Therefore, $P \subseteq \uparrow$.

The concept of co-induction originates from universal algebra and category theory, but was not employed in computer science until the 1970s. Michael Brandt and Fritz Henglein in a paper published in 1998 were the first to give a co-inductive axiomatisation of the equality of recursive types (Brandt and Henglein 1998).

4 Membership tests for equi-recursive types

The last section has argued that equi-recursive types can be formalised as the greatest fixed point of some generating function F. The present section will outline an effective algorithm that checks whether or not a given type belongs to a certain recursive type—or, to name the more abstract problem, whether or not a given object belongs to the greatest fixed point of F.

The exposition of the generic membership algorithm is based on chapter 21 of Benjamin Pierce's book on *Types and Programming Languages,* which in turns draws from work by Martín Abadi, Luca Cardelli, Michael Brandt, and Fritz Henglein, published during the second half of the 1990s.

4.1 A generic algorithm

Every $x \in \nu F$ is generated by F from one or more sets $S_x \subseteq \nu F$ with the property that $x \in F(S_x)$. In fact, as F is required to be monotone, it suffices to consider the smallest sets with this property. Let us call the individual sets S_x the (smallest) *support sets* of x and their collection

$$G_x = \{\, S_x \subseteq \mathcal{U} \mid x \in F(S_x) \text{ and } \forall X \subseteq \mathcal{U} : x \in F(X) \Rightarrow S_x \subseteq X \,\}$$

the *generating set* for x. One way to solve the membership problem for an object x then is to start from νF and to trace F "backwards" in order to find a support set for x. If such a set exists, one can conclude that $x \in \nu F$; if it cannot be found, one has to conclude that $x \notin \nu F$. Pictorally, the chase for a $S_x \in G_x$ corresponds to a search graph with νF as its root and the inverse applications of F as its edges.

Clearly, the search for S_x will be inefficient, if elements $x \in \nu F$ can be generated by F in different ways—that is, if $|G_x| > 1$: The search tree then would necessarily be branching, potentially leading to combinatorial explosion and to an exponential complexity of the algorithm. There is no problem, however, if there is always at most *one* path backwards. Generating functions F with this property (that is, with the property that $\forall x \in \mathcal{U}$: $0 \leq |G_x| \leq 1$) will be called *invertible,* and only invertible generating functions will be considered in what follows.

For invertible generating functions, the *support function*

$$\mathsf{support}_F(x) = \begin{cases} X & \text{if } G_x = \{X\}, \\ \uparrow & \text{if } G_x = \emptyset \end{cases}$$

is well-defined and can be lifted to sets by defining

$$\mathsf{support}_F(X) = \begin{cases} \bigcup_{x \in X} \mathsf{support}_F(x) & \text{if } \mathsf{support}_F(x)\downarrow \text{ for all } x \in X, \\ \uparrow & \text{otherwise.} \end{cases}$$

The *support graph* for a universe $\mathcal{U}$ of objects and a generating function F is the graph whose edges are the elements of $\mathcal{U}$ and that contains an edge (x, y) whenever x is F-supported by y, that is, $y \in \mathsf{support}_F(x)$.

With all these concepts defined, the membership problem can be reduced to a reachability problem in the support graph: For a set X of objects to belong to the greatest fixed point of F on $\mathcal{U}$, X must be F-supported, and all the elements that F-support X must be reachable from X in the support graph:

$$\mathsf{gfp}_F(X) = \mathsf{support}_F(X)\downarrow \wedge \big(\mathsf{support}_F(X) \subseteq X \vee \mathsf{gfp}_F(\mathsf{support}_F(X) \cup X)\big) \qquad (1)$$

4.2 Correctness

This section will establish a correctness result for the generic algorithm in (1). Remember that F is taken to be invertible.

The first step is to establish the relation between the notion of "being generated by F" and the notion of "being F-supported".

Lemma 1 $X \subseteq F(Y)$ if and only if $\mathsf{support}_F(X)\downarrow$ and $\mathsf{support}_F(X) \subseteq Y$.

Proof: I will show that $x \in F(Y)$ if and only if $\mathsf{support}_F(x)\downarrow$ and $\mathsf{support}_F(x) \subseteq Y$.

If $x \in F(Y)$, then G_x is non-empty; at least Y is a generating set for x. In particular, since F is invertible, $\mathsf{support}_F(x)$, the smallest generating set, exists, and $\mathsf{support}_F(x) \subseteq Y$. Now, if $\mathsf{support}_F(x) \subseteq Y$, then $F(\mathsf{support}_F(x)) \subseteq F(Y)$ due to the monotonicity of F. By the definition of $\mathsf{support}$, $x \in F(\mathsf{support}(x))$, so $x \in F(Y)$. □

Lemma 2 Let P be a fixed point of F. Then $X \subseteq P$ if and only if $\mathsf{support}_F(X)\downarrow$ and $\mathsf{support}_F(X) \subseteq P$.

Proof: Recall that $P = F(P)$ and apply the previous lemma. □

With the previous two lemmas, it is possible to show partial correctness of the algorithm: If the algorithm terminates, it yields the correct result.

Theorem 2 (Partial correctness) Consider the algorithm in equation (1).

1. If $\mathsf{gfp}_F(X) = \textbf{true}$, then $X \subseteq \nu F$.
2. If $\mathsf{gfp}_F(X) = \textbf{false}$, then $X \nsubseteq \nu F$.

Proof: The proof works by induction on the recursive structure of gfp_F.

1. If $\mathsf{gfp}_F(X) = \textbf{true}$, then either
   (a) $\mathsf{support}_F(X) \subseteq X$, or
   (b) $\mathsf{gfp}_F(\mathsf{support}_F(X) \cup X) = \textbf{true}$.

   In the first case, by lemma 2, $X \subseteq F(X)$, that is, X is F-consistent; therefore, $X \subseteq \nu F$ by the coinduction principle. In the second case, $\mathsf{support}_F(X) \cup X \subseteq \nu F$ by the induction hypothesis, and so $X \subseteq \nu F$.

2. If $\mathsf{gfp}_F(X) = \textbf{false}$, then either
   (a) $\mathsf{support}_F(X)\uparrow$, or
   (b) $\mathsf{gfp}_F(\mathsf{support}_F(X) \cup X) = \textbf{false}$.

   In the first case, by lemma 2, $X \nsubseteq \nu F$. In the second case, $\mathsf{support}_F(X) \cup X \nsubseteq \nu F$, that is, $X \nsubseteq \nu F$ or $\mathsf{support}_F(X) \nsubseteq \nu F$. Either way, $X \nsubseteq \nu F$ by lemma 1.

This concludes the proof. □

Termination measure

To prove termination of the algorithm, it is important to identify reasons for possible divergence. Clearly, gfp_F will diverge if $\mathsf{support}_F(X)$ is infinite for some $X \subseteq \mathcal{U}$. Translated to the graph metaphor, this is the case when the set of nodes reachable from the nodes associated to X is infinite.

Definition 6 (Reachable elements) Let X be a set of nodes in the search graph of the algorithm in (1), and let $\mathsf{predecessors}^i$ be a function that returns the predecessors of the nodes in X reachable via paths of length i. Then the *set of elements reachable from* X is defined as

$$\mathsf{reachable}_F(X) = \bigcup_{n \geq 0} \mathsf{predecessors}^n(X).$$

Termination can be guaranteed for generating functions that yield finite sets of reachable elements.

Definition 7 An invertible function F is said to be *finite state* if $\mathsf{reachable}_F(X)$ is finite for all $X \subseteq \mathcal{U}$.

Theorem 3 If F is finite state, $\mathsf{gfp}_F(X)$ terminates for any finite $X \subseteq \mathcal{U}$.

Proof: Define $\mathsf{next}_F(X) := \mathsf{support}_F(X) \cup X$. For each call of gfp_F,

$$\mathsf{next}_F(X) \subseteq \mathsf{reachable}_F(X).$$

Moreover, $\mathsf{next}_F(X)$ strictly increases on each recursive call. Since $\mathsf{reachable}_F(X)$ is finite, the following serves as a decreasing termination measure:

$$m(\mathsf{gfp}_F(X)) := |\mathsf{reachable}_F(X)| - |\mathsf{next}_F(X)|$$

This concludes the proof. □

4.3 Refinements of the algorithm

The generic algorithm presented in (1) can be refined in a number of ways.

Adding an accumulator

First, an obvious deficiency of the current form is that the set of F-supporting elements for a given set X, $\mathsf{support}_F(X)$, is recomputed in every recursive call. This inefficiency can be overcome by distinguishing between *goals,* elements that still need to be shown to be reachable in the support graph, and *assumptions,* elements already known to be reachable—a common technique called "adding an accumulator". The modified form of the algorithm will be named gfp^a_F:

$$\mathsf{gfp}^a_F(A,X) = \mathsf{support}_F(X)\downarrow \wedge \big(X = \emptyset \vee \mathsf{gfp}^a_F(A \cup X, \mathsf{support}_F(X) \setminus (A \cup X))\big) \quad (2)$$

Initially, gfp^a_F is called with an empty set of assumptions and X as the set of goals. The relation between gfp^a_F and gfp_F then is that

$$\mathsf{gfp}_F(X) = \mathbf{true} \quad \Longleftrightarrow \quad \mathsf{gfp}^a_F(\emptyset, X) = \mathbf{true}.$$

A nice property of gfp^a_F is that it is tail-recursive.

Threading the assumptions

Form (2) of the algorithm shares assumptions about F-supporting elements among function calls at different levels of recursion, but not among calls at the same level of recursion. This can be achieved by splitting up the goal set into individual elements, calling the algorithm recursively for each of them, and threading the assumptions through the call sequence. The resulting function will be named gfp^t_F; in contrast to the former two forms of the algorithm, it does not return **true** or **false**, but the set of newly computed assumptions. It also is no longer tail-recursive, as gfp^a_F was. The relation between gfp^t_F and the original gfp_F is that

$$\mathsf{gfp}_F(\{x\}) = \mathbf{true} \quad \Longleftrightarrow \quad \mathsf{gfp}^t_F(\emptyset, x)\downarrow.$$

5 Subtyping for recursive types

This last section of the paper will instantiate the generic algorithm that was developed in the previous section with the subtype relation established in section 21.3 of Pierce's book, extended to recursive types.

$$\frac{S <: T[X/\mu X.\, T]}{S <: \mu X.\, T} \qquad \frac{S[X/\mu X.\, S] <: T}{\mu X.\, S <: T}$$

Figure 4: Subtyping rules for μ-folding

The task of a subtype algorithm is to check whether or not two (potentially recursive) μ-types stand in the subtype relation. This subtype relation is obtained by extending the subtype relation for types in $\lambda_{\rightarrow}$ by two clauses for types formed using the μ-operator (figure 4). These rules reflect the intuition that the subtype question is decided "in the limit": A recursive type $\mu X.\, S$ is a subtype (supertype) of a type T if its unfolding is.

The subtype relation on μ-types is the greatest fixed point of the following generating function:

$$\begin{aligned} S_\mu(R) = {} & \{\, (S, \top) \mid S \in \mathcal{T}_\mu \,\} \\ & \cup \{\, (S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1) \in R, (S_2, T_2) \in R \,\} \\ & \cup \{\, (S_1 \rightarrow S_2, T_1 \rightarrow T_2) \mid (T_1, S_1) \in R, (S_2, T_2) \in R \,\} \\ & \cup \{\, (S, \mu X.\, T) \mid (S, T[X/\mu X.\, T]) \in R \,\} \\ & \cup \{\, (\mu X.\, S, T) \mid (S[X/\mu X.\, S], T) \in R \,\} \end{aligned}$$

Before the formal results of the previous section can be applied to this generating function, we need to establish that it is both monotone and invertible. The first task is simple, although a proof is omitted due to lack of space. Note however, that, in its present form, S_μ is *not* invertible: The generating set for the pair $(\mu X.\, \top, \mu Y.\, \top)$ contains two sets $\{(\top, \mu Y.\, \top)\}$ and $\{(\mu X.\, \top, \top)\}$. In order to make S_μ invertible, the symmetry between its last two clauses needs to be eliminated by introducing additional conditions in either clause.

Assuming that S_μ can be made invertible, from (1) we automatically obtain an algorithm gfp_{S_μ} that checks for membership in the subtype relation, as well as its partial correctness.

To prove termination, we still need to show that $\mathsf{reachable}_{S_\mu}(S, T)$ is finite for any pair (S, T) of μ-types. This proof requires a surprising amount of work, which is due to the fact that the subexpressions of μ-types can be defined in two different ways: The set of *top-down subexpressions* directly corresponds to the subexpressions generated by the $\mathsf{support}_{S_\mu}$ function; the set of *bottom-up subexpressions* is more suitable for the proof that the set of subexpressions is finite. The complete proof proceeds by showing that the former set is a subset of the latter.

This proof approach is due to Brandt and Henglein (Brandt and Henglein 1998).

Definition 8 (Bottom-up subexpressions of a μ-type)

$$\begin{aligned}
\mathsf{Sub}_B(R) = {} & \{\, (T,T) \mid T \in \mathcal{T}_\mu \,\} \\
& \cup \{\, (S, T_1 \times T_2) \mid (S,T_1) \in R \,\} \\
& \cup \{\, (S, T_1 \times T_2) \mid (S,T_2) \in R \,\} \\
& \cup \{\, (S, T_1 \to T_2) \mid (S,T_1) \in R \,\} \\
& \cup \{\, (S, T_1 \to T_2) \mid (S,T_2) \in R \,\} \\
& \cup \{\, (S[X/\mu X.\, T], \mu X.\, T) \mid (S,T) \in R \,\}
\end{aligned}$$

Proposition 1 $\{\, S \mid (S,T) \in \mu\,\mathsf{Sub}_B \,\}$ is finite.

Definition 9 (Top-down subexpressions of a μ-type)

$$\begin{aligned}
\mathsf{Sub}_T(R) = {} & \{\, (T,T) \mid T \in \mathcal{T}_\mu \,\} \\
& \cup \{\, (S, T_1 \times T_2) \mid (S,T_1) \in R \,\} \\
& \cup \{\, (S, T_1 \times T_2) \mid (S,T_2) \in R \,\} \\
& \cup \{\, (S, T_1 \to T_2) \mid (S,T_1) \in R \,\} \\
& \cup \{\, (S, T_1 \to T_2) \mid (S,T_2) \in R \,\} \\
& \cup \{\, (S, \mu X.\, T) \mid (S, T[X/\mu X.\, T]) \in R \,\}
\end{aligned}$$

Proposition 2 $\mu\,\mathsf{Sub}_T \subseteq \mu\,\mathsf{Sub}_B$

More efficient algorithms

The first, exponential-time algorithm for the subtyping problem for recursive types is due to Abadi and Cardelli (1993).

Dexter Kozen, Jens Palsberg and Michael Schwartzbach published a quadratic algorithm based on a reformulation of the subtype relation on μ-terms in terms of finite-state automata in 1993 (Kozen et. al. 1993). The proof that two μ-terms are in the subtype relation can then be reduced to the proof that a product automaton of their term automata (which can be constructed in quadratic time) does not contain any accepting states (a linear-time problem).

Please note that all references refer to the bibliography of Pierce's book.