Declaration-free Type Checking[1]

Prateek Mishra, Uday S. Reddy[2]
University of Utah

Abstract

Conventional Milner-style polymorphic type checkers automatically infer types of functions and simple composite objects such as tuples. Types of recursive data structures (e.g. lists) have to be defined by the programmer through an abstract data type definition. In this paper, we show how abstract data types, involving type union and recursion, can be automatically inferred by a type checker. The language for describing such types is that of *regular trees*, a generalization of regular expressions to denote sets of tree structured terms. Inference of these types is reducible to the problem of solving simultaneous inclusion inequations over regular trees. We present algorithms to solve such inequations. Using these techniques, programs without any type definitions and type annotations for functions can be type checked.

Keywords

Type inference, Equational languages, Regular Trees, Solving inequations

1. Introduction

Static type checking is the compile time prediction of type errors that may occur during the execution of a program. *Static type inference*, on the other hand, is the compile time prediction of the type of values that an expression may take at run-time. While any type-checker must carry out some amount of type inference, the amount required depends upon the declarative information presumed to be available in the program.

[1]Work supported by University of Utah fellowships and a grant from IBM Corporation.

[2]Authors' address: Department of Computer Science, University of Utah, Salt Lake City, UT 84112. Arpanet mailboxes: MISHRA@UTAH-20, U-REDDY@UTAH-20.

Pascal-style type checkers [6] (for functional languages) require type declarations for all functions. Types of expressions (function applications) are then inferred from these declarations. On the other hand, Milner-style type checkers [10] require declarations only for *primitive functions* which are then used to infer the types of defined functions. "Primitive functions" here include not only those on base types (int, bool etc.) but also those on user-defined types introduced through abstract data type definitions. *We present here a type-checker that can infer types of functions in the absence of such type definitions.* In other words, programs with no declarative information can be type checked.

We illustrate our approach in a first-order functional language which treats terms over a set of *constructors* (uninterpreted functions) as data structures. There are several applicative languages which use such data structures (e.g. HOPE [1], Standard ML [11], SASL [20], FEL [8, 12] and Prolog [3]). Consider the following definitions in such a language:

```
flatten nil = nil
flatten [x] = [x].nil
flatten (x.y) = append [flatten x, flatten y]

append [nil, y] = y
append [a.x, y] = a . append[x,y]

for example:
  flatten ([1].([2].[3])).(nil.[4]) =
        [1].[2].[3].[4].nil
  append [ [1].nil, [2].[3].nil ] =
        [1].[2].[3].nil
```

Example 1: The function flatten

Our type checker reasons about the type of the first argument of append as follows. Suppose the type is A. It is clear that A includes the term nil (from the first equation) and all terms of the form a.x (from the second equation) provided x is itself included in A (because it is used as argument to the recursive call of append). Postulating that a is of an unknown type α, the type checker can infer the definition of A to be

$A = \text{nil} + \alpha.A$

which describes precisely "lists" over objects of type α. Reasoning similarly, the type checker can infer the types for append and flatten given in example 2.

In the above + denotes type union and **fix** is the fixed

$$\text{append} : \forall\, \alpha,\psi.\ [\textbf{fix}\ \xi.(\text{nil}+\alpha.\xi),\ \psi] \rightarrow \textbf{fix}\ \tau.(\psi+\alpha.\tau)$$

$$\textbf{fix}\ \xi.(\text{nil}+\alpha.\xi) = \{\text{nil},\ a_1.\text{nil},\ a_1.a_2.\text{nil},\ \ldots \mid a_1,a_2,\ldots \in \alpha\}$$

$$\textbf{fix}\ \tau.(\psi+\alpha.\tau) = \{y,\ a_1.y,\ a_1.a_2.y,\ \ldots \mid y \in \psi,\ a_1,a_2,\ldots \in \alpha\}$$

$$\text{flatten} : \forall\, \alpha.\ \textbf{fix}\ \xi.(\text{nil}+[\alpha]+\xi.\xi) \rightarrow \textbf{fix}\ \tau.(\text{nil}+[\alpha].\tau)$$

$$\textbf{fix}\ \xi.(\text{nil}+[\alpha]+\xi.\xi) = \{\text{nil},\ [a_1],\ \text{nil}.[a_1],\ [a_1].\text{nil},\ [a_1].[a_2],\ \ldots \mid a_1,a_2,\ldots \in \alpha\}$$

$$\textbf{fix}\ \tau.(\text{nil}+[\alpha].\tau) = \{\text{nil},\ [a_1].\text{nil},\ [a_1].[a_2].\text{nil},\ \ldots \mid a_1,a_2,\ldots \in \alpha\}$$

Example 2: Types of append and flatten

point operator on types used to express recursive types. Each of the data types is expressed in the language of *regular trees* introduced in section 2.

Our type checker infers the type of append and flatten from the respective function definitions and "invents" the appropriate abstract data types required to express the function types. In contrast, a conventional Milner-style type checker requires an abstract data type definition for trees of the form

$$\text{tree}\ \alpha = \text{nil} + [\alpha] + (\text{tree}\ \alpha)\ .\ (\text{tree}\ \alpha)$$

with the following (implicit) declarations for the types of primitive functions:

$$\text{nil} : \text{tree}\ \alpha$$
$$[\_] : \alpha \rightarrow \text{tree}\ \alpha$$
$$\_.\_ : [\text{tree}\ \alpha,\ \text{tree}\ \alpha] \rightarrow \text{tree}\ \alpha$$

It would then infer the types of the defined functions to be

$$\text{append} : [\text{tree}\ \alpha,\ \text{tree}\ \alpha] \rightarrow \text{tree}\ \alpha$$
$$\text{flatten} : \text{tree}\ \alpha \rightarrow \text{tree}\ \alpha$$

Comparing the types inferred by the two type checkers, notice that the type inferred by our type checker is more general for flatten and somewhat less general but more accurate for append. A Milner-style type checker, depending upon primitive function declarations for its information, can only infer that flatten produces trees. On the other hand, our type checker, working directly with the function definition, infers that flatten can only produce flat trees ($\textbf{fix}\ \tau.(\text{nil}+[\alpha].\tau)$). For append, our type checker infers the first argument type to include "lists" ($\textbf{fix}\ \xi.(\text{nil}+\alpha.\xi)$) so that the function application

append [[1].[2], nil]

is found to be ill-typed. Such an application is well-typed for the Milner-style type checker (because [1].[2] is a tree) though it would produce an error at run-time.

Parametric polymorphism [10], that obtained through (universally quantified) variables in the types of functions, is available in our system. For instance, the types of flatten and append inferred above contain variables α and ψ. In addition, our system offers additional polymorphic flexibility in the following fashion. Firstly, since there are no predefined types in our framework, the constructors do not possess any *a priori* types. Consequently, a constructor can be used in constructing any number of types. In the types above, the constructor "." is used in four distinct types. There is no reason why one of these should be considered the "rightful owner" of ".". Our system thus incorporates polymorphism due to "unbounded overloading" of constructors. This feature allows us to infer more accurate types than is possible in Milner's framework, as commented earlier.

Secondly, through the use of the + operator which yields the sum of two types, a form of "additive" polymorphism becomes available. If a value belongs to the type τ it belongs to any type which includes τ as a summand. This is a form of *containment polymorphism* [14, 2]. To the best of our knowledge, this is the first attempt to use additive polymorphism in its full power by permitting the sum of arbitrary types to be a type. Semantic soundness, as formulated in [10], is still preserved in spite of these rather free polymorphic notions.

The remainder of the paper describes *effective* algorithms to carry out type inference as illustrated above. In section 2, we define regular trees which form the language of our base types. Section 3 deals with type inference. The problem of type correctness is reduced to that of solving systems of simultaneous inclusion inequations over regular terms. The rest of the paper describes how to solve such inequations. In order to focus on the novel aspects of our work, we shall only consider first order functions over a flat domain. Our type inference techniques can be combined with those of Milner's to infer the types of higher order functions. A semantic model of types based on ideals [9] instead of sets would be necessary to capture partial and infinite objects.

2. Regular Trees

2.1. Definitions

We assume that the data structures in a program are built from an alphabet Γ of *constructors*. In some languages, the constructors are syntactically distinguishable from other symbols. In the others, functions without definitions are assumed to be constructors. A *term* is either a constructor used as an atom or a constructor applied to a term. The tuple constructor [] is always assumed to be available as a primitive constructor. It is the only constructor that can be applied to an arbitrary number of arguments. We do not assume that constructors have any prescribed arities. As arities are nothing but a weak notion of types, such a prescription conflicts with our philosophy that constructors have no *a priori* types.

Definition: A *regular tree* over a set of constructors Γ using a set of variables **Var** is

1. the empty regular tree ϕ,
2. a variable $\alpha \in$ **Var**,
3. an atom $c \in \Gamma$,
4. an application cr, where $c \in \Gamma$ and r is a regular tree,
5. a tuple $[r_1, \ldots, r_n]$, where each r_i is a regular tree,
6. a sum $r_1 + r_2$, where r_1 and r_2 are regular trees,
7. a conjunction $r_1 \& r_2$, where r_1 and r_2 are regular trees, or
8. a fixed point **fix** α. r where $\alpha \in$ **Var** and r is a regular tree.

Note that every term is a regular tree. We shall often use the set of all terms, denoted by H, as a formal symbol, even though it is not a regular tree.

$\Psi_0[[\ \text{red} + \text{blue} + \text{green}\ ]] = \{\ \text{red}, \text{blue}, \text{green}\ \}$
$\Psi[[\ \text{succ}(\alpha)\ ]]\ \rho = \{\ \text{succ}(t) \mid t \in \rho\alpha\ \}$
$\Psi_0[[\ \text{fix}\ \beta.\ (\text{nil} + 1.\beta)\ ]] =$
$\quad \{\ \text{nil}, 1.\text{nil}, 1.1.\text{nil}, \ldots\ \}$
$\Psi_0[[\ \textbf{fix}\ \rho.\ (\text{nil} + \alpha.\rho)\ ]] =$
$\quad \{\text{nil}, \alpha.\text{nil}, \alpha.\alpha.\text{nil}, \ldots\}$
$\Psi[[\ \text{fix}\ \beta.\ (\text{nil} + \alpha.\beta)\ ]]\ \rho =$
$\quad \{\text{nil}, t_1.\text{nil}, t_1.t_2.\text{nil}, \ldots \mid t_1, t_2, \ldots \in \rho\alpha\}$
$\Psi_0[[\ (\beta.\gamma)\ \&\ (\textbf{fix}\ \rho.\ (\text{nil} + \alpha.\rho))\ ]] =$
$\quad \{(\alpha\&\beta).(\text{nil}\&\gamma), (\alpha\&\beta).(\alpha.\text{nil}\&\gamma),$
$\quad (\alpha\&\beta).(\alpha.\alpha.\text{nil}\&\gamma), \ldots\ \}$

Example 3: Regular Trees

Regular sets of trees that are recognizable by tree automata have been studied before [19] and used in data flow analysis problems similar to type inference [17, 7]. Our regular trees are different from them in that they contain free variables and they have a conjunction operation.

2.2. Semantic Types

Semantically, a "monotype" (type without parameters) is simply a set of terms. But, a parameterized type over a set of free variables Δ is a function[3]

$$\text{Type} = \text{Environment} \rightarrow \text{P(Term)}$$

where

$$\text{Environment} = \Delta \rightarrow \text{P(Term)}$$

P(Term) is a complete lattice under set inclusion and types are monotonic functions of environments.

We shall use another equivalent domain of types, viz., sets of terms with free variables. This representation is convenient to define the notion of a "language" denoted by a regular tree.

Definition: A *parameterized tree* over constructors Γ and free variables Δ is either ϕ or a term over $\Gamma \cup \Delta \cup \{\&\}$.

Sets of parameterized types can now be considered types. We first define a mapping Ψ that maps a parameterized tree to a type.

$\Psi[[\ \phi\ ]]\ \rho = \{\}$
$\Psi[[\ \alpha\ ]]\ \rho = \rho\alpha$
$\Psi[[\ c\ ]]\ \rho = \{c\}$
$\Psi[[\ cp\ ]]\ \rho = \{ct \mid t \in \Psi[[p]]\rho\}$
$\Psi[[\ [p_1, \ldots, p_n]\ ]]\ \rho =$
$\quad \{[t_1, \ldots, t_n] \mid t_1 \in \Psi[[p_1]]\rho, \ldots, t_n \in \Psi[[p_n]]\rho\}$
$\Psi[[\ p_1\&p_2\ ]]\ \rho = \{t \mid t \in \Psi[[p_1]]\rho, t \in \Psi[[p_2]]\rho\}$

Unfortunately, parameterized trees are not yet suitable as semantic objects because many of them are equivalent.

Lemma 1: The following equations form a complete equational theory of parameterized trees.

1. $r \ \& \ s = s \ \& \ r$
2. $r \ \& \ (s \ \& \ t) = (r \ \& \ s) \ \& \ t$
3. $r \ \& \ \phi = \phi$
4. $\alpha \ \& \ \alpha = \alpha$
5. $c \ \& \ c = c$
6. $cr \ \& \ cs = c\ (r \ \& \ s)$
7. $cr \ \& \ ds = \phi$ where $c \neq d$
8. $c \ \& \ cr = \phi$

[3] P(A) denotes the powerset of A.

9. $[r_1,\ldots,r_n]$ & $[s_1,\ldots,s_n]$ = $[r_1\&s_1,\ldots,r_n\&s_n]$
10. $[r_1,\ldots,r_m]$ & $[s_1,\ldots,s_n]$ = ϕ where $m \neq n$
11. c & $[s_1,\ldots,s_n]$ = ϕ
12. cr & $[s_1,\ldots,s_n]$ = ϕ
13. cϕ = ϕ
14. $[t_1,\ldots,t_{k-1},\phi,t_{k+1},\ldots,t_n]$ = ϕ

Further, these equations, excepting 1 and 2, treated as rewrite rules form a term rewriting system that is confluent up to associativity and commutativity of &.

The term rewriting system can be used to normalize parameterized trees. For example,

$\beta.\gamma$ & α.nil $\twoheadrightarrow$ ($\alpha\&\beta$).(γ&nil)

We call the normal forms under this term rewriting system *normal parameterized trees.* They have the pleasant property that if $t_1\&\ldots\&t_n$ is a conjunctive term occurring in a normal parameterized tree, then all but one of t_1, ..., t_n are variables. Therefore, we usually write normal parameterized trees in the form **g&t** where g, called the *guard* of the parameterized tree, is a conjunction of free variables, and t is an atom or a constructor application. Since H&t = t for all t, a term without a guard may be considered to have H as its guard.

Non-empty sets of normal parameterized trees are now equivalent to parameterized trees, with the bijection h between them defined by

$h\ T\ \rho = \bigcup_{t \in T} \Psi[[t]]\ \rho$

Definition: The *language* of a regular tree r, denoted $\Psi_0[[r]]$, is a set of non-empty parameterized trees obtained from the following inductive definition after normalization and deletion of ϕ:

$\Psi_0[[\ \phi\ ]] = \{\}$
$\Psi_0[[\ \alpha\ ]] = \{\alpha\}$
$\Psi_0[[\ c\ ]] = \{c\}$
$\Psi_0[[\ cr\ ]] = \{ct \mid t \in \Psi_0[[r]]\}$
$\Psi_0[[\ [r_1,\ldots,r_n]\ ]] =$
  $\{[t_1,\ldots,t_n] \mid t_1 \in \Psi_0[[r_1]],\ \ldots,\ t_n \in \Psi_0[[r_n]]\}$
$\Psi_0[[\ r_1 + r_2\ ]] = \Psi_0[[r_1]] \cup \Psi_0[[r_2]]$
$\Psi_0[[\ r_1\ \&\ r_2\ ]] =$
  $\{t_1\&t_2 \mid t_1 \in \Psi_0[[r_1]],\ t_2 \in \Psi_0[[r_2]]\}$
$\Psi_0[[$ **fix** $\alpha.\ r\ ]] =$
  the smallest set T of terms such that
  $T = \bigcup_{t \in T} \Psi_0[[r]][t/\alpha]$

We can now extend the semantic function Ψ for all regular trees

$\Psi[[r]]\ \rho = \bigcup_{t \in \Psi_0[[r]]} \Psi[[t]]\ \rho$

If the programming language works in a non-flat (lazy) domain of terms, then the simple semantics for regular trees given here is not adequate. As discussed in [9] types have to be considered as ideals. An ideal semantics can be given for regular trees.

2.3. Cartesian closed types

In conventional type systems, type addition is interpreted as "discriminated union" and objects of sum types have to be explicitly projected to their component types by projection functions such as *outl* and *outr* [10]. In our framework, the constructor functions serve the purpose of projection. For example, the list type involves the sum

nil + α . (list α)

A term belonging to the list type can be projected to one of the summands depending on whether the outermost constructor is nil or ".". When the two summands of a type have the same outermost constructor, such as in cr+cs, the distributivity of constructor application can be used to rewrite it as c(r+s), in effect delaying the discrimination. Types for which discrimination can be successfully delayed are *discriminative types.* We shall give a complete definition of discriminative types in section 4.2. Here we examine a particular aspect of discriminative types. When the type is a sum of two n-ary cartesian products

$[r_1,\ldots,r_n] + [s_1,\ldots,s_n]$

discrimination cannot be delayed. Therefore, we restrict our language of types to exclude such sums of cartesian products.

Definition: Let T be a set of terms. Let t be a term in T, in which a tuple $[x_1,\ldots,x_n]$ occurs. Suppose there is another term u in T, so that

$u = t\ [[y_1,\ldots,y_n]\ /\ [x_1,\ldots,x_n]]$

T is said to be *cartesian closed* if whenever such t and u are in T all the terms in

$\{t[\ [z_1,\ldots,z_n]/[x_1,\ldots,x_n]\ ] \mid z_i=x_i \text{ or } z_i=y_i\ \}$

are also in T. A regular tree r is *cartesian closed* if $\Psi_0[[r]]$ is.

We consider functions that are not defined on cartesian closed domains to be incomplete. The following definition of equality on natural numbers

```
eq [0, 0] = true
eq [succ x, succ y] = eq [x, y]
```

is incomplete, because it does not define eq[0, succ 0]. In order to make it complete, the clauses

```
eq [0, succ y] = false
eq [succ x, 0] = false
```

have to be added. When presented with an incomplete definition, the type checker considers one or more subsets of the domain type which are cartesian closed. For the above incomplete definition there exists only one cartesian closed subset of the domain, viz., [0,0].

A family of subsets of a type is called its *cartesian basis*, if every cartesian closed subset of the type is included in some member of the family. The cartesian basis of a regular tree type is composed of only regular tree types. To get a flavor of the concept, consider the following result.

Lemma 2: If $[r_1,\ldots,r_n]$ and $[s_1,\ldots,s_n]$ are cartesian closed regular trees then the following regular trees form a cartesian basis of $[r_1,\ldots,r_n] + [s_1,\ldots,s_n]$

$[r_1,\ldots,r_n]$,
$[s_1,\ldots,s_n]$,
$[r_1+s_1, r_2\&s_2,\ldots,r_n\&s_n]$,
$\ldots$
$[r_1\&s_1,\ldots,r_{n-1}\&s_{n-1},r_n+s_n]$

The property of cartesian closedness limits the "resolution" of our language of types. For example, the regular tree

fix α.(1.α + 2.nil)

that denotes the set of lists with arbitrary number of 1's followed by a 2, is not cartesian closed. Its cartesian closure is

fix α.((1+2).(nil+α))

which includes all non-empty lists of 1's and 2's. Thus, some of the fine-grained information about a set of terms is lost in taking its cartesian closure. If the program needs to make use of this fine-grained information, the programmer has to use a different constructor for the tail end of the list, e.g.

fix α.(1.α + 2:nil)

3. Type Inference

In this section, we illustrate how type inference is performed for a simple first-order functional language which operates on terms as data objects. For specifying functions, we use a *pattern-based* λ-abstraction construct similar to the one in Standard ML [11]. Its syntax is

$(\lambda p_1.e_1;\ \ldots;\ \lambda p_n.e_n)$

where p_i are *patterns* (linear non-ground terms made of constructors and variables), and e_i are expressions. Function application is performed through pattern-matching. The application

$((\lambda p_1.e_1;\ \ldots;\ \lambda p_n.e_n)\ d)$

is rewritten by $e_i\theta$ provided θ is a substitution so that $p_i\theta$=d. *If there is no* p_i *with such a substitution, or if there is more than one, then a type error occurs.*

The semantic domain of values of the language includes the set of all ground terms (the Herbrand Universe) written H and functions defined on H. The distinguished value wrong is used to formalize run-time errors.

V = H + [H -> H] + {wrong}

The type inference is based on inference rules similar to those in [4, 9]. We give below the rules that are specific to our language.

FUN $\dfrac{A, A' \vdash p : r \quad A, A' \vdash e : s}{A \vdash \lambda p.e : r \to s}$
where A' is the type environment for the variables in p

UNION $\dfrac{A \vdash e_1 : r_1 \quad A \vdash e_2 : r_2}{A \vdash (e_1;e_2) : r_1\&r_2}$

CONT $\dfrac{A \vdash e : r \quad r \leq s}{A \vdash e : s}$

COMB $\dfrac{A \vdash e : s\to r \quad A \vdash e' : s' \quad A \vdash s' \leq s}{A \vdash (ee') : r}$

APPROX $\dfrac{A \vdash e : (s \to r)\ \&\ (u \to t)}{A \vdash e : (s + u) \to (r + t)}$

REC $\dfrac{A, f : t \vdash e : t}{A \vdash (\text{fix } f.e) : t}$

The rule for type containment (CONT) is not directly used in our type inference algorithm. It is used in obtaining the derived rules COMB and APPROX. The rule APPROX can be justified easily using the properties of the -> constructor given in [9].

$$\begin{aligned}(s \to r)\ \&\ (u \to t) &\leq (s \to r+t)\ \&\ (u \to r+t)\\ &= (s+u \to r+t)\end{aligned}$$

Theorem 3: *The above type inference system is semantically sound.*

The rule COMB is the main workhorse of the inference system. In inferring the type of an expression, it produces a set of inclusion inequations which have to be solved to find instantiations for the variables. Let us illustrate this through an example. The equations for append given in section 1, are expressed using pattern-based λ-abstraction as

append = $\mathtt{fix}\ f.\ e$
where $e = (\lambda[\mathtt{nil},y].y;\ \lambda[a.x,z].(a.f[x,z]))$

Using the type inference rules, we can deduce the following:

$$\vdash \lambda[\mathtt{nil},y].y : [\mathtt{nil},\psi]\rightarrow\psi$$

$$\frac{f : \sigma\rightarrow\tau \vdash [\xi,\zeta] \leq \sigma}{f : \sigma\rightarrow\tau \vdash \lambda[a.x,z].(a.f[x,z]) : [\alpha.\xi,\zeta]\rightarrow\alpha.\tau}$$

Combining the two, using rules UNION and APPROX,

$$\frac{f : \sigma\rightarrow\tau \vdash [\xi,\zeta] \leq \sigma}{f : \sigma\rightarrow\tau \vdash e : ([\mathtt{nil},\psi]+[\alpha.\xi,\zeta]) \rightarrow (\psi+\alpha.\tau)}$$

In order to use the rule REC, we identify the types on both the sides of $\vdash$.

$$\frac{[\xi,\zeta] \leq ([\mathtt{nil},\psi]+[\alpha.\xi,\zeta])}{\vdash \text{append} : ([\mathtt{nil},\psi]+[\alpha.\xi,\zeta]) \rightarrow \mathbf{fix}\ \tau.(\psi+\alpha.\tau)}$$

The inclusion inequation above the bar has to be now "solved" to find instantiations for the free variables. A solution comprises values for the type variables α, ψ, ξ and ζ which determine the domain type of the function. We are looking for the largest solution if one exists, or a complete set of maximal solutions otherwise.

The following steps are then followed:

1. Replace the RHS by each member of its cartesian basis. This gives rise to four possibilities which have to be treated separately.
   A. $[\xi,\zeta] \leq [\mathtt{nil},\psi]$
   B. $[\xi,\zeta] \leq [\alpha.\xi,\zeta]$
   C. $[\xi,\zeta] \leq [\mathtt{nil}+\alpha.\xi,\ \psi\&\zeta]$
   D. $[\xi,\zeta] \leq [\mathtt{nil}\&\alpha.\xi,\ \psi+\zeta]$
   The case D is trivial since $\mathtt{nil}\&(\alpha.\xi) = \phi$.
2. Solve the inclusion inequations to produce inclusion assumptions. (An inclusion assumption is an inequation whose LHS or RHS is a variable).
   A. $\{\xi \leq \mathtt{nil},\ \zeta \leq \psi\}$
   B. $\{\xi \leq \alpha.\xi,\ \zeta \leq \zeta\}$
   C. $\{\xi \leq \mathtt{nil}+\alpha.\xi,\ \zeta \leq \psi,\ \zeta \leq \zeta\}$
3. Eliminate variables shared by both sides of an inclusion assumption, to obtain inclusion bounds. (An assumption of the form $\sigma < f(\sigma)$ holds only if σ is included in the largest fixed point of f. Similarly, $f(\sigma) \leq \sigma$ holds only if $\mathbf{fix}\ \sigma.f(\sigma) < \sigma$. Often there is a unique fixed point of the f involved. If the largest fixed point is the entire Herbrand Universe H, then σ has no constraints and should be left free).
   A. $\{\xi \leq \mathtt{nil},\ \zeta \leq \psi\}$
   B. $\{\xi \leq \phi,\ \zeta \leq H\}$
   C. $\{\xi \leq \mathbf{fix}\ \xi.(\mathtt{nil}+\alpha.\xi),\ \zeta \leq \psi\}$
   If the semantic domain contained infinite terms, then the case B would produce a nontrivial upper bound
   $\xi \leq \mathbf{fix}\ \xi.(\alpha.\xi)$
4. Find an instantiation for each variable depending on whether it is to be maximized or minimized. (The variables introduced as types of formal parameters contribute to the domain; they should be maximized. Other variables should be minimized. If σ is to be maximized and the only upper bound on it is H, then σ has no constraints and should be left free).
   A. $\{\xi \rightarrow \mathtt{nil},\ \zeta \rightarrow \psi\}$
   C. $\{\xi \rightarrow \mathbf{fix}\ \xi.(\mathtt{nil}+\alpha.\xi),\ \zeta \rightarrow \psi\}$

The instantiations can now be substituted in the domain type of append. There are four possible cartesian closed domain types, but, for this example, one of them (C) happens to be the largest. (In general, the distinct cartesian closed domain types may not be comparable).

A. $[\mathtt{nil},\ \psi]$
C. $[\mathbf{fix}\ \xi.(\mathtt{nil}+\alpha.\xi),\ \psi]$

The type of append is finally inferred to be

$$\forall\ \alpha,\psi.\ [\mathbf{fix}\ \xi.(\mathtt{nil}+\alpha.\xi),\ \psi]\rightarrow \mathbf{fix}\ \tau.(\psi+\alpha.\tau)$$

To illustrate function application, let us infer the type of the function rev defined by

rev = $\mathtt{fix}\ f.$
$\quad(\lambda[\mathtt{nil}].\mathtt{nil};\ \lambda[b.z].\text{append}\ [f[z],\ b.\mathtt{nil}])$

The type information obtained from the definition is

$$\frac{\begin{array}{l}\vdash \zeta \leq (\mathtt{nil} + \beta.\zeta),\\ \vdash [(\mathtt{nil} + \mathbf{fix}\ \tau.(\psi+\alpha.\tau)),\ \beta.\mathtt{nil}]\\ \qquad \leq [\mathbf{fix}\ \xi.(\mathtt{nil}+\alpha.\xi),\ \psi]\end{array}}{\vdash \text{rev} : (\mathtt{nil} + \beta.\zeta) \rightarrow (\mathtt{nil} + \mathbf{fix}\ \tau.(\psi+\alpha.\tau))}$$

The inclusion assumptions obtained by solving these inequations are

$\zeta \leq (\mathtt{nil} + \beta.\zeta)$
$\psi \leq \mathbf{fix}\ \xi.(\mathtt{nil}+\alpha.\xi)$
$\beta.\mathtt{nil} \leq \psi$

Combining the last two assumptions gives rise to a new inequation

$\beta.\mathtt{nil} \leq \mathbf{fix}\ \xi.(\mathtt{nil}+\alpha.\xi)$

solving which we obtain the assumption

$\beta \leq \alpha$

Since β and ζ are domain variables, they should be instantiated to their upper bounds. β has no upper bound; so it is left free. (Even though it is included in α, α itself has no upper bound).

$$\zeta \rightarrow \mathbf{fix}\ \zeta.\ (\mathrm{nil}+\alpha.\zeta)$$

α and ψ, introduced through the function type of append, are range variables. They are instantiated to their lower bounds.

$$\alpha \rightarrow \beta$$
$$\psi \rightarrow \beta.\mathrm{nil}$$

giving the type of rev to be

$$\forall\ \beta.\ \mathbf{fix}\ \zeta.(\mathrm{nil} + \beta.\zeta) \rightarrow \mathbf{fix}\ \zeta.(\mathrm{nil} + \beta.\zeta)$$

3.1. Generic Recursion

Even though the type produced for append above is a valid type, it is not clear what its relationship is to the most general type of append and how it automatically turned out to be regular. We have made two approximations to the most general type, first, in using the rule APPROX, and second, in assuming that recursive functions do not behave generically within their own definitions as in [4]. Using the inference rule REC:

$$\frac{f:\sigma\rightarrow\tau \vdash [\xi,\zeta] \leq \sigma}{f:\sigma\rightarrow\tau \vdash e : ([\mathrm{nil},\psi] + [\alpha.\xi,\zeta]) \rightarrow \alpha.\tau}$$

and identifying $\sigma\rightarrow\tau$ with the type of e, amounts to assuming that the free variables α,ξ,ζ are quantified over the entire assertion and cannot be *instantiated* to behave generically *within* the definition of f. (Once we complete the computation of the type of append, we use universal instantiation to obtain a polymorphic type, and thereby allow f to behave generically in other expressions).

If, on the other hand, we permit recursive functions to behave generically within their own definitions [18, 15], we can quantify free variables before completing the computation of the function type:

$$f:\sigma\rightarrow\tau \vdash$$
$$e : (\forall\ \psi,\alpha,\xi,\zeta \mid [\xi,\zeta]\leq\sigma)\ ([\mathrm{nil},\psi]+[\alpha.\xi,\zeta]\rightarrow \alpha.\tau)$$

$\sigma\rightarrow\tau$ now only needs to be a generic instance of this type. But the type itself involves σ and τ; consequently we must consider a second distinct generic instance for this occurrence of σ and τ. After instantiation and identifying the types of f and e, we obtain

$$f:\sigma_1\rightarrow\tau_1 \vdash$$
$$\begin{aligned} e : (\forall\ &\psi,\alpha,\xi,\zeta,\psi_1,\alpha_1,\xi_1,\zeta_1 \mid \\ &[\xi,\zeta] \leq ([\mathrm{nil},\psi_1] + [\alpha_1.\xi_1,\zeta]), \\ &[\xi_1,\zeta_1] \leq \sigma_1) \\ &[\mathrm{nil},\psi] + [\alpha.\xi,\zeta] \rightarrow \psi + \alpha.\psi + \alpha.\alpha_1.\tau_1 \end{aligned}$$

If we continue this process of generic instantiation to the limit, effectively allowing each invocation of f to behave generically, we will obtain the type:

$$\begin{aligned} e : [&\mathrm{nil}+\alpha.\mathrm{nil}+\alpha.\alpha_1.\mathrm{nil}+\ldots,\ \psi\&\psi_1\&\psi_2\&\ldots]\rightarrow \\ &\psi + \alpha.\psi_1 + \alpha.\alpha_1.\psi_2 + \ldots \end{aligned}$$

which is clearly not regular.

Instead of the two extremes, we choose to treat recursion as being generic for the first invocation and non-generic thereafter. This is achieved by instantiating all the quantified variables in the type of e, but not those that appear in the type environments. A generic instance of the type of e by this method is

$$([\xi_1,\zeta_1] \leq \sigma)\ ([\mathrm{nil},\psi_1] + [\alpha_1.\xi_1,\zeta_1] \rightarrow \psi_1+\alpha_1.\tau_1)$$

The advantage of this method may be seen from the following example

```
rev [nil,x] = x
rev [a.y,z] = rev [y,a.z]
```

The type information obtained from the program is

$$\frac{\mathrm{rev}:\sigma\rightarrow\tau \vdash [\psi,\alpha.\zeta] \leq \sigma}{\mathrm{rev}:\sigma\rightarrow\tau \vdash \mathrm{rev} : [\mathrm{nil},\xi]+[\alpha.\psi,\zeta] \rightarrow \xi+\tau}$$

If we do not treat recursion as generic at all, solving the inequation yields the assumptions

$$\psi \leq \mathrm{nil} + \alpha.\psi$$
$$\alpha.\zeta \leq \xi$$
$$\alpha.\zeta \leq \zeta$$

and the instantiations

$$\psi \rightarrow \mathbf{fix}\ \psi.\mathrm{nil} + \alpha.\psi$$
$$\xi = \zeta \rightarrow \mathbf{fix}\ \zeta.\delta + \alpha.\zeta$$

which yields the following valid but very coarse type for reverse:

$$\begin{aligned} \forall\alpha,\delta.\ &[\mathbf{fix}\ \psi.(\mathrm{nil} + \alpha.\psi),\ \mathbf{fix}\ \zeta.(\delta + \alpha.\zeta)] \\ &\rightarrow \mathbf{fix}\ \zeta.(\delta + \alpha.\zeta) \end{aligned}$$

With one step generic treatment of recursion, we have the type

$$\frac{\begin{array}{l}\vdash [\psi,\alpha.\zeta] \leq [\mathrm{nil},\xi_1]+[\alpha_1.\psi_1,\zeta_1] \\ \vdash [\psi_1,\alpha_1.\zeta_1] \leq [\mathrm{nil},\xi_1]+[\alpha_1.\psi_1,\zeta_1]\end{array}}{\vdash \mathrm{rev} : [\mathrm{nil},\xi]+[\alpha.\psi,\zeta] \rightarrow \xi_1+\xi}$$

The inequations yield the assumptions

$$\psi \leq \mathrm{nil} + \alpha.\psi_1$$
$$\alpha.\xi \leq \xi_1$$
$$\alpha.\xi \leq \zeta_1$$
$$\psi_1 \leq \mathrm{nil} + \alpha_1.\psi_1$$
$$\alpha_1.\xi_1 \leq \xi_1$$
$$\alpha_1.\xi_1 \leq \zeta_1$$

and the instantiations

$\psi \rightarrow \mathtt{nil} + \alpha_1 . (\mathbf{fix}\ \psi_1 . (\mathtt{nil} + \alpha_1 . \psi_1))$

α, α_1, ξ and ζ free

$\xi_1 \rightarrow \mathbf{fix}\ \xi_1 . (\alpha . \xi + \alpha_1 . \xi_1)$

$\zeta_1 \rightarrow \mathbf{fix}\ \xi_1 . (\alpha . \xi + \alpha_1 . \xi_1)$

Substituting in the type of reverse, and identifying α_1 and α which corresponds to a restriction of the inferred type and hence is a valid step, we have the following type:

$$\forall \alpha, \xi, \zeta .\ [\mathbf{fix}\ \beta . (\mathtt{nil} + \alpha . \beta),\ \xi \& \zeta] \\ \rightarrow \xi + \mathbf{fix}\ \xi_1 . (\alpha . \zeta + \alpha . \xi_1)$$

In solving for the type of a function we extract from the function representation two pieces of information. Firstly, we estimate the largest possible *domain* type. To do this, we express the values a parameter may take in any instantiation of the function in terms of the constraints placed on the parameter by *future* instantiations of the function. Secondly, we estimate the smallest possible *range* type for the function. In doing this, we express the values a parameter may take in any instantiation in terms of the values constructed during *previous* instantiations of the function. This bidirectional information flow is modeled by the one-step generic treatment of recursion.

4. Algorithms on Regular Trees

4.1. Leaf-linear systems

In this section we introduce an alternative representation for regular trees called leaf-linear systems of equations which are easier to work with.

Definition: A *system of leaf-linear equations* is a 5-tuple $<V,\Delta,\Gamma,Z,E>$ where V is a set of variables, Δ is a set of free variables, Γ is a set of constructors, $Z \in V \cup \Delta$ is a distinguished variable, and E is a set of equations of the form

$$X = g_1 \& t_1 + \ldots + g_n \& t_n$$

with exactly one equation for each $X \in V$. Each g_i, called a *guard*, is a conjunction of free variables Δ. Each t_i, called a *transition*, is of one of the following forms:

1. H, signifying "no-transition",
2. an atom $c \in \Gamma$,
3. a variable $Y \in V$,
4. an application cY where $c \in \Gamma$ and $Y \in V \cup \Delta$,
5. a tuple $[Y_1,\ldots,Y_n]$ where $Y_i \in V \cup \Delta$

An *interpretation* of a leaf-linear system associates with each variable $X \in V$, a set of parameterized trees over Γ and Δ. There exists a unique least interpretation that satisfies the system of equations. We denote this interpretation by Ω. The *language* of a leaf-linear system is ΩZ.

Operationally leaf-linear systems can be viewed in two ways. An equation may be considered as a rewrite rule so that a variable may be rewritten by a term in the RHS of its equation. By this interpretation, a leaf-linear system acts as a grammar that generates its language. Alternatively, a leaf-linear system may be interpreted as a finite-state machine whose states are the variables and the equations specify the possible transitions from the states.

Theorem 4: A set of parameterized trees over Γ and Δ is denoted by a regular tree R if and only if it is the language of a leaf-linear system $<V,\Delta,\Gamma,Z,E>$.

Proof:

* If:

Construction of an equivalent leaf-linear system L by induction of R. Let

$L_1 = <V_1,\Delta_1,\Gamma,Z_1,E_1>$

...

$L_n = <V_n,\Delta_n,\Gamma,Z_n,E_n>$

be leaf-linear systems equivalent to regular trees $R_1,\ldots,R_n$ such that $V_1,\ldots,V_n$ are pairwise disjoint and Z is a new variable not in any of them.

- $R = \phi$
  $L = <\{Z\},\{\},\Gamma,Z,\{\}>$
- $R = \alpha$
  $L = <\{\},\{\alpha\},\Gamma,\alpha,\{\}>$
- $R = c$
  $L = <\{Z\},\{\},\Gamma,Z,\{Z = c\}>$
- $R = R_1 + R_2$
  $L = <V_1 \cup V_2 \cup \{Z\},\ \Delta_1 \cup \Delta_2,\ \Gamma,\ Z,\ E_1 \cup E_2 \cup \{Z = Z_1 + Z_2\}>$
- $R = cR_1$
  $L = <V_1 \cup \{Z\},\ \Delta_1,\ \Gamma,\ Z,\ E_1 \cup \{Z = cZ_1\}>$
- $R = [R_1,\ldots,R_n]$
  $L = <V_1 \cup \ldots \cup V_n \cup \{Z\},\ \Delta_1 \cup \ldots \cup \Delta_n,\ \Gamma,\ Z,\ E_1 \cup \ldots \cup \{Z = [Z_1,\ldots,Z_n]\}>$
- $R = \mathbf{fix}\ \alpha . R_1$
  $L = <V_1 \cup \{\alpha\},\ \Delta_1 - \{\alpha\},\ \Gamma,\ \alpha,\ E_1 \cup \{\alpha = Z_1\}>$
- $R = R_1 \ \&\ R_2$
  $L = <V_1 \cup V_2 \cup (V_1 \times V_2),\ \Delta_1 \cup \Delta_2,\ \Gamma,\ <Z_1,Z_2>,\ E_1 \cup E_2 \cup E>$
  If $(X_1 = \Sigma\ T_1) \in E_1$ and $(X_2 = \Sigma\ T_2) \in E_2$ include in E, an equation
  $$<X_1,X_2> = \Sigma_{t_1 \in T_1,\ t_2 \in T_2}\ t_1 \& t_2$$
  after normalizing and replacing conjunctions of variables $Y_1 \& Y_2$ by defined variables $<Y_1,Y_2>$ in the summands.

* Only if:

Let $<\{X_1,\ldots,X_n\},\ \Delta,\ \Gamma,\ X_d,\ \{X_1=t_1,\ldots,X_n=t_n\}>$ be a leaf-linear system. We give an algorithm to eliminate the free variables from the right hand sides of the equations.

```
for i := 1 to n do
  replace the equation $X_i=t_i$ by $X_i$=fix $X_i.t_i$
  for j := i+1 to n do
    substitute $X_i$ in $t_j$ by fix $X_i.t_i$
(* Now jth equation does not have $X_i$ free for
   $i \leq j$ *)
(* Let the new set of equations be
   $\{\ldots X_i = t'_i \ldots\}$ *)
for i := n downto 1 do
  for j := i-1 downto 1 do
    substitute $X_i$ in $t'_j$ by $t'_i$
(* Now each equation has a regular tree
   on its right hand side *)
```

The right hand side of the equation for X_d is the regular tree denoting the language of the leaf-linear system.

The leaf-linear system

$$L\alpha\beta = <\{X,Y,Z,A,B,C,D\},\{\alpha,\beta\},\{nil,\hat{}\},X,E> \text{ with}$$

$$\begin{aligned} E = \{X &= \hat{}Y + \hat{}Z, \\ Y &= [A,B], \\ A &= \beta, \\ B &= nil, \\ Z &= [C,D], \\ C &= \alpha, \\ D &= X\} \end{aligned}$$

has as its language, the regular tree

$$\text{fix } \xi.(\hat{}[\alpha,nil] + \hat{}[\beta,\xi])$$

Example 4: Leaf-linear systems

4.2. Discriminative Types

Discriminativity means that whenever a type is expressed as a sum of two types T_1+T_2, a value belonging to the sum type can be projected into one of the subtypes by looking only at its outermost constructor. Thus, discriminativity is a notion describing type expressions. However, not all types have discriminative expressions. Therefore, we shall first characterize the types which have such expressions. Note that it is not necessary for all parameterized trees in a discriminative type to have distinct outermost constructors. For example, the type {ca, cb} can be expressed by the discriminative expression c(a+b).

Definition: A type T is *unambiguous* if for all distinct t_1, $t_2 \in$ T, $t_1\&t_2 = \phi$.

If a type is ambiguous, whatever maybe its *representation, a value can genuinely belong to two* subtypes. So, it cannot be discriminative. Domain types of functions are guaranteed to be unambiguous.

For an unambiguous type, we attempt to find a discriminative representation by delaying discrimination (by "pushing + down") whenever necessary. Such delaying cannot be always be done if the summands have cartesian products and conjunctions at the outermost level. Hence, we have the following definition of discriminativity.

Definition: A type T is *discriminative* if

1. it is unambiguous,
2. it is cartesian closed, and
3. if, whenever *T* contains two parameterized trees t and u which differ at some position containing $g_1\&t'$ and $g_2\&u'$ respectively with t' and u' having the same outermost constructor, $g_1 = g_2$.

If a type is not cartesian closed then we consider each member of its maximal cartesian basis. If it is nondiscriminative due to conjunctions, then we transform it to a new type over a different set of free variables. For example, consider the type $\{\alpha\&ca,\ cb\}$. The two parameterized trees have the same outermost constructor, but different guards. In order to find an equivalent discriminative type, we replace α by $c\alpha_1 + \alpha_2$ involving new free variables α_1 and α_2. Further, we place a constraint on α_2 that it cannot be bound to a type containing c-terms. We denote this constraint by

$$\alpha_2 \sim c?$$

where "~" denotes disjointness relation, and "?" stands for a unique variable. With this transformation, the type becomes $\{c(\alpha_1\&a),\ cb\}$ which is discriminative. Note that the transformation does not involve any loss of generality. Given any instantiation $\rho\alpha$, we can find an environment ρ' for α_1 and α_2:

$$\rho'\alpha_1 = \{t \mid ct \in \rho\alpha\}$$
$$\rho'\alpha_2 = \rho\alpha - \{ct \mid t \in \rho'\alpha_1\}$$

In general, when a type over free variables Δ is transformed to a discriminative type over free variables Δ', we produce a set of disjointness constraints Λ for Δ', and *a substitution Φ mapping Δ to types over Δ'. This induces* a function Ψ from environments of Δ' satisfying Λ to those of Δ.

$$(\Psi\rho)\alpha = \Psi\ [[\Phi\alpha]]\ \rho\ \forall\ \alpha \in \Delta$$

The Γ and Λ are such that Γ^{-1} always exists.

Definition: A leaf-linear system is *discriminative* if, on right hand side of each equation,

1. there is a no-transition summand, there are no other summands,
2. there is at most one atom transition c,
3. there is at most one constructor application transition for any constructor c,
4. there is at most one n-tuple transition for any n, and
5. there is no variable transition X.

We now describe the transformation of a given leaf-linear system to a discriminative one, so that latter is a member of the cartesian basis of the former.

Variable transitions can be eliminated, without altering the language of a leaf-linear system, using the following rules:

1. if $X = g\&X + \Sigma\ T$ is an equation then rewrite the equation to $X = \Sigma_{t \in T}\ g\&t$.
2. if $X = g\&Y + \Sigma\ T$ is an equation where $Y \in V$, $Y \neq X$, then replace Y by its definition.

The first rule preserves least fixed points whereas the second rule is a substitution of equals for equals. The rules can be applied in a two-pass elimination algorithm similar to the one in theorem 4.

In transforming a leaf-linear system into a discriminative one, we need to construct equations for new variables whose languages are conjunctions and sums of those of original variables. As notation for these new variables we use "e" where e is an expression involving sums and conjunctions of original variables. The definition of a conjunct variable "A&B" can be constructed from those of A and B using the algorithm in theorem 4. The definition of a sum variable "A+B" is obtained by adding the individual definitions of A and B and transforming it to discriminative form.

A sum of leaf-linear trees $\Sigma\ T$ is transformed into discriminative form $\Sigma\ T'$ as follows:

* If T contains a no-transition summand g, it has to be a singleton since it is unambiguous. Then T' = {g}.
* T can contain only one summand with an atom transition, g&c, for any c, since it is unambiguous. Include g&c in T'.
* Let $g_1\&cA_1, \ldots, g_k\&cA_k$ be all the summands in T with c-application transitions. For each variable α appearing in the guards $g_1, \ldots, g_n$, if α has a disjointness constraint $\alpha \sim c?$, replace the summand by ϕ. Otherwise, instantiate α to $c\alpha_1 + \alpha_2$ together with a disjointness constraint $\alpha_2 \sim c?$. Construct a new guard g as the conjunction of all the freshly introduced α_1's. Include the leaf-linear tree

  g & c "A_1+...+A_n"

  in T'.
* Let $g_1\&[A_{11},\ldots,A_{1n}], \ldots, g_k\&[A_{k1},\ldots,A_{kn}]$ be all the summands in T with n-tuple transitions. For each variable α appearing in the guards, if α has a disjointness constraint $\alpha \sim [?,\ldots?]$ with n ?'s replace the summand by ϕ. Otherwise, instantiate α to $[\alpha_{11},\ldots,\alpha_{1n}] + \alpha_2$ together with a disjointness constraint $\alpha_2 \sim [?,\ldots,?]$. Nondeterministically choose a member M from the cartesian basis of $[A_{11},\ldots,A_{1n}] + \ldots + [A_{k1},\ldots,A_{kn}]$. Construct a new guard g as the conjunction of the freshly introduced α_{1i}'s corresponding to the guards of the transitions involved in M. Include the leaf-linear tree

  g & M

  in T'.

The discriminative systems for the cartesian basis of the leaf-linear system L$\alpha\beta$ of example 4 are the following:

```
L1 = <{X,"Y+Z",A,B}, {β}, {nil,^}, X, E1>
E1 = {X = ^ "Y+Z", "Y+Z" = [A,B], A = β, B = nil}

L2 = <{X,"Y+Z",C,D}, {α}, {nil,^}, X, E2>
E2 = {X = ^ "Y+Z", "Y+Z" = [C,D], C = α,
      D = ^ "Y+Z"}

L3 = <{X,"Y+Z","A+C","B&D"}, {α,β}, {nil,^},X, E3>
E3 = {X = ^ "Y+Z", "Y+Z" = ["A+C","B&D"],
      "A+C" = α+β, "B&D" = ϕ}

L4 = <{X,"Y+Z","A&C","B+D"}, {α,β}, {nil,^},X, E4>
E4 = {X = ^"Y+Z", "Y+Z" = ["A&C","B+D"],
      "A&C" = α&β, "B+D" = nil+^"Y+Z"}
```

L2 and L3 have empty languages, L1 has only lists of length 1, while L4 permits arbitrary lists over $\alpha\&\beta$.

Example 5: Cartesian basis

4.3. Solving Inequations

An inclusion inequation $r \leq s$ where r and s are regular trees (with possibly free variables shared by r and s) is *satisfiable* if there exists an environment ρ such that $\Psi[[r]]\rho$ is a subset of $\Psi[[s]]\rho$. An *inclusion assumption* on ρ is either of the form $\alpha \leq t$ or $t \leq \alpha$ where t is a regular tree in which α may occur. A set of inclusion assumptions is a *solution* of an inclusion inequation if every ρ that satisfies the assumptions satisfies the inequation. A set of solutions for an inequation is *complete* if every ρ that satisfies the inequation satisfies one of the solutions. There is a natural partial order on solutions based on generality. An inequation may not in general have a unique most general solution that is complete. The following result identifies the case that is relevant in type inference.

Theorem 5: If r is a conjunction-free regular tree and s is regular tree denoting a discriminative type, then the inequation $r \leq s$ has a most general solution that is complete.

If the right hand side type is not discriminative, the inequation may not have a finite number of complete solutions. For example, consider

$$\text{fix}\ \tau.\ (\text{nil} + \text{a}.\tau) \leq \alpha + \beta$$

Any set of assumptions of the form $\{r_1 \leq \alpha,\ r_2 \leq \beta\}$ where r_1 denotes of a subset of lists over a, and r_2 denotes its complement, is a solution of the inequation. These solutions are incomparable and there are an infinite number of them.

In order to solve the inequation $r \leq s$, we construct a discriminative leaf-linear system $<V,\Delta,\Gamma,Z,E>$ for s. We then use the following inference system in a goal-directed fashion to find the set of inclusion assumptions necessary to infer $r \leq Z$.

1. $$\frac{A \vdash r \leq t \qquad (X{=}t) \in E}{A \vdash r \leq X}$$

2. $$\frac{A \vdash r \leq t}{A \vdash r \leq t{+}u}$$

3. $$\frac{A \vdash r \leq t \qquad A \vdash r \leq u}{A \vdash r \leq t\&u}$$

4. $A,\ r \leq \alpha \vdash r \leq \alpha$

5. $A,\ \alpha \leq t \vdash \alpha \leq t$

6. $A \vdash c \leq c$

7. $$\frac{A \vdash r \leq t}{A \vdash cr \leq ct}$$

8. $$\frac{A \vdash r_1 \leq t_1 \quad \ldots \quad A \vdash r_n \leq t_n}{A \vdash [r_1,\ldots,r_n] \leq [t_1,\ldots,t_n]}$$

9. $$\frac{A \vdash r \leq t \qquad A \vdash s \leq t}{A \vdash r{+}s \leq t}$$

10. $A,\ \text{fix}\ \alpha.r \leq X \vdash \text{fix}\ \alpha.r \leq X$

11. $$\frac{A,\ \text{fix}\ \alpha.r \leq X \vdash r[\text{fix}\ \alpha.r\ /\ \alpha] \leq X}{A \vdash \text{fix}\ \alpha.r \leq X}$$

The inference system can be used with resolution semantics to resolve goals of the form

$A \vdash r \leq Z$

where A is an unknown. Resolution produces a binding for A consisting of the inclusion assumptions required to infer $r \leq Z$ [5, 16]. To argue about the effectiveness of the inference system, let us note that it is deterministic except for the rule 2 and the choice between rules 10 and 11. Since the sum $t{+}u$ in rule 2 is discriminative, the choice of the summand can be made deterministically by looking at the outermost constructor. Among rules 10 and 11, choosing 10 whenever applicable guarantees termination because the number of variables is finite. Choosing rule 11 does not produce any new assumptions, but reproves an inequation that is already proved. Therefore, the set of inclusion assumptions produced by resolution is unique.

4.4. Consistency of inclusion assumptions

The set of inclusion assumptions is closed for transitivity of $\leq$ by adding the assumptions corresponding to $r \leq s$ whenever it contains two assumptions of the form $r \leq \beta$ and $\beta \leq s$. This follows from the transitivity of the inclusion relation; a set of inclusion assumptions is satisfiable only if the transitive closure of the inclusion assumptions is satisfiable.

Definition: The transitive closure of a set of inclusion assumptions S, is the smallest set of inclusion assumptions R, satisfying:

1. $S \leq R$,
2. if $\alpha \leq r$ and $s \leq \alpha$ belong to R, then the inclusion assumptions generated from the inclusion inequation $r \leq s$ belong to R.

In order to guarantee termination the transitive closure must be generated with some care. As the following example demonstrates taking the transitive closure may yield an existing inclusion assumption.

$\mathtt{succ}(\alpha) \leq \alpha,\ \alpha \leq \mathtt{fix}(\zeta).\ \mathtt{zero} + \mathtt{succ}(\zeta)$

$\vdash \mathtt{succ}(\alpha) \leq \mathtt{fix}(\zeta).\ \mathtt{zero} + \mathtt{succ}(\zeta)$

$\vdash \alpha \leq \mathtt{fix}(\zeta).\ \mathtt{zero} + \mathtt{succ}(\zeta)$

Example 6: Transitive Closure

The transitive closure algorithm picks all pairs of inclusion assumptions of the form $\alpha \leq \mathtt{r},\ \mathtt{s} \leq \alpha$ and adds the result of applying Theorem 5 to the set of inclusion assumptions. If any new assumptions have been generated the process is repeated. Termination is assured as all *new* inclusion assumptions involve regular trees that are strictly smaller than the regular trees submitted to the inclusion algorithm. The computation cannot therefore continue indefinitely and must terminate.

5. Generating instantiations

As illustrated in section 4.3, the inequations produced in type inference can be solved to obtain a set of inclusion assumptions. In order to obtain the "most general" or "principal" type for an expression, we have to find the most general instantiations for the variables involved. The type variables that appear in the inequations are of two kinds: those assumed to be the types of formal parameters in rule FUN, called *domain variables* and those appearing in the polymorphic types of functions used in the body, called *generic variables*. The domain variables determine the domain type of a function, whereas generic variables determine - directly or indirectly - the range type of a function.

Definition: An instantiation $<\bar{\alpha}_1,\bar{\delta}_1>$ of domain variables $\bar{\alpha}$ and generic variables $\bar{\delta}$, is *more general* than another instantiation $<\bar{\alpha}_2,\bar{\delta}_2>$ if

1. $\bar{\alpha}_2 \leq \bar{\alpha}_1$ is satisfiable, and
2. if there is a substitution θ such that $\bar{\alpha}_2\theta = \bar{\alpha}_1\theta$ then $\bar{\delta}_2\theta \geq \bar{\delta}_1\theta$.

The notion of generality on instantiations corresponds directly to that on types of functions. Condition 1 in the above definition captures the fact that a larger domain type yields a more general type of a function. Condition 2 says that given identical domain types (up to substitutions θ), the more general instantiation produces a smaller range type.

We now describe an algorithm to find the most general instantiation for a set of inclusion assumptions generated during type inference. Note the following properties that the assumptions satisfy:

1. As function parameters can only occur as components of expressions input to other functions, their types - domain variables $\bar{\alpha}$ - occur only on the left hand sides of inequations. Consequently, the generated inclusion assumptions also have them only on left hand sides.
2. Further, the generic variables can be separated into two disjoint classes. Some generic variables, $\bar{\gamma}$, occur in regular tree expressions acting as upper bounds on domain variables (say $\alpha_i \leq \mathtt{r}(\bar{\gamma})$). These generic variables arise from the polymorphic types of functions to which function parameters are input. As a type variable can occur only once in the domain type of a function, there are no other assumptions of the form $\mathtt{r} \leq \gamma_i$; such variables can only participate in assumptions of the form $\gamma_i \leq \mathtt{r}(\bar{\gamma})$. The second class of generic variables, β, may appear in assumptions of the form $\beta_i \leq \mathtt{s}$ and $\mathtt{r} \leq \beta_i$.

We now organize the inclusion assumptions into four classes ($\bar{\alpha}$ are domain variables, β and $\bar{\gamma}$ are generic variables):

1. $\alpha_i \leq \mathtt{r}_i(\bar{\gamma})$
2. $\gamma_i \leq \mathtt{s}_i(\bar{\gamma})$
3. $\beta_i \leq \mathtt{u}_i(\beta,\bar{\gamma})$
4. $\mathtt{t}_i(\bar{\alpha},\beta,\bar{\gamma}) \leq \beta_i$

so that there is exactly one assumption for each α_i, β_i and γ_i in each appropriate class. If there is no upper bound assumption for a variable γ_i (or α_i) an implicit assumption $\gamma_i \leq \mathtt{H}$ or $\alpha_i \leq \mathtt{H}$ is added. If there is more than one upper bound assumption for a variable, $\alpha_i \leq \mathtt{r}$ and $\alpha_i \leq \mathtt{s}$, they are combined into one assumption $\alpha_i \leq \mathtt{r\&s}$. Similarly, if there is no lower bound assumption for β_i an implicit assumption $\phi \leq \beta_i$ is added, and two lower bound assumptions for the same variable, $\mathtt{r} \leq \beta_i$ and $\mathtt{s} \leq \beta_i$ are combined into $\mathtt{r+s} \leq \beta_i$. Note that this reorganization of the inclusion assumptions does not alter their satisfiability.

The variable elimination algorithm described in sub-section 5.2 is used to obtain most general instantiations for $\bar{\alpha}$, β and $\bar{\gamma}$ as follows:

1. Eliminate variables from inclusion assumptions in 2 and compute *upper bounds* $\bar{\gamma}^0$ for $\bar{\gamma}$.

2. Substitute $\bar{\gamma}^0$ for $\bar{\gamma}$ in the inclusion assumptions in 1; we now have *upper bounds* $\bar{\alpha}^0$ for $\bar{\alpha}$. Instantiating $\bar{\alpha}$ to $\bar{\alpha}^0$ yields new inequations of the form

   $r_i(\bar{\gamma}^0) \le r_i(\bar{\gamma})$

   A simplification of these yields exactly the assumptions

   $\gamma_i^0 \le r_i$

   So, $\bar{\gamma}^0$ are also *lower bounds* on $\bar{\gamma}$.
3. Substitute $\bar{\alpha}^0$ and $\bar{\gamma}^0$ in the lower bound assumptions for β_i in 4. Variable elimination yields lower bounds β_0 for the variables β. Note that the inclusion assumptions in 3, which are upper bound *constraints on β, do not play any role in determining* instantiations for β. $\bar{\gamma}_0$ and β_0 taken together are the lower bound instantiations for the generic variables.

As an example, consider the inclusion assumptions generated for tail recursive **rev** function from section 3.1. The domain variables ($\bar{\alpha}$) are ξ, α, ψ and ζ. The generic variables are ξ_1, α_1, ψ_1 and ζ_1. Of these, α_1 and ψ_1 are used in the upper bounds of domain variables (denoted $\bar{\gamma}$ above). The others, ξ_1 and ζ_1 are bounded below. The assumptions are now organized into four classes:

1. $\psi \le \texttt{nil} + \alpha.\psi_1$

   $\alpha \le H$

2. $\psi_1 \le \texttt{nil} + \alpha_1.\psi_1$

   $\alpha_1 \le H$

3. none

4. $\alpha.\xi + \alpha_1.\xi_1 \le \xi_1$

   $\alpha.\xi + \alpha_1.\xi_1 \le \zeta_1$

Variable elimination on class 2 yields the instantiations $\bar{\gamma}^0$

$\psi_1 \to \textbf{fix}\ \psi_1.(\texttt{nil} + \alpha_1.\psi_1)$

α_1 free

Substituting these in class 1, we get

$\psi \to \texttt{nil} + \alpha.(\textbf{fix}\ \psi_1.\ (\texttt{nil} + \alpha_1.\psi_1))$

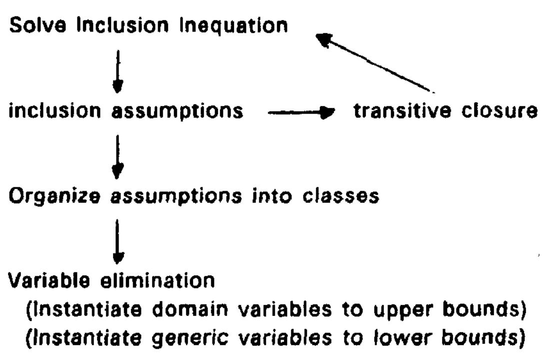
α free

Finally, variable elimination on 4 yields

$\xi_1 \to \textbf{fix}\ \xi_1.(\alpha.\xi + \alpha_1.\xi_1)$

$\zeta_1 \to \textbf{fix}\ \xi_1.(\alpha.\xi + \alpha_1.\xi_1)$

We now describe techniques for elimination of variables from sets of inclusion assumptions. In subsection 5.1, we discuss the computation of the largest or least instantiation satisfying a single inclusion assumption. In subsection 5.2, we discuss its generalization to sets of inclusion assumptions.



Solve Inclusion Inequation
inclusion assumptions
transitive closure
Organize assumptions into classes
Variable elimination
(Instantiate domain variables to upper bounds)
(Instantiate generic variables to lower bounds)

Figure 5-1: Solving inclusion inequations

5.1. Single Inclusion Assumptions

A possible infinity of instantiations may satisfy a given inclusion assumption. We display the solution as a *min-max* pair consisting of the smallest and the largest instantiations that satisfy the inclusion assumption.

As **P(Term)** is a complete lattice, all monotonic functions possess both largest and least fixed points written **lfp** and **fix** respectively. For $\alpha \le r$ the min-max instantiation is $[\phi,\ \textbf{lfp}\ \alpha.r]$; similarly for $r \le \alpha$ the min-max instantiation is $[\textbf{fix}\ \alpha.r,\ H]$.

In our formalism, we do not possess any means of expressing largest fixed points in a closed form. Theorem 6 states that for most regular trees, the largest fixed point is precisely equal to the least fixed point; further it provides a simple static test for determining whether the two are the same.

Theorem 6: If $r(\alpha)$ is a regular tree, then

1. If $\alpha \in \Psi_0[[r(\alpha)]]$, then **lfp** $\alpha.r(\alpha) = H$.
2. If $\alpha\&t \notin \Psi_0[[r(\alpha)]]$ for any parameterized tree t, then **lfp** $\alpha.r(\alpha)$ = **fix** $\alpha.r(\alpha)$.
3. Let $T = \alpha\&t \in \Psi_0[[r(\alpha)]]$. Define $r'(\alpha)$ such that

   $\Psi_0[[r'(\alpha)]] = \Psi_0[[r(\alpha)]] - \{\alpha\&t \mid t \in T\} + T$

   Then **lfp** $\alpha.r(\alpha)$ = **fix** $\alpha.r'(\alpha)$

Proof: If $\alpha \in \Psi_0[[r(\alpha)]]$, H is a fixed point of $r(\alpha)$. It has to be the largest one.

Suppose $\alpha\&t \notin \Psi_0[[r(\alpha)]]$. $\Psi_0[[r(\alpha)]]$ can be written as the (infinite) sum $\Sigma c_i + \Sigma r_i(\alpha)$ where the $c_i \in$

$\Psi_0[[r(\alpha)]]$ are terms in which α does not occur and $r_i(\alpha) \in \Psi_0[[r(\alpha)]]$ are terms in which α does occur. By abuse of notation we denote by $r_i(S)$ the sum of terms obtained by replacing α by each term in S. The largest fixed point is the limit (intersection) of the approximations $r^n(H)$.

$$r^n(H) = \Sigma c_i + \Sigma r_i(\Sigma c_i) + \ldots + \Sigma r_i(.. \Sigma r_i(\Sigma c_i)..) + \Sigma r_i(.. (\Sigma r_i(H))..)$$

The trailing summand cannot contain any finite terms in the limit. So, its limit is ϕ. The least fixed point is *similarly the limit (union) of the approximations* $r^n(\phi)$.

$$r^n(\phi) = \Sigma c_i + \Sigma r_i(\Sigma c_i) + \ldots + \Sigma r_i(.. \Sigma r_i(\Sigma c_i)..) + \Sigma r_i(.. (\Sigma r_i(\phi))..)$$

Again the trailing summand is ϕ in the limit. The other summands are equal in all the approximations of the least and largest fixed points. Hence, the least fixed point and the largest fixed point are equal.

In the third case, let $\Psi_0[[r(\alpha)]] = \Sigma\ \alpha\&t_i + \Sigma\ \alpha\&u_i(\alpha) + \Sigma\ c_i + \Sigma\ r_i(\alpha)$. Since $H\&t = t$ for any t,

$$r(H) = \Sigma\ t_i + \Sigma\ c_i + \Sigma\ u_i(H) + \Sigma\ r_i(H)$$

which is the same as $r'(H)$. The succeeding approximations of the largest fixed points of r and r' are identical. But the largest fixed point of r' is the *same as its least fixed point, from case 2.*

```
lfp α.(α + zero + nil) = H
lfp α.(zero + succ(α)) = fix α.(zero + succ(α))
lfp α.(α&β) = β
lfp α.(α&zero + succ(α)) = fix α.(zero + succ(α))
```

Example 7: Largest fixed points

Even though the case 3 of theorem 6 looks a little complicated, it is easy to implement using the leaf-linear system representation of regular trees. All the terms of the form $\alpha\&t$ can be found in the equation of the distinguished variable. Moreover, this case subsumes the cases 1 and 2.

5.2. Variable elimination in sets of inclusion assumptions

There are two sets of assumptions from which we eliminate variables: in computing upper bounds on β from assumptions $\beta_i \leq s_i(\beta)$ of class 2 and in computing lower bounds β from assumptions $t_i(\bar{\alpha},\beta) \leq \beta_i$ of class 3. To do this, we use a variant of the variable elimination algorithm of theorem 4.

```
for i := 1 to n do
  replace the assumption $\beta_i \leq r_i$ by $\beta_i \leq$ lfp $\beta_i . r_i$
  for j := i+1 to n do
    substitute $\beta_i$ in $r_j(\beta)$ by lfp $\beta_i . r_i(\beta)$
for i := n downto 1 do
  for j := i-1 downto 1 do
    substitute $\beta_i$ in $r'_j$ by $r'_i$
```

The computation of lower bounds follows in a similar fashion; the least fixed point **fix** is used instead of **lfp**.

```
β1 ≤ nil + β2^β1,
β2 ≤ nil + β1^β2

has solution
let p = lfp β1 . nil + β2^β1
in
β1 = lfp β1 . nil + β2^β1,
β2 = lfp β2 . nil + p^β2
```

Example 8: Computing upper bounds

In computing upper bounds we need to give special consideration to the interaction between containment polymorphism and parametric polymorphism. If we were using only containment polymorphism then a variable α with an upper bound H can be instantiated to H. But, such an upper bound means that the expression whose type is being computed places no constraint on α. So, the type of the expression should be parameterized by α. All the variables with H as their upper bound are left uninstantiated during variable elimination.

Note that no such special consideration is necessary in dealing with lower bounds. A variable β with a lower *bound* ϕ is necessarily a generic variable. It signifies a type in the domain of a polymorphic function that is not being fed any input. So, such a variable is instantiated to ϕ.

6. Conclusion

In summary, our type checking algorithm for first-order functional languages achieves

1. type checking in the absence of abstract data type definitions,
2. polymorphic flexibility through *unbounded* overloading of constructors, and
3. containment polymorphism through type addition.

These are in addition to the features offered by Milner-style type checkers, viz., parametric polymorphism and type checking in the absence of function type declarations.

We are extending our techniques so as to be able to carry out type inference for a higher-order functional language and are also engaged in developing an implementation for a higher-order lazy functional language [8]. We are also investigating the application of the type-checking algorithm for logic languages based on the philosophy outlined in [13].

Acknowledgements

Our thanks to Neil Jones, Jean-Pierre Jounnaud, Robert Keller, David MacQueen, Alan Mycroft and Esther Shilcrat for discussions and comments.

References

[1] Burstall, R. M. , MacQueen, D. B., Sanella, D. T. HOPE: an experimental applicative language. *ACM LISP Conference*, 1980, pp. 136-143.

[2] Cardelli, L., A semantics of multiple inheritance. In *Lecture Notes in Comp. Sci.* Volume 173: *International symposium on semantics of data types*, Springer-Verlag,, 1984, pp. 51-68.

[3] Clocksin, W. F., Mellish, C. S.. *Programming in Prolog*. Springer-Verlag, New York, 1981.

[4] Damas, L., Milner, R. Principal type-schemes for functional programs. *ACM Symp. Principles of Prog. Lang.*, 1982, pp. 207-212.

[5] Despeyroux, T. Executable specifications of static semantaics. In *Lecture Notes in Comp. Sci.* Volume 173: *Semantics of Data Types*, Springer-Verlag,, 1984, pp. 215-233.

[6] Jensen K. and Wirth N.. *PASCAL: User Manual and Report*. Springer-Verlag, 1976.

[7] Jones, N. D., Muchnick, S. S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *ACM Symp. Principles of Prog. Lang.*, 1982, pp. 66-74.

[8] Keller, R. M. *FEL (Function Equation Language) Programmer's Guide*. AMPS Technical Memorandum 7, University of Utah, April, 1982.

[9] D.B. MacQueen, R. Sethi. A semantic model of types for applicative languages. *Conference on LISP and Functional Programming*, August, 1982, pp. 243-252.

[10] Milner, R. A theory of type polymorphism in programming. *J. Computer and System Sciences 17* (1978), pp. 348-375.

[11] Milner, R. A proposal for Standard ML. *ACM Symp. LISP and Functional Prog.*, 1984, pp. 184-197.

[12] Mishra, P. *Data types in applicative languages: abstraction and inference*. University of Utah, 1983.

[13] Mishra, P. Towards a theory of types in Prolog. *International symposium on logic programming*, IEEE, 1984, pp. 289-298.

[14] Mitchell, J.C. Coercion and type inference. *ACM Symp. Principles of Prog. Lang.*, 1984, pp. 175-185.

[15] Mycroft A. Polymorphic Type schemes and Recursive Definitions. *Intl. Symposium on Programming - LNCS*, 1984.

[16] Reddy, U. S. On the relationship between logic and functional languages. In DeGrot, D, Lindstrom, G., Eds.,*Functional and Logic Programming*, Prentice-Hall,, 1985.

[17] Reynolds R. C. Automatic Computation of Data Set Definitions. *IFIP 68*, 1968, pp. 456-461.

[18] A. Shamir, W. Wadge. Data Types as Objects. In *Lecture Notes in Comp. Sci.* Volume 52: Salomaa, A, Ed.,*ALP*, Springer-Verlag,, 1977, pp. 465-479.

[19] Thatcher, J.W. Tree automata: an informal survey. In A.V. Aho, Ed.,*Currents in theory of computing*, Prentice-Hall,, 1973, pp. 143-172.

[20] Turner, D.A. *SASL Language Manual*. St. Andrews University, 1976.