Implementing Regular Tree Expressions

Alexander Aiken
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
aiken@ibm.com

Brian R. Murphy
Computer Science Department
Stanford University
Stanford, CA 94305
brm@cs.stanford.edu

Abstract

Regular tree expressions are a natural formalism for describing the sets of tree-structured values that commonly arise in programs; thus, they are well-suited to applications in program analysis. We describe an implementation of regular tree expressions and our experience with that implementation in the context of the FL type system. A combination of algorithms, optimizations, and fast heuristics for computationally difficult problems yields an implementation efficient enough for practical use.

1 Introduction

Regular tree expressions are a natural formalism for describing the sets of tree-structured values that commonly arise in programs. As such, several researchers have proposed using (variations on) regular tree expressions in type inference and program analysis algorithms [JM79, Mis84, MR85, HJ90,

HJ91, AM91]. We are not aware of any implementations based on regular tree expressions, however, except for our own work on type analysis for the functional language FL [B+89].

A previous paper described the theoretical basis for our FL type inference system, in which types are represented by regular tree expressions [AM91]. This paper describes an implementation of regular tree expressions and our experience with that implementation in the context of the FL type system. Implementing regular tree expressions efficiently is challenging, because some of the basic operations have exponential time complexity [Sei89]. Even some operations with polynomial time algorithms perform poorly in practice. The fundamental operations on regular tree expressions are: computing fixed points, union, intersection, negation, and testing inclusion (including solving sets of inclusion constraints). Of these, both negation and inclusion require exponential time in general. This is particularly troublesome in the case of inclusion, because in our system it is by far the most commonly used operation, and we would expect the same to be true for most applications.

We give an abstract description of a complete implementation of regular tree expressions, including algorithms, heuristics (where necessary), and a number of optimizations. We also present the results of performance measurements on our implementation. The performance measurements show that the amortized cost per regular tree operation is nearly constant in our system.

We begin in Section 2 with a straightforward definition of regular tree expressions; we use this representation to illustrate their usefulness. In Section 3, we introduce the representation used in our system: *leaf-linear systems of equations* [GS84, MR85]. We discuss why leaf-linear systems are better suited to implementation than the representation of Section 2. Section 4 outlines an incremental algorithm to test whether a leaf-linear system is empty. The next several sections describe, at a high level, implementations of the operations on regular tree expressions. For brevity, we call these operations *regular tree operations*. In Section 6, we present a fast heuristic for testing inclusion of leaf-linear systems. The heuristic is conservative: the result is either "yes, the containment holds" or "don't know." This heuristic works surprisingly well in our application; in almost a year of use in the FL type analysis system, we have yet to find a practical example where the quality of information produced by the system is affected by the use of this heuristic.

Section 7 covers two simple optimizations that substantially improve the

E	$::=$	0	$\Psi(0,\sigma)$	$=$	$\emptyset$
	$\mid$	1	$\Psi(1,\sigma)$	$=$	H
	$\mid$	α	$\Psi(\alpha,\sigma)$	$=$	$\sigma(\alpha)$
	$\mid$	$E_1 \vee E_2$	$\Psi(E_1 \vee E_2,\sigma)$	$=$	$\Psi(E_1,\sigma) \cup \Psi(E_2,\sigma)$
	$\mid$	$E_1 \wedge E_2$	$\Psi(E_1 \wedge E_2,\sigma)$	$=$	$\Psi(E_1,\sigma) \cap \Psi(E_2,\sigma)$
	$\mid$	$\neg E_1$	$\Psi(\neg E_1,\sigma)$	$=$	$H - \Psi(E_1,\sigma)$
	$\mid$	$\mathit{fix}\ \alpha.E_1$	$\Psi(\mathit{fix}\ \alpha.E_1,\sigma)$	$=$	least T s.t. $T = \Psi(E_1,\sigma[\alpha \leftarrow T])$
	$\mid$	$c(E_1,\ldots,E_n)$	$\Psi(c(E_1,\ldots,E_n),\sigma)$	$=$	$\{c(t_1,\ldots,t_n) \mid t_i \in \Psi(E_i,\sigma)\}$

Figure 1: Syntax and semantics of regular tree expressions.

performance of intersection, union, negation, and testing inclusion. Both optimizations focus on avoiding computations whenever possible. The first optimization comes from facts such as $(A \subseteq B) \Leftrightarrow (A \cap B = A)$. If it is known that $A \subseteq B$, then we can save time and space by not computing $A \cap B$ but just returning A. Applying similar ideas uniformly dramatically improves performance. The second optimization is to use *memoization* to record and reuse the results of operations [Mic68]. While the usefulness of this optimization depends on the particular application, in our system it is very common for the same operations to be performed again and again. The performance of the system with these two optimizations is more than two orders of magnitude faster than without them. Measurements and discussion of the system's performance are also presented in Section 7.

It is worth explaining our choice of the term "regular tree expression". In [MR85], a regular tree is what we term a regular tree expression. We use a different name because the term "regular tree" is usually a finite or infinite tree with a finite number of subtrees [Cou83]. Using this definition, a regular tree expression together with a substitution denotes a set of regular trees. Regular tree expressions can also be thought of as tree automata with free variables; we discuss this view in Section 3.

2 Regular Tree Expressions

In this section, we introduce regular tree expressions, describe some of their properties, and develop a few examples. The representation shown here is not the one used for implementation; however, we present it first because it is easy to understand and illustrates the important operations. We begin with some basic definitions. Let $C = \{b, c, \ldots\}$ be a set of constructors, each with an arity $a(c)$. We assume this set is constant hereafter.

Definition 2.1 Let C^0 be the set of zero-ary constructors in C. The Herbrand universe is the least set of terms H such that

$$H = C^0 \cup \{c(t_1, \ldots, t_{a(c)}) | c \in C, t_i \in H\}$$

Definition 2.2 A *substitution* σ is a function from some set of variable symbols V to the power set of H. We define $\sigma \uparrow \Delta$ to be the substitution σ restricted to the set of variables Δ.

Regular tree expressions denote sets of terms of the free algebra H. A syntax and semantics for regular tree expressions is given in Figure 1. The meaning function Ψ maps an expression under an environment σ to a subset of H. A variable α is *bound* in an expression if it appears in the scope E of a *fix* $\alpha.E$; otherwise the variable is *free*. We restrict the negation operator to argument expressions with no free variables; this ensures that Ψ is monotonic in its second argument, and thus the least fixed-point operator is well-defined.

Other proposals for program analysis systems based on regular tree expressions adopt slightly different definitions, depending upon the application. For example, Mishra and Reddy restrict the use of disjunction in a type inference algorithm for a statically typed functional language [MR85], and Heintze and Jaffar use projection functions in a program analysis for logic programs [HJ90]. Almost all of our implementation design would apply to these other systems with little or no modification.

The following examples illustrate the usefulness of regular tree expressions. Let c be a binary constructor and let b be a zero-ary constructor. If we interpret c as describing a Lisp `cons` operation and b as describing the atom `nil`, then the set of Lisp lists is *fix* $\alpha.c(1, \alpha) \vee b$. Similarly, the set of binary trees with leaves described by β is *fix* $\alpha.c(\alpha, \alpha) \vee \beta$. A more sophisticated example uses regular tree expressions to infer the types of recursively

defined functions. Consider a recursive function $def\ f \equiv e(f)$. If we assume that f may return a member of the set α and prove from this assumption that $e(f)$ returns a term in the set $E(\alpha)$, we can conclude that f returns any term in $fix\ \alpha.E(\alpha)$.

The grammar of Figure 1 illustrates most of the important operations on regular tree expressions (least fixed-points, union, intersection, and negation). It is also possible to add a greatest fixed-point operation, but this introduces no new ideas in an implementation, so we we do not consider it here. Two other important algorithms test emptiness and inclusion relationships. In the type inference example above, it may be useful to know if $fix\ \alpha.E(\alpha)$ is empty, since this amounts to a proof that f is a non-terminating function (i.e., f's set of possible results is empty). A more general method to determine the results of recursive functions involves the solution of inclusion constraints on regular tree expressions [AM91].

In the sections that follow, we describe a high-level implementation of these operations with considerable attention to efficiency. As a rule, we state results needed to justify our algorithms, but omit the proofs for brevity. Many of these results may be found in the literature [GS84, MR85, HJ90, AW91].

3 Systems of Equations

Regular tree expressions are easy to understand, but are not well suited to the implementation of some algorithms. The algorithms we present are formalized as transformations on sets of equations of the form $\{x_i = Rhs_i\}$, where the x_i are variables and the Rhs_i are regular tree expressions not using the *fix* operator. To discuss the correctness of such transformations, we use the following definition.

Definition 3.1 Let $S = \{x_i = Rhs_i\}$ be a set of equations. We define the following functions on S: $Vars(S)$ is the set of variables in S, $Bound(S)$ is the set of *bound* variables x_i appearing on the left-hand side of equations, and $Free(S)$ is the set of *free* variables $Vars(S) - Bound(S)$.

Throughout this paper, we uniformly use x_i for bound variables, greek letters for free variables, and v for an arbitrary free or bound variable. We also assume that every system of equations contains an equation $x_H =$

S	::=	$\{x_1 = Rhs_1, \ldots, x_n = Rhs_n\}$	$\Psi_S(x_j, \sigma) = \Psi(x_j, \sigma')$ where σ' is the unique substitution such that $\sigma' \uparrow Free(S) = \sigma \uparrow Free(S)$ and, for all $(x_i = Rhs_i) \in S$, $\Psi(x_i, \sigma') = \Psi(Rhs_i, \sigma')$
C	::=	$\{b, c, \ldots\}$	
Rhs	::=	$0 \mid G \wedge T \mid Rhs_1 \vee Rhs_2$	
G	::=	$\alpha_1 \wedge \ldots \wedge \alpha_k$ where $\alpha_i \in Free(S)$	
T	::=	$1 \mid c(y_1, \ldots, y_{a(c)})$ where $y_i \in X, c \in C$	

Figure 2: Syntax and Semantics of Systems of Leaf-Linear Equations

$\vee_{c \in C} c(x_H, ...)$. This allows a concise description of negation (see Section 4). Note that $Rhs_H = 1$.

Definition 3.2 The set of *solutions* $\mathcal{S}(S)$ of S is the set of substitutions for variables in $Vars(S)$ that satisfy the equations:

$$\mathcal{S}(S) = \{\sigma | \Psi(x_i, \sigma) = \Psi(Rhs_i, \sigma)\}$$

Two sets of equations S_1 and S_2 are *equivalent* (written $S_1 \equiv S_2$) if $\mathcal{S}(S_1) = \mathcal{S}(S_2)$. S_1 and S_2 are equivalent over a set of variables Δ if their solutions are the same over those variables:

$$S_1 \equiv_\Delta S_2 \qquad \Leftrightarrow \qquad \{\sigma \uparrow \Delta | \sigma \in \mathcal{S}(S_1)\} = \{\sigma \uparrow \Delta | \sigma \in \mathcal{S}(S_2)\}$$

The inputs and outputs of our algorithms are *leaf-linear systems of equations* [MR85] or *regular* ΣX*-grammars* [GS84]. These are systems of equations with conjunction and disjunction operators syntactically restricted. A syntax and semantics for leaf-linear systems is given in Figure 2. The following theorem shows that the semantics is well-defined [AW91].

Theorem 3.3 Let S be a leaf-linear system. Then for any substitution σ for variables $Free(S)$, there is exactly one substitution σ' for variables $Vars(S)$ such that σ' extends σ and $\sigma' \in \mathcal{S}(S)$.

In other words, each substitution for the free variables determines a substitution for the bound variables. Regular tree expressions and leaf-linear systems are equivalent in a strong sense [MR85]:

Theorem 3.4 For any regular tree expression R there is a leaf-linear system of equations S, and for any leaf-linear system of equations S there is a regular tree expression R such that

$$\forall\sigma\ \Psi(R,\sigma) \quad = \quad \Psi_S(x_1,\sigma)$$

Thus far, we have treated a leaf-linear system as a system of equations; however, a leaf-linear system also can be viewed as a tree automaton [GS84]. We occasionally adopt this view to make use of results from automata theory. When a leaf-linear system is viewed as a tree automaton, an equation represents a state and the transition function for that state. By convention, x_1 is the initial state. The variable on the left-hand side of an equation is the name of the state, and the right-hand side represents the possible transitions. A disjunction corresponds to non-determinism; the automaton may choose one of several transitions. A constructor transition $c(x_1,\ldots,x_{a(c)})$ accepts if the input is of the form $c(t_1,\ldots,t_{a(c)})$ and each x_i accepts t_i. A 1 transition accepts everything; a 0 transition rejects everything. Given some substitution σ for the free variables in a leaf-linear system S, the language $\mathcal{L}(S)$ accepted by a leaf-linear system S is just $\Psi_S(x_1,\sigma)$.

Representing regular tree expressions by leaf-linear systems in an implementation has two significant advantages. First, leaf-linear systems allow sharing among equations, whereas subexpressions are not shared in regular tree expressions. This is obviously more economical. Second, two of the most important algorithms, testing emptiness and containment, are more easily expressed and efficiently implemented using leaf-linear systems. The potential disadvantage of using leaf-linear systems is that they may be, in the worst case, exponentially larger than the smallest equivalent regular tree expression. We have found that, in practice, all of this potential size explosion can be avoided by using the algorithms and optimizations presented here.

As an example, the expression *fix* $\alpha.c(1,\alpha)\vee b$ (the set of Lisp lists) is represented by the leaf-linear system:

$$\begin{aligned} x_1 &= c(x_2,x_1)\vee b \\ x_2 &= 1 \end{aligned}$$

Our system implements leaf-linear systems almost exactly as described here. A system of equations is represented by an array. Free and bound variables are indices into the array, and right-hand sides of equations are the

entries in the array. There are four types of right-hand sides: one each for 0, 1, free variables, and disjunctions of constructor expressions. As operations are performed, new equations may be added, extending the array, and free variables may be instantiated, in which case the corresponding index of the array acquires a new entry.

4 Testing Emptiness

One fundamental problem is determining whether an expression denotes the empty set under all substitutions for free variables. More formally, for a given leaf-linear system S, we wish to test the predicate $\forall \sigma \in \mathcal{S}(S)\ \Psi_S(E, \sigma) = \emptyset$. Testing emptiness is useful for two reasons. First, it can be an important part of the application. For example, in our type inference algorithm for FL, proving that a function is type-safe is reduced to proving that a regular tree expression is empty [AM91]. Second, emptiness testing is important for the efficiency of the system. It is wasteful, both of time and space, to build and maintain expressions of the form $c(Y, X)$ or $Y \wedge X$ where Y is empty. The following easy lemma shows that the problem of testing emptiness in all substitutions reduces to the problem of testing emptiness in one particular substitution.

Lemma 4.1 Let τ be the substitution $\forall v\ \tau(v) = H$. Then

$$\forall \sigma\ \Psi_S(E, \sigma) = \emptyset \qquad \Leftrightarrow \qquad \Psi_S(E, \tau) = 0$$

To test whether $\Psi_S(x_i, \tau) = \emptyset$, we use the function Φ_S defined in Figure 3. This definition is a straightforward adaptation of algorithms for reachability and emptiness for finite automata [HU79, GS84].

Lemma 4.2 Let $S = \{x_1 = Rhs_1, \ldots, x_k = Rhs_k\}$ be a leaf-linear system, and let Φ_S be defined as in Figure 3. Then

$$\Phi_S(x_i) = \perp \qquad \Leftrightarrow \qquad \Psi_S(x_i, \tau) = \emptyset$$

Proof: [sketch] It is easy to show by induction on the height of terms t that

$$\Phi_S(x_i) = \perp \qquad \Leftrightarrow \qquad \forall t\ height(t) \leq k \Rightarrow t \notin \Psi_S(x_i, \tau)$$

Φ_S is the least function (under the ordering $\bot \leq \top$) s.t.

$$\begin{aligned}
\Phi_S(x_i) &= \Phi_S(Rhs_i) \\
\Phi_S(1) &= \top \\
\Phi_S(0) &= \bot \\
\Phi_S(c(E_1, \ldots, E_n)) &= \begin{cases} \top & \text{if } \forall i\ \Phi_S(E_i) = \top \\ \bot & \text{otherwise} \end{cases} \\
\Phi_S(E_1 \wedge E_2) &= \begin{cases} \top & \text{if } \Phi_S(E_1) = \top \text{ and } \Phi_S(E_2) = \top \\ \bot & \text{otherwise} \end{cases} \\
\Phi_S(E_1 \vee E_2) &= \begin{cases} \top & \text{if } \Phi_S(E_1) = \top \text{ or } \Phi_S(E_2) = \top \\ \bot & \text{otherwise} \end{cases} \\
\Phi_S(\alpha) &= \top \text{ if } \alpha \in \mathit{Free}(S)
\end{aligned}$$

Figure 3: Testing for emptiness

To finish the proof, we observe that the language of a tree automaton of k states is non-empty if and only if it accepts a term of height at most k [GS84]. □

Let n be the total number of symbols appearing in a leaf-linear system S. The function Φ_S is computable for every equation in the system in $\mathcal{O}(n^2)$ time using a standard fixed-point computation. Initially, $\Phi_S(x_i)$ is assumed to be $\bot$ for all x_i. Iteratively updating Φ_S requires at most $\mathcal{O}(n)$ passes to compute a fixed point, and each iteration requires examining $\mathcal{O}(n)$ right-hand side symbols.

In our application, we found that up to a third of expressions turn out to be empty. Consequently, our system maintains the following invariant:

Invariant 4.3 Let $S = \{x_i = Rhs_i\}$, and let E be any subexpression appearing in a Rhs_i.

$$\forall \sigma \in \mathcal{S}(S)\ \Psi_S(E, \sigma) = \emptyset \qquad \Rightarrow \qquad (E = 0)$$

Thus if x_i is empty in all solutions, then $Rhs_i = 0$. This makes testing whether an equation is empty very cheap; it is empty if its right-hand side is 0. In the rest of this section, we describe the incremental algorithm used to maintain this invariant.

$$
\begin{array}{rcl}
E_1 \wedge E_2 & = & E_2 \wedge E_1 \\
E_1 \vee E_2 & = & E_2 \vee E_1 \\
E \wedge 0 & \Rightarrow & 0 \\
E \wedge 1 & \Rightarrow & E \\
E \wedge E & \Rightarrow & E \\
E \vee 0 & \Rightarrow & E \\
E \vee 1 & \Rightarrow & 1 \\
E \vee E & \Rightarrow & E \\
c(E_1, \ldots, E_{a(c)}) \wedge c(E'_1, \ldots, E'_{a(c)}) & \Rightarrow & c(E_1 \wedge E'_1, \ldots, E_{a(c)} \wedge E'_{a(c)}) \\
c(\ldots) \wedge d(\ldots) & \Rightarrow & 0 \text{ if } c \neq d \\
c(\ldots, 0, \ldots) & \Rightarrow & 0 \\
\neg 0 & \Rightarrow & 1 \\
\neg 1 & \Rightarrow & 0 \\
\neg\neg E & \Rightarrow & E \\
\neg(E_1 \wedge E_2) & \Rightarrow & \neg E_1 \vee \neg E_2 \\
\neg(E_1 \vee E_2) & \Rightarrow & \neg E_1 \wedge \neg E_2 \\
\neg(c(E_1, \ldots, E_{a(c)})) & \Rightarrow & \bigvee_{d \in C - \{c\}} d(x_H, \ldots) \\
 & & \vee \bigvee_{1 \leq i \leq a(c)} c(\ldots, x_H, \neg E_i, x_H, \ldots)
\end{array}
$$

Figure 4: Simplifying expressions.

The function Φ_S is the basis for the first transformation on sets of equations.

$$S \cup \{x_i = Rhs_i\} \quad \equiv \quad S \cup \{x_i = 0\} \qquad \text{if } \Phi_S(x_i) = \bot \tag{1}$$

This transformation enforces Invariant 4.3 for entire right-hand sides. To enforce the invariant for every expression, we introduce a group of equivalences given in Figure 4. These are obvious simplifications, and, when regarded as rewrite rules from left to right, form a confluent rewrite system that is noetherian up to the commutativity of $\wedge$ and $\vee$. For an expression E, $Simp(E)$ is the normalization of expressions under the rules of Figure 4. The

$$
\begin{aligned}
\mathit{Vars}(E) &= \mathit{Free}(E) \cup \mathit{Bound}(E) \\
\mathit{Users}(v) &= \{x_i | v \in \mathit{Vars}(\mathit{Rhs}_i)\} \\
\mathit{AllUsers}(v) &= \text{least } V = \mathit{Users}(v) \cup \{x_i | x_i \in \mathit{Users}(v'), v' \in V\}
\end{aligned}
$$

Figure 5: Simple functions on leaf-linear systems.

complexity of computing $\mathit{Simp}(E)$ is potentially exponential in the size of E due to the rules for intersection and negation; however, in practice right-hand sides of equations are quite small, so this has not been a problem in our implementation. Using these simplifications, the rest of Invariant 4.3 is enforced by the following transformation.

$$S \cup \{x_i = 0, x_j = E(x_i)\} \quad \equiv \quad S \cup \{x_i = 0, x_j = \mathit{Simp}(E(0))\} \qquad (2)$$

For efficiency, we would like to locate the equation x_j in Rule (2) quickly. The set $\mathit{Users}(x)$, defined in Figure 5, is the set of all equations that mention x on the right-hand side. Our system incrementally maintains $\mathit{Users}(x)$ for each equation, thus allowing candidates for the application of Rule (2) to be located in constant time.

Lemma 4.4 Let S' be any leaf-linear system closed under application of Rules (1) and (2). Then S' satisfies Invariant 4.3.

As an example, consider the following system.

$$
\begin{aligned}
x_1 &= d(x_2, x_2) \\
x_2 &= c(x_2)
\end{aligned}
$$

Both x_1 and x_2 are empty in all substitutions. Rule (1) sets x_2 to 0, then Rule (2) sets x_1 to 0.

Given any leaf-linear system, we can apply Rules (1) and (2) to produce a leaf-linear system that satisfies the invariant. However, a more practical situation is that a leaf-linear system S satisfying the invariant is modified in one equation to produce a slightly different system S', and then we wish to enforce the invariant for S'.

We briefly describe an incremental version of this algorithm that solves this problem efficiently. Let $x_i = Rhs_i$ be the equation of S that is modified to produce S'. The meaning of an equation can change as a result of this modification only if it depends, directly or indirectly, on x_i. The function *AllUsers*, defined in Figure 5, captures the set of equations that depend on a particular variable. A straightforward restriction of the function Φ_S to $AllUsers(x_i)$ is all that is required. We add one new clause to the definition in Figure 3:

$$\Phi_S(x_j) = \top \text{ if } x_j \notin \{x_i\} \cup AllUsers(x_i)$$

In practice, an incremental algorithm is important, because the size of $AllUsers(x_i)$ is small (typically tens) compared to the number of equations in the system (typically thousands). Our implementation computes *AllUsers* using the obvious $\mathcal{O}(n^3)$ fixed-point computation.

5 Construction, Or, And, Fix, and Not

In this section we show how to perform the operations given in Section 2 on leaf-linear systems. Section 7 covers the optimization of these algorithms. In the following, $E_1, \ldots, E_n$ are regular tree expressions, S is a leaf-linear system where

$$\forall \sigma \ \Psi_S(x_i, \sigma) = \Psi(E_i, \sigma)$$

and x is a fresh variable. For each operation $f(E_1, \ldots, E_n)$, we show how to extend S to a leaf-linear system S' with an equation $x = E'$ such that

$$\forall \sigma \ \Psi_S(x, \sigma) = \Psi(f(E_1, \ldots, E_n), \sigma)$$

For each case, unless otherwise noted, it is necessary to enforce Invariant 4.3 for all new and modified equations. We begin with the base cases and constructor expressions. If E_1 is a free variable α, we define $S' = S \cup \{x = \alpha \wedge 1\}$. If E_1 is 0 or 1, we define $S' = S \cup \{x = 0 \text{ or } 1\}$.

5.1 Constructors

Constructors are the easiest operations to implement using leaf-linear systems. Consider a regular tree expression $c(E_1, \ldots, E_{a(c)})$. We define a leaf-linear system $S' = S \cup \{x = c(x_1, \ldots, x_n)\}$. This is a constant time operation.

5.2 Or

Union is also a constant-time operation on leaf-linear systems. Consider a regular tree expression $E_1 \vee E_2$. We define a new leaf-linear system S' as follows:

$$S' = S \cup \{x = Rhs_1 \vee Rhs_2\}$$

Clearly $x = x_1 \vee x_2$, as desired. Furthermore, S' is in leaf-linear form, because the disjunction of right-hand sides is still a valid right-hand side. Finally, if S satisfies Invariant 4.3, then S' satisfies Invariant 4.3 without any additional work.

5.3 And

Intersection is easy if we are not concerned with efficiency. Consider a regular tree expression $E_1 \wedge E_2$. To build a leaf-linear system with an equation representing $x_1 \wedge x_2$, we may add all equations of the form

$$x_i \wedge x_j = Simp(Rhs_i \wedge Rhs_j)$$

and then replace conjunctions of bound variables $x_i \wedge x_j$ by new variable names [MR85]. This is wasteful, however, because many of these equations may not be required to express the desired intersection, and this algorithm always uses $\Theta(n^2)$ time and space. Thus a series of only m intersections of systems of n equations consumes $\Theta(n^m)$ time and space.

While we cannot improve on the worst-case time and space complexity, a different algorithm does much better in the typical case. The idea is to generate only those conjunctions of bound variables that are needed to express the result. We define a system S' as follows:

$$S' = S \cup \{x = Simp(Rhs_1 \wedge Rhs_2)\}$$

S' is not necessarily a leaf-linear system, but only a limited violation of leaf-linearity can occur in S'.

Lemma 5.1 Let Rhs_i and Rhs_j be right-hand sides in leaf-linear form. Then, if $Simp(Rhs_i \wedge Rhs_j)$ is not in leaf-linear form, it contains subexpressions of the form $c(\ldots, x_h \wedge x_k, \ldots)$.

The only way $Simp(Rhs_i \wedge Rhs_j)$ can fail to be leaf-linear is if it contains conjunctions of pairs of bound variables inside of constructors. We use two transformations to eliminate the conjunctions of bound variables. These transformations use an auxiliary set of equations A.

$$S \cup \{x_i = E(x_j \wedge x_k)\} \cup A \equiv_{Vars(S)} \begin{array}{l} S \cup \{x_i = E(x'), x' = Simp(Rhs_j \wedge Rhs_k)\} \\ \quad \cup A \cup \{x' = x_j \wedge x_k\},\ x' \text{ new} \end{array} \quad (3)$$

$$\begin{array}{l} S \cup \{x_i = E(x_j \wedge x_k)\} \\ \quad \cup A \cup \{x' = x_j \wedge x_k\} \end{array} \equiv S \cup \{x_i = E(x')\} \cup A \cup \{x' = x_j \wedge x_k\} \quad (4)$$

To transform S' into leaf-linear form, we repeatedly apply Rules (3) and (4) to equations that are not leaf-linear. For efficiency (and to guarantee termination) Rule (4) is always chosen in preference to (3) when both apply. It is easy to prove by induction on the number of transformations performed by this algorithm that if $x_i \wedge x_j$ is a conjunction of bound variables introduced by a transformation, then $x_i = Rhs_i$ and $x_j = Rhs_j$ are equations in S'. Thus, this algorithm terminates because there are only $\mathcal{O}(|Vars(S)|^2)$ possible conjunctions of bound variables. Finally, we can drop the set A of auxiliary equations.

Lemma 5.2 Suppose a leaf-linear system $S' \cup A$ is obtained by application of Rules (3) and (4) to $S \cup \emptyset$, as directed above. Then $S \cup \emptyset \equiv_{Vars(S)} S' \cup A$ and $S' \equiv_{Vars(S)} S' \cup A$.

5.4 Fix

This case is easy enough that the solution can be given directly, without additional rules. Consider a regular tree expression $fix\ \alpha.E_1$, and let S be a leaf linear system with free variable α such that

$$\forall \sigma\ \Psi_S(x_1, \sigma) = \Psi(E_1, \sigma)$$

We define a new leaf-linear system

$$\begin{array}{rcl} S' & = & S \cup \{\alpha = Least(\alpha, Rhs_1)\} \text{ where} \\ & & Least(\alpha, Rhs) = \left\{ \begin{array}{ll} Least(\alpha, W) & \text{if } Rhs = W \vee (\alpha \wedge Y) \text{ for } Y \neq 0 \\ Rhs & \text{otherwise} \end{array} \right. \end{array}$$

The system S' differs from S in that α becomes a bound variable and there is one new equation. The function *Least* is needed because $\alpha = W \vee (\alpha \wedge Y)$ has (potentially) many solutions for every substitution of the free variables $\mathit{Free}(S) - \{\alpha\}$; however, because *fix* is a least fixed-point operator, we are interested only in the least solution. The following lemma makes this precise.

Lemma 5.3 Consider a set of equations S with free variable α. Given a substitution π for $\mathit{Free}(S) - \alpha$, let σ be the least substitution that extends π to $\mathit{Vars}(S)$ such that $\sigma \in \mathcal{S}(S \cup \{\alpha = \mathit{Rhs}_1\})$. Then $\sigma \in \mathcal{S}(S \cup \{\alpha = \mathit{Least}(\alpha, \mathit{Rhs}_1)\})$.

The equation $\alpha = \mathit{Least}(\alpha, \mathit{Rhs}_1)$ is in leaf-linear form, because *Least* removes any occurrences of the newly bound variable from a guard in the right-hand side of the equation. Other equations in S' may not be in leaf-linear form because they contain α in intersections. The algorithm for intersection is used to restore leaf-linear form to the equations in $\mathit{Users}(\alpha)$. As an aside, a greatest fixed point operator is obtained by using

$$\mathit{Grtst}(\alpha, \mathit{Rhs}) = \begin{cases} \mathit{Grtst}(\alpha, W) \vee \mathit{Grtst}(\alpha, Y) & \text{if } \mathit{Rhs} = W \vee (\alpha \wedge Y) \text{ for } Y \neq 0 \\ \mathit{Rhs} & \text{otherwise} \end{cases}$$

in place of *Least*.

5.5 Negation

Let S be a leaf-linear system for the regular tree expression E_1, which contains no free variables. A system S' for $\neg E_1$ is

$$S' = S \cup \{x = \mathit{Simp}(\neg \mathit{Rhs}_1)\}$$

The approach used to put S' into leaf-linear form is very similar to the algorithm for intersection. The transformations are:

$$S \cup \{x_i = E(\neg x_j)\} \cup A \equiv_{\mathit{Vars}(S)} S \cup \{x_i = E(x'), x' = \mathit{Simp}(\neg \mathit{Rhs}_j)\} \cup A \cup \{\neg x_j = x'\} \quad x' \text{ new} \tag{5}$$

$$S \cup \{x_i = E(\neg x_j)\} \cup A \cup \{\neg x_j = x'\} \equiv S \cup \{x_i = E(x')\} \cup A \cup \{\neg x_j = x'\} \tag{6}$$

As with the algorithm for intersection, these transformations are iterated until neither applies, and then the set of auxiliary equations is dropped.

However, the resulting system may not be leaf-linear, because these transformations may introduce intersections of bound variables. These are eliminated as before using the algorithm for intersection.

Computing a leaf linear system for $\neg E$ may require time and space exponential in the size of E, and we have found that this does, occasionally, become a problem in practice. To compensate, we allow the computed negation to be either a subset or superset of the exact result, depending on which direction is conservative for the context in which the result is used. The heuristic we use is bounding the depth to which negations are computed; beyond this fixed depth k, the result is either 0 or 1, depending on the context. The following revised rules express the idea.

$$
\begin{array}{lcl}
S \cup \{x_i = E(\neg_0^+ x_j)\} \cup A & \Rightarrow & S \cup \{x_i = \mathit{Simp}(E(1))\} \\
S \cup \{x_i = E(\neg_0^- x_j)\} \cup A & \Rightarrow & S \cup \{x_i = \mathit{Simp}(E(0))\} \\
S \cup \{x_i = E(\neg_k^d x_j)\} \cup A & \equiv_{\mathit{Vars}(S)} & S \cup \{x_i = E(x'), x' = \mathit{Simp}(\neg_{k-1}^d \mathit{Rhs}_j)\} \\
 & & \quad \cup A \cup \{x' = \neg x_j\}\ x' \text{ new, if } k > 0 \\
S \cup \{x_i = E(\neg_k^d x_j)\} & \equiv & S \cup \{x_i = E(x')\} \cup A \cup \{x' = \neg x_j\} \text{ if } k > 0 \\
\quad \cup A \cup \{x' = \neg x_j\} & &
\end{array}
$$

Note that it is necessary to discard the set of assumptions A in the first two transformations, because when the approximation rules are used the constraints in A may no longer hold.

6 Testing Inclusion

Given a set of equations S and two expressions E_1 and E_2, we often wish to test the predicates $\forall\sigma\ \Psi_S(E_1, \sigma) \subseteq \Psi_S(E_2, \sigma)$ and $\exists\sigma\ \Psi_S(E_1, \sigma) \subseteq \Psi_S(E_2, \sigma)$. Performing these tests is critical in our application. The first predicate arises when type analysis proves that a function is always applied to an argument in its appropriate domain (i.e., that the actual arguments are a subset of the appropriate domain). The second predicate arises in analyzing recursive functions, when it becomes necessary to solve constraints to assign types to recursive functions [AM91] (in this case, it is necessary to actually compute a substitution σ that satisfies the constraints). A fast algorithm for containment has a third application: it can dramatically increase the performance

of the other regular tree operations (see Section 7). Given the importance of testing inclusion, the following result is discouraging.

Theorem 6.1 Evaluating the predicates

$$\forall\sigma\ \Psi_S(E_1,\sigma) \subseteq \Psi_S(E_2,\sigma)$$

$$\exists\sigma\ \Psi_S(E_1,\sigma) \subseteq \Psi_S(E_2,\sigma)$$

is exponential-time hard [AW91].

For the first predicate, $\forall\sigma\ \Psi_S(E_1,\sigma) \subseteq \Psi_S(E_2,\sigma)$, a decision procedure is known with the restrictions upon negation used here [Mur90]. This algorithm proved impractical in an implementation. The predicate $\exists\sigma\ \Psi_S(E_1,\sigma) \subseteq \Psi_S(E_2,\sigma)$ is computable in general, and in fact it is possible to compute all substitutions that make the inclusion relationship true [AW91]. Unfortunately, this algorithm runs in non-deterministic exponential time.

In this section, we present a single mechanism implementing a conservative heuristic for both inclusion tests. As we discuss below, this heuristic has worked very well in our implementation. The heuristic is formalized using a logic with theorems of the form $A \vdash E_1 \subseteq E_2$. The set A contains constraints on the free variables that make the inclusion relationship true. The predicate $\forall\sigma\ \Psi_S(E_1,\sigma) \subseteq \Psi(E_2,\sigma)$ is reduced to the question $\emptyset \vdash E_1 \subseteq E_2$, while the $\exists\sigma\ \Psi_S(E_1,\sigma) \subseteq \Psi_S(E_2,\sigma)$ predicate reduces to finding any A such that $A \vdash E_1 \subseteq E_2$. The logic is not complete; that is, there may be no proof $A \vdash E_1 \subseteq E_2$, even if there exists a substitution for which the inclusion relationship holds.

The axioms and inference rules for proving inclusion relationships are given in Figure 6. Several of the rules in Figure 6 could be combined to give a more concise system; however, because we are presenting an implementation, we prefer to describe the cases that are actually handled by the algorithm. To help explain these rules, we make the following definition. For a set of constraints A, $F(A)$ is the subset of A consisting of constraints of the form $\alpha \subseteq E$ or $E \subseteq \alpha$. Note that the only other possible constraints are between bound variables (rule [BASSUME]).

Our implementation uses a proof procedure based on these rules. This proof procedure is goal-oriented; it begins with a fact to prove, and runs the inference rules backward to axioms, building up the needed assumptions as

$\dfrac{}{\emptyset \vdash E \subseteq 1}$	[ONE]	$\dfrac{}{\emptyset \vdash 0 \subseteq E}$	[ZERO]
$\dfrac{A \vdash Rhs_H \subseteq E}{A \vdash 1 \subseteq E}$	[LONE]	$\dfrac{}{\emptyset \vdash E \subseteq E}$	[TAUT]
$\dfrac{}{\{x_1 \subseteq x_2\} \vdash x_1 \subseteq x_2}$	[BASSUME]	$\dfrac{A \cup \{x_i \subseteq x_j\} \vdash Rhs_i \subseteq Rhs_j}{A \vdash x_i \subseteq x_j}$	[REC]
$\dfrac{}{\{\alpha \subseteq E_2\} \vdash \alpha \subseteq E_2}$	[VASSUM1]	$\dfrac{}{\{E_1 \subseteq \alpha\} \vdash E_1 \subseteq \alpha}$	[VASSUM2]
$\dfrac{}{\emptyset \vdash \alpha \wedge E \subseteq \alpha}$	[LVAR]	$\dfrac{A \vdash \alpha \subseteq E}{A \vdash \alpha \subseteq \alpha \wedge E}$	[RVAR]
$\dfrac{\forall i\ A_i \vdash E \subseteq E_i}{\bigcup_i A \vdash E \subseteq \bigwedge_i E_i}$	[AND]	$\dfrac{A \vdash \alpha \subseteq E_1 \vee E_3,\ \Psi_S(E_3, \sigma) = H - \Psi_S(E_2, \tau)}{A \vdash \alpha \wedge E_2 \subseteq E_1}$	[VAR]
$\dfrac{\forall i\ A_i \vdash x_{1i} \subseteq x_{2i}}{\bigcup_i A_i \vdash\ c(\ldots, x_{1i}, \ldots) \subseteq c(\ldots, x_{2i}, \ldots)}$	[CONS]	$\dfrac{\exists i\ A \vdash x_i \subseteq 0}{A \vdash c(\ldots, x_i, \ldots) \subseteq 0}$	[ZCONS]
$\dfrac{\forall i\ A_i \vdash E_i \subseteq E}{\bigcup_i A_i \vdash \bigvee_i E_i \subseteq E}$	[LOR]	$\dfrac{\exists i\ A \vdash E \subseteq E_i}{A \vdash E \subseteq \bigvee_i E_i}$	[ROR]

Figure 6: Inference rules for testing inclusion.

$$\frac{}{\{\alpha \subseteq \gamma\} \vdash \alpha \subseteq \gamma}\ [\text{VASSUM1}] \qquad \frac{}{\{x_1 \subseteq x_2\} \vdash x_1 \subseteq x_2}\ [\text{BASSUME}]$$

$$\frac{\{\alpha \subseteq \gamma, x_1 \subseteq x_2\} \vdash c(\alpha, x_1) \subseteq c(\gamma, x_2)}{\{\alpha \subseteq \gamma, x_1 \subseteq x_2\} \vdash c(\alpha, x_1) \subseteq c(\gamma, x_2) \vee b}\ [\text{ROR}] \qquad \frac{}{\emptyset \vdash b \subseteq b}\ [\text{CONS}]$$

$$\frac{}{\emptyset \vdash b \subseteq c(\gamma, x_2) \vee b}\ [\text{ROR}]$$

$$\frac{\{\alpha \subseteq \gamma, x_1 \subseteq x_2\} \vdash c(\alpha, x_2) \vee b \subseteq c(\gamma, x_2) \vee b}{\{\alpha \subseteq \gamma\} \vdash x_1 \subseteq x_2}\ [\text{LOR}]\ [\text{REC}]$$

$$\begin{aligned} x_1 &= c(\alpha, x_1) \vee b \\ x_2 &= c(\gamma, x_2) \vee b \end{aligned}$$

Figure 7: An inclusion example.

it goes. Figure 7 gives an example of a simple proof derived by our proof procedure using the logic.

The rules in Figure 6 almost define a deterministic proof procedure. To eliminate non-determinism, axioms [TAUT], [ONE], and [ZERO] are always applied in preference to all other rules. An assumption on bound variables is introduced by [BASSUME] if and only if there is a [REC] step to eliminate the assumption; this constraint guarantees that the conclusion of a proof has the form $A \vdash x_i \subseteq x_j$ where A contains assumptions only on free variables (i.e., $F(A) = A$).

When both sides of an inclusion are disjunctions, [LOR] is applied to break up the left-hand side before [ROR] is used. This is the order used in Figure 7. Inference rules [LVAR] or [RVAR] are used, if applicable, before [VASSUM1] or [VASSUM2]. This guarantees that in a proof there are no assumptions of the form $\alpha \subseteq \alpha \wedge E$ or $\alpha \wedge E \subseteq \alpha$ in $F(A)$. Finally, the last source of non-determinism is the order in which possibilities are considered in [ZCONS] and [ROR]; in our implementation, this order is fixed to be from left to right.

6.1 Computing Substitutions from Constraints

A proof of $A \vdash x_i \subseteq x_j$ does not necessarily yield a substitution that makes the relationship $x_i \subseteq x_j$ true. For example, it is entirely possible to have a proof

$$\{\alpha \subseteq 0, 1 \subseteq \alpha\} \vdash x_i \subseteq x_j$$

In this case, the set of constraints implies (by transitivity) that $1 \subseteq 0$; that is, the set of constraints is inconsistent. The following definition identifies the sets of constraints that do yield substitutions.

Definition 6.2 A set of constraints A is *closed* if

$$\{X \subseteq \alpha, \alpha \subseteq Y\} \subseteq A \Rightarrow A \vdash X \subseteq Y$$

Let $A \vdash x_i \subseteq x_j$. If A is not closed, choose constraints $\{X \subseteq \alpha, \alpha \subseteq Y\} \subseteq A$ that do not satisfy Definition 6.2 and and find a proof $A' \vdash X \subseteq Y$. Repeat this procedure on $A \cup A'$ until the set is closed, or an inconsistency is discovered. This process terminates because constraints are built only from existing expressions and expressions introduced by [VAR]. An easy calculation shows that the total number of such expressions is finite, and thus so is the set of possible constraints. The set of constraints in Figure 7 is closed.

A closed set of constraints is simplified using the rules in Figure 8 so that there is at most one upper and one lower bound per free variable. We also add trivial constraints $0 \subseteq \alpha$ and $\alpha \subseteq 1$ to guarantee that there is exactly one upper and lower bound per free variable. In Figure 7, there is only one constraint; adding the trivial constraints yields the constraints $\{0 \subseteq \alpha \subseteq \gamma, \alpha \subseteq \gamma \subseteq 1\}$. In the case where constraints are between free variables only, our system performs a small optimization and does not add trivial constraints for both variables. In this case, the set of constraints produced by our system is $\{\alpha \subseteq \gamma \subseteq 1\}$.

Theorem 6.3 Let $A \vdash x_i \subseteq x_j$ be a proof where A is closed and

$$A = \{L_i \subseteq \alpha_i \subseteq U_i | \alpha_i \textit{ free in } S\}$$

Let $\beta_1, \ldots, \beta_n$ be fresh variables, and let S' be the set of equations S extended with the additional equations (for each α_i)

$$\alpha_i = L_i \vee (\beta_i \wedge U_i)$$

Then $\forall \sigma\ \Psi_{S'}(x_i, \sigma) \subseteq \Psi_{S'}(x_j, \sigma)$.

$$
\begin{array}{rcl}
A \cup \{\alpha \subseteq E_1, \alpha \subseteq E_2\} & \equiv & A \cup \{\alpha \subseteq E_1 \wedge E_2\} \\
A \cup \{E_1 \subseteq \alpha, E_2 \subseteq \alpha\} & \equiv & A \cup \{E_1 \vee E_2 \subseteq \alpha\}
\end{array}
$$

Figure 8: Simplifying sets of constraints.

The proof of this theorem is difficult; see [AW91]. The intuition behind the construction is that the free variable β_i allows the actual value of α_i to be anything "in between" the lower and upper bounds L_i and U_i. Referring again to Figure 7, using the constraints $\{\alpha \subseteq \gamma \subseteq 1\}$ our system produces the system of equations

$$
\begin{array}{rcl}
x_1 & = & c(\alpha, x_1) \vee b \\
x_2 & = & c(\gamma, x_2) \vee b \\
\gamma & = & \alpha \vee \beta
\end{array}
$$

6.2 Discussion

For the most part, the rules in Figure 6 have the property that the conclusion holds if and only if the hypotheses hold. The two exceptions are the inference rules [VAR] and [ROR]. These two rules are the "heuristics" in our proof procedure. We explain the rational behind each below.

The rule [VAR] is an approximation of the fact $\alpha \wedge E_2 \subseteq E_1 \Leftrightarrow \alpha \subseteq E_1 \vee \neg E_2$. The problem is that $\neg E_2$ may not be expressible in our language—that is, it may introduce negations on free variables. As discussed in Section 2, we do not permit this because negation is not monotonic, and therefore admitting negations at this point makes it impossible to define a least fixed-point operator. If E_2 does not depend on free variables, then this rule is precise; the effect of [VAR] is to use the best approximation of E_2 that does not depend on free variables. In our system, when [VAR] is used E_2 rarely depends on free variables; thus, this heuristic appears to have little practical effect on our system.

Rule [ROR] states that to prove $C \subseteq D \vee E$, prove either $C \subseteq D$ or $C \subseteq E$. Many facts are not provable with this rule, such as

$$c(a \vee b) \subseteq c(a) \vee c(b)$$

In our application, [ROR] appears to be more than adequate. While we cannot completely explain this, one reason is that our system optimizes disjunctions $c(a) \vee c(b)$ into the equivalent $c(a \vee b)$. Similarly, $d(a,x) \vee d(b,x)$ becomes $d(a \vee b, x)$. Apparently this suffices to cover common cases missed by the [ROR] rule; our type analysis system builds very complex equations, and yet we have not found a practical example where this approximation fails to be accurate.

It is worth pointing out that there are much stronger rules than [ROR]. Consider the following lemma.

Lemma 6.4 Let S be a subset of $\{1,\ldots,n\}$ and let $\overline{S}$ be $\{1,\ldots,n\} - S$.

$$c(x,y) \subseteq \bigvee_{1 \le i \le n} c(x_i, y_i) \qquad \Leftrightarrow \qquad \forall S \; (x \subseteq \bigvee_{j \in S} x_j) \vee (y \subseteq \bigvee_{j \in \overline{S}} y_j)$$

See [Mur90] for a generalization to constructors of arbitrary arity. The problem with an inference rule based on Lemma 6.4 is that it consumes exponential time and space, since it generates exponentially many new regular tree expressions. For an implementation, the rules in Figure 6 have the advantage that, except for sets of assumptions, they consume no space.

7 Optimizations

The implementation we have described thus far is complete but still performs poorly in practice. This section covers the remaining performance problems and the optimizations that overcome them. In order to make the effectiveness of the optimizations clear and concrete, we use an example from our type inference system for FL. In the FL type inference algorithm, types are represented by regular tree expressions. In the process of analyzing a program, the type inference algorithm performs many (typically thousands) of regular tree operations. The FL type system is implemented on an IBM RT/PC in Lucid Common Lisp 3.0.

The example we use is a heapsort program written in FL. This program was written to exercise the type system, especially its ability to solve systems of constraints arising from recursively defined higher-order functions. The type system generates large and complex types while analyzing heapsort. The text of the heapsort program is about 60 lines of FL; after parsing and

abbreviation elimination, the program that the type system actually analyzes has about 100 lines.

Using the algorithms described so far, the type system analyzes heapsort for about five minutes before the Lisp system crashes with a stack overflow. Just before it dies, the system has generated 12,948 equations, including one with a disjunction of 1040 constructors on the right-hand side! Inspection shows that the intersection algorithm is generating an enormous number of equations. Before the crash, the system computes 1407 intersections, which require a total of 84,409 applications of Rules (3) and (4), for an average of 60 rules per intersection computed. The cost of 60 rules per intersection is misleading—this cost grows rapidly during the computation and would presumably far exceed 60 rules per intersection if the Lisp system had more stack space. Although the cost of intersections dominates in this example, the inclusion and negations algorithms are also slow.

Analyzing the actual sequence of intersections, negations, and inclusion tests performed reveals part of the problem: many of the operations are being computed over and over again. *Memoization* is a simple optimization that caches and reuses the results of computations [Mic68]. We have already used something quite like memoization in the auxiliary systems of equations used for computing intersection and negation in Section 5. If fact, instead of discarding those auxiliary equations, they can be retained and reused if the same computations are performed again.

We formalize memos as a set of auxiliary equations. For regular tree operations, every memo is an equation of the form $x = f(x_1, \ldots, x_n)$ for a regular tree operation f. Whenever one of these operations is computed, the set of auxiliary equations is searched to see if the answer is already known. If so, the left-hand side of the equation is used instead of performing the computation again. If an auxiliary equation is not found, then the computation is performed and the result is recorded in the auxiliary equations. These memos cost only constant space and time; every memo is of constant size and corresponds to some result that must be computed anyway. Furthermore, lookup can be done in constant time if the auxiliary equations are implemented as a hash table.

Memos for inclusion are similar, but instead of auxiliary equations the memos are auxiliary constraints. Whenever the system proves a fact $\emptyset \vdash x_1 \subseteq x_2$, a constraint $x_1 \subseteq x_2$ is added to the system. During inclusion tests, if the subgoal $x_1 \subseteq x_2$ is in the list of auxiliary constraints, then that subgoal

is discharged. Inclusion memos are more expensive than memos for regular tree operations. While the added time to search for a memo is still effectively constant with a hashtable implementation, the space consumed is potentially $\mathcal{O}(n^2)$ for a system of n equations.

With memoization of all regular tree operations and inclusion tests, the system performs somewhat better on heapsort. In this trial, the system nearly fills the Lisp memory and begins to thrash after running for two hours. Stopping the computation at this point, the system has 36,194 equations. Again, the main culprit is intersection, with an average of 45 rules per computation. The remaining problem is that many trivial operations add new equations. For example, to compute $E \wedge 1$, the system generates a new equation instead of just using the existing equation for E. Generalizing, if $E_1 \subseteq E_2$, then the result of computing $E_1 \wedge E_2$ should be just the equation for E_1, thus avoiding a possibly large number of redundant equations produced by using the intersection algorithm to compute $E_1 \wedge E_2$. We add the following rules to the intersection algorithm, which are applied in preference to all other rules.

$$\begin{aligned} S \cup \{x = E(x_i \wedge x_j)\} &\equiv S \cup \{x = E(x_j)\} \text{ if } \emptyset \vdash x_i \subseteq x_j \\ S \cup \{x = E(x_i \wedge x_j)\} &\equiv S \cup \{x = E(x_i)\} \text{ if } \emptyset \vdash x_j \subseteq x_i \end{aligned}$$

Similarly, to compute the union $E_1 \vee E_2$, if $E_1 \subseteq E_2$ then the equation for E_2 is used, and if $E_2 \subseteq E_1$ then the equation for E_1 is used. If neither case applies, the equation given in Section 5.2 is added to the system.

Combining memoization and the inclusion optimizations results in a dramatic improvement. Using these optimizations, the system is able to analyze the heapsort program in just under two minutes. The system generates 5081 equations in this trial; the most complex equation has three constructors on the right-hand side.

The first half of the table in Figure 9 gives the number of operations, the number of steps, and the steps per operation for each of inclusion, intersection, and negation in this trial. For negation and intersection, the number of steps is the total number of equations added and successful memo lookups. For inclusion, it is the number of subgoals of the form $x_1 \subseteq x_2$ in proofs and the number of successful memo lookups. Thus, the measure for inclusion counts the total number of pairs of equations that are compared.

We have run the same experiment on ten other programs; these programs

heapsort			
	operations	total steps	steps/operation
inclusion	3965	8591	2.17
intersection	1947	2493	1.28
negation	541	569	1.05
range for other trials			
	operations	total steps	steps/operation
inclusion	8098-29724	12765-55398	1.54-1.87
intersection	2190-8236	2894-11225	1.29-1.38
negation	506-2165	540-2325	1.05-1.08

Figure 9: Results of experiments.

are between one hundred and five hundred lines long. The second half of Figure 9 gives the range of measurements in these trials. With the exception of inclusion, the amortized cost of each operation is about the same as in heapsort. Inclusion tests are noticeably cheaper in the general trial; presumably this is because heapsort was designed specifically to stress the inclusion algorithm. Overall, the result of this experiment shows that, with the optimizations, the amortized cost of regular tree operations is nearly constant in practice.

8 Conclusion

Regular tree expressions are a powerful tool for describing sets of terms of a free algebra; as such, several program analysis algorithms based on regular tree expressions have been proposed. We have described our implementation of regular tree expressions for a type inference system. Our experience is that, with carefully designed algorithms and some optimizations, regular tree operations can be efficiently implemented.

Acknowledgements

The authors would like to thank Jennifer Widom, John Williams, and Ed Wimmers for discussions and their comments on earlier versions of this paper.

References

[AM91] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, 1991.

[AW91] A. Aiken and E. Wimmers. A decision problem for set constraints. Research Report Forthcoming RJ, IBM, 1991.

[B+89] J. Backus et al. FL language manual, parts 1 and 2. Research Report RJ 7100, IBM, 1989.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[GS84] F. Gecseg and M. Steinby. *Tree Automata.* Academei Kaido, Budapest, 1984.

[HJ90] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.

[HJ91] N. Heintze and J. Jaffar. Set-based program analysis. Draft manuscript, 1991.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[JM79] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[Mic68] D. Michie. 'Memo' functions and machine learning. *Nature*, (218):19–22, April 1968.

[Mis84] P. Mishra. Towards a theory of types in PROLOG. In *Proceedings of the First IEEE Symposium in Logic Programming*, pages 289–298, 1984.

[MR85] P. Mishra and U. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 7–21, 1985.

[Mur90] B. R. Murphy. A type inference system for FL. Master's thesis, MIT, 1990.

[Sei89] H. Seidl. Deciding equivalence of finite tree automata. In *6th Annual Symposium on Theoretical Aspects of Computer Science*. Lecture Notes in Computer Science, February 1989.