A Logic Your Typechecker Can Count On:
Unordered Tree Types in Practice

J. Nathan Foster Benjamin C. Pierce
University of Pennsylvania
{jnfoster,bcpierce}@cis.upenn.edu

Alan Schmitt
INRIA Rhône-Alpes
Alan.Schmitt@inrialpes.fr

Abstract
Type systems featuring counting constraints are often studied, but seldom implemented. We describe an *efficient* implementation of a type system for unordered, edge-labeled trees based on Presburger arithmetic constraints. We begin with a type system for unordered trees and give a compilation into counting automata. We then describe an optimized implementation that provides the fundamental operations of membership and emptiness testing. Although each operation has worst-case exponential complexity, we show how to achieve reasonable performance in practice using a combination of techniques, including syntactic translations, lazy automata unfolding, hash-consing, memoization, and incremental tree processing implemented using partial evaluation. These techniques avoid constructing and examining large structures in many cases and amortize the costs of expensive operations across many computations. To demonstrate the effectiveness of these optimizations, we present experimental data from executions on realistically sized examples drawn from the Harmony data synchronizer.

Categories and Subject Descriptors F.4.3 [*Mathematical Logic and Formal Languages*]: Formal Languages—Classes defined by grammars or automata; I.1.3 [*Languages and Systems*]: Evaluation strategies; E.1 [*Data Structures*]: Trees

General Terms Languages, Algorithms, Performance

Keywords Tree Automata, Presburger Arithmetic

1. Introduction

Type systems with semantics based on arithmetic constraints permit natural descriptions of unordered trees, which arise in diverse areas of computing—memory management, mobile processes, computational linguistics, constraint-based logic programming, and processors for record- and semi-structured data, to name but a few. Such type systems have been studied extensively, but there are few serious implementations—most research has focused on issues of expressiveness and decidability. This work describes implementation techniques aimed at achieving good performance in practice.

Over many structures—including trees—type systems and automata have the same expressive power. Despite this connection, there are differences between the two notions that are important in practice. In particular, type systems typically have a rich, declarative syntax, while automata provide direct algorithms for deciding membership and emptiness. For these reasons, types usually appear at the front-ends of implementations, while automata provide the machinery to realize the fundamental operations at the back-end.

Classical tree automata are not well-suited to processing unordered trees, because their internal transitions are specified using the *positions* of children in the tree: to decide if an ordered tree is accepted, a tree automaton traverses the tree—either bottom-up or top-down—and tests, at each node, if every child is accepted by the corresponding successor state. By contrast, in a tree automaton with arithmetic constraints [22, 8, 2], instead of *matching* children and states by position, the automaton *counts* the number of children accepted by each successor state and tests if the resulting tallies satisfy a given arithmetic constraints. These counting automata form a natural basis for implementations of type systems over unordered trees (they are also useful over ordered trees where they express order-agnostic properties, also known as "numeric document queries" [22]). We will focus on the efficient implementation of counting automata where arithmetic constraints are expressed as formulas in Presburger arithmetic, also known as sheaves automata [8].

It is reasonably straightforward to build a naive implementation of counting automata. One only needs to know how to count the number of children accepted by each successor state and how to test if a set of tallies satisfies an arithmetic constraint. However, such an implementation will be unusable on any but the tiniest of inputs, for several reasons. First, testing the satisfiability of arithmetic constraints is expensive; when the constraints are stated as formulas in Presburger arithmetic, it requires double-exponential time. One might hope to keep the formulas small, but, in the obvious translation from types to automata, their length grows quadratically. Moreover, even if deciding arithmetic constraints could somehow be made fast, another source of exponential cost would remain: in the worst case, counting the number of children accepted by each successor state requires a recursive membership test on every immediate subtree and successor state.

Proceedings of the 5th ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X 2007).
January 20, 2007, Nice, France.

To avoid these pitfalls, our implementation employs a combination of techniques, which can be divided into three broad categories; the structure of the paper follows this division. After a brief review of tree types, Presburger arithmetic, and sheaves automata (Section 2), we present in detail the naive algorithm sketched above (Section 3), then consider each of the three categories of optimizations.

The first set of optimizations, described in Section 4, consists of simple syntactic transformations on formulas and automata to reduce their sizes. As an example, when compiling a type to a sheaves automaton, we often need to merge the transition relations for the automata compiled from the type's sub-expressions. This can be done using an all-pairs intersection of the elements of the transition relation, but leads to a quadratic blowup. However, elements in the intersection that are empty need not be included in the result; the blowup can be avoided in most cases. Similarly, the construction on Presburger formulas that lifts the addition operator from terms to formulas introduces an existential quantifier for every free variable that the two formulas have in common. Since addition is used frequently in the compilation of types to automata, this construction introduces many quantifiers, leading to very complicated formulas and slowing down constraint solvers significantly. If, however, it can be determined statically that the constraint expressed by a formula forces a given free variable to be zero, then no quantifier needs to be introduced for that variable. These simple optimizations do not address the exponential costs inherent in the operation of a counting automaton, but merely try to reduce the size of the inputs. Even so, because they apply so often and so dramatically, their impact is significant.

The second prong of our implementation strategy, described in Section 5, is an incremental algorithm for deciding membership. The naive algorithm first obtains an exact count of the number of tree slices belonging to each state, before testing whether the formula is satisfiable. In the worst case, this strategy can require testing every subtree and successor state. Our incremental algorithm is based on the observation that it is often possible to determine that a tree is *not* accepted by the automaton after collecting only partial counts. For example, if the formula specifies that a free variable must be zero but the tree has a child accepted by the corresponding state, then the counts of any other states are irrelevant. More specifically, when given a tree and an automaton state, we use partial evaluation to construct a specialized member function for those inputs that, at every step, takes as input a single tree and determines which state that child belongs to. We then translate the additional information gained at that step into further constraints on the formula and test its satisfiability. If the new formula is not satisfiable, then the tree is not accepted by the automaton and the algorithm returns false immediately. Although the fast paths in the incremental algorithm are followed only when a tree is *not* accepted by a state, the incremental algorithm also accelerates accepting runs because these typically involve many recursive calls that ultimately return false.

The third area of optimization, described in Section 6, uses hash-consing and memoization to share common structures and reuse the results of expensive computations. These optimizations turn out to be critical, for many reasons. As an example, consider the problem of representing and deciding the satisfiability of Presburger formulas. Rather than implementing a decision procedure directly, we use the excellent MONA tool [18] via the GENEPI interface [1]. We developed a simple interface to the C library, which translates the structures representing formulas into C structures that can be manipulated by MONA. However, this means that every formula has two representations; if our system produced too many formulas, the costs of allocating and maintaining both representations would become significant. Fortunately, the number of *distinct* formulas used in most programs is small—on the order of ten thousand. Thus, formulas are a good candidate for hash-consing. Additionally, our implementation caches the results returned by MONA's satisfiability procedure, ensuring that the emptiness of each formula is computed at most once. It also hash-conses trees and automata states and memoizes the results of the incremental member algorithm at several levels with similar benefits.

The discussion up to this point has focused on the membership testing algorithm. Section 7 briefly describes two more fundamental operations—*emptiness testing*, which, given a state S determines if there is some tree accepted by S, and *domain membership testing*, which, given a set of names d and a state S determines if d is the domain of some tree accepted by S—and sketches one final optimization that greatly improves their performance.

Of course, the history of engineering is full of examples of seemingly promising optimizations that fail to make much difference in practice. We are using our implementation in two components of a larger project called Harmony [13]: a type checker for a bi-directional tree transformation language [11], and a type-aware generic data synchronizer [10]. Section 8 presents experimental results for some real-world examples with various optimizations turned on and off.

Sections 9 and 10 discuss related and future work. Several appendices present optional material that may be of interest to expert readers.

2. Preliminaries

This section defines notation for trees, types, and automata and reviews the translation from types to automata. The notions of tree types and counting automata, as well as the translations between them, are closely based on ones invented by Dal Zilio, Lugiez, and Meyssonnier [7]. For completeness we give a brief, self-contained review here and highlight the differences in our formulation; a more leisurely introduction to unordered tree types and counting automata can be found in their original article. The structures in our data model, however, are slightly different than the "information trees" used in previous studies: we work with unordered trees where every tree node has at most one child with a given label; information tree nodes may have multiple children with the same label. We chose the simpler notion of trees because it is the one used in our main application—the Harmony data synchronizer. In that setting, certain tasks, such as identifying and aligning data from each replica, become somewhat simpler when the trees do not have repeated children. However, as each of our trees is also an information tree, we believe that each of our results carry over with only minor modifications.

Data Model Let $\mathcal{N}$ denote a set of labels. We work with the set $\mathcal{T}$ of *deterministic trees* over $\mathcal{N}$: unranked, unordered, edge-labeled trees with labels drawn from $\mathcal{N}$, where a given tree node has at most one child with a given label. We write trees sideways. The empty tree is written **{}**; in non-empty trees each pair of curly braces denotes a node and every "$\mathtt{n} \mapsto t_n$" denotes a child labeled n leading to a subtree t_n. Every deterministic tree can be represented as a partial function from names to trees; we write $\mathsf{dom}(t)$ for the domain of t—i.e. the set of names of its children, and $t(n)$ for the immediate subtree of t labeled with n. The concatenation operator $\cdot$ is commutative and defined only for pairs of trees with disjoint domains; $t \cdot t'$ denotes the tree mapping n to $t(n)$ for $n \in \mathsf{dom}(t)$ and to $t'(n)$ for $n \in \mathsf{dom}(t')$. Meta-variables n and m range over labels and t ranges over trees.

Deterministic Tree Types The syntax of *deterministic tree types* (DTTs) is as follows:

$$T ::= \texttt{\{\}} \mid r\texttt{[}T\texttt{]} \mid r\texttt{[}T\texttt{]*} \mid T\texttt{+}T \mid T\texttt{|}T \mid \texttt{\~{}}T \mid X$$

Meta-variables r range over regular expressions over labels—i.e., labels closed under union, concatenation, and Kleene-star. To distinguish regular expressions from DTTs, we enclose them in angled brackets unless the regular expression is a single character—e.g., —$\langle a \,|\, b\rangle$ denotes the set $\{a, b\}$. We also use the negation operator—e.g., $\langle \texttt{\~{}}\{\}\rangle$ denotes the set of all labels. A label *matches* a regular expression if it belongs to the set it denotes.

The type `{}` denotes the singleton set containing just the empty tree (note that `{}` is not the empty type); a type atom $r\texttt{[}T\texttt{]}$ denotes the set of trees with a single child whose label matches r and subtree belongs to the set denoted by T; repeated atoms $r\texttt{[}T\texttt{]*}$ represent the Kleene-closure of the same set; types $T_1\texttt{+}T_2$ and $T_1\texttt{|}T_2$ denote concatenation and union lifted element-wise to sets of trees respectively; $\texttt{\~{}}T$ denotes the relative complement of the set denoted by T in $\mathcal{T}$; and a recursion variable X denotes the same set as the denotation of the type it is bound to in the static, global type environment $\Delta = \{X_1 = T_1, .., X_k = T_k\}$. Type definitions in Δ may be mutually-recursive, but must obey a strong contractiveness constraint discussed below: recursion variables may only appear below atoms and repeated atoms. Note that although deterministic trees do not have repeated children, the type $r\texttt{[}T\texttt{]*}$ can still be useful since r denotes a set of names, and that types such as $a\texttt{[}T_1\texttt{]+}a\texttt{[}T_2\texttt{]}$, where a is a single name, are allowed but are semantically empty.

DTT terms resemble Tree Logic formulas, as described by Dal Zilio *et al.* [7], but there are some key differences. First, in Tree Logic, the labels appearing in type atoms are specified using finite or cofinite sets, whereas in DTTs they are specified using regular expressions. This small, somewhat obvious change nevertheless enhances the expressiveness of the logic in a useful way—e.g., compare the sizes of the descriptions of the set of date strings of the form "yyyy-mm-dd" as an explicit finite set and as a regular expression. Second, we give a full treatment of (vertical) recursion; in Tree Logic the extension to recursive formulas is only sketched informally. In particular, reasoning that the compilation from types to automata terminates requires choosing the restrictions on contractiveness of type definitions and identifying the state space of the automaton with care. Third, in Tree Logic formulas, the Kleene-star operator can be applied to arbitrary types, whereas DTTs can only express the Kleene-closure of atoms. This represents a real restriction—there are sets definable by Tree Logic formulas that cannot be described using DTTs; e.g., the set of trees with an even number of children. We chose to make the restriction because many types expressed using Kleene-star collapse to simpler types when interpreted over deterministic trees. For example, the type $(\texttt{a[}\top\texttt{]+b[}\top\texttt{]})\texttt{*}$ (where $\top$ denotes the set of all trees and can be defined as $\top = r_\top\texttt{[}\top\texttt{]*}$ and the regular expression $r_\top$ is $\langle \texttt{\~{}}\{\}\rangle$) is semantically equivalent to $\texttt{\{\}|(a[}\top\texttt{]+b[}\top\texttt{])}$, because the root of every tree has at most one child labeled `a` and one labeled `b`. We also made this decision because the compilation of the full Kleene-star operator requires an expensive computation on Presburger formulas. However, as our automata implementation handles arbitrary Presburger formulas—including formulas representing the Kleene-closure of DTTs—adding support for Kleene-star would only involve changes to the front-end.

Presburger Arithmetic To describe counting automata in detail, we must first fix a formalism for writing down arithmetic constraints. Presburger arithmetic is the decidable first-order theory of the naturals with addition but without multiplication. Expressions in Presburger arithmetic include constants, variables, and sums, and formulas include equalities between expressions, boolean combinations of formulas, and quantified formulas:

$$\begin{array}{rcl} e & ::= & i \mid x_j \mid e\texttt{+}e \\ \phi & ::= & e\texttt{=}e \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \exists.\ \phi \end{array}$$

We use a de Bruijn representation—a variable x_j within the scope of k quantifiers represents the $(j-k)$th free variable if $j \geq k$, and otherwise is bound by the jth enclosing quantifier, counting from the inside-out. This representation of variables is slightly more complicated on paper, but simplifies the presentation of sheaves automata.

Some of the connectives are semantically redundant; we choose a larger set because the translation of formulas to a minimal set increases their size. However, because the GENEPI interface to MONA can only represent linear equalities, we are forced to treat inequalities as syntactic sugar—e.g., $(e_1 \leq e_2)$ becomes $(\exists.\ e_1 = e_2 + x_0)$. We write $\mathsf{fv}(\phi)$ for the set of free variables of ϕ; when $\mathsf{fv}(\phi) = \{x_0, .., x_k\}$, we write $\phi[e_1, .., e_i]$ for the instantiation of the first $j = min(i, k)$ free variables of ϕ with the corresponding expressions.

The semantics of a Presburger formula is the set of vectors of naturals that satisfy it. We write vector variables in a bold-face type and individual vectors with angled brackets: $\mathbf{v} = \langle n_0, .., n_k\rangle$. Projection is defined in the usual way: $\langle n_0, .., n_i, .., n_k\rangle(i) \triangleq n_i$. We write $\mathbf{v} \models \phi$ if $\mathbf{v}$ satisfies ϕ, and $\models \phi$ if $\mathbf{v} \models \phi$ for some $\mathbf{v}$. Appendix A gives the standard definition of the satisfaction relation.

Sheaves Automata A *sheaves automaton* comprises a finite set of states, and a mapping Γ from states to *sheaves formulas*. The transition behavior from a state is given by the sheaves formula associated to it in Γ. Each sheaves formula has two components—a Presburger formula ϕ and a list of *elements*, each of the form $r_i[S_i]$, where r_i is a regular expression called the *tag* of the element, and S_i is a state. The operation of a sheaves automaton is like a bottom-up regular tree automaton. Let t be a tree and S be an automaton state with $\Gamma(S) = (\phi, [r_0[S_0], .., r_k[S_k]]))$. For each i in the range 0 to k, let c_i be the number of children $n \in \mathsf{dom}(t)$ for which $n \in r_i$ and $t(n)$ is accepted by S_i. Then t is accepted by S iff $\langle c_0, .., c_k\rangle \models \phi$.

Note that the integers that represent variables in de Bruijn notation give the correspondence between free variables in ϕ and elements—the constraint on x_i controls the number of children whose name matches r_i with subtrees accepted by S_i.

Sheaves automata and sheaves formula are subject to certain well-formedness conditions. A sheaves formula $(\phi, \mathbf{E})$ with $|\mathbf{E}| = k$ is well-formed iff the free variables of ϕ are $\{x_0, .., x_{k-1}\}$; the elements are pairwise disjoint—i.e., if the list includes $r_i[S_i]$ and $r_j[S_j]$ and there exists a tree accepted by both S_i and S_j, then the regular languages denoted by r_i and r_j are disjoint; and the elements are generating—i.e., for every tree t and label n there is an element $r_i[S_i]$ such that $n \in r_i$ and t is accepted by S. A list of elements obeying these conditions is called a *basis*. A sheaves automaton is well-formed iff every sheaves formula in the range of Γ is well-formed. (Although bases are characterized semantically, it is simple to check syntactically that the sheaves formulas compiled from DTTs are well-formed.) These well-formedness conditions guarantee two properties. First, because the elements are non-overlapping, every tree has a unique decomposition over the basis, which means that the semantics of a sheaves automata is well-defined. Second, because the elements generate the set of all tree slices, certain constructions are simple. For example, $(\phi, \mathbf{E})$ and $(\neg\phi, \mathbf{E})$ accept complementary sets of trees.

As an example, the type `({}|(a[⊤]+b[⊤]))` is equivalent to the sheaves automaton state S where $\Gamma(S)$ is

$$\left(\begin{array}{l} [(x_0=0 \wedge x_1=0) \vee (x_0=1 \wedge x_1=1)] \wedge (x_2=0), \\ [\mathtt{a}[\top]\,, \mathtt{b}[\top]\,, \langle \tilde{}\{\mathtt{a}, \mathtt{b}\}\rangle[\top]] \end{array} \right)$$

and $\top$ is a state that accepts all of $\mathcal{T}$. To see that the two are equivalent, observe that the constraints on x_0 and x_1 force the number of children described the elements $\mathtt{a}[\top]$ and $\mathtt{b}[\top]$ to both be 0 or 1, and that the constraint on x_2 forces the number of children belonging to the final element to be 0.

Compilation Next we describe a translation from DTTs into sheaves automata.

The type `{}` can be compiled into an equivalent sheaves automaton directly: $(x_0=0, [\langle\tilde{}\{\}\rangle[\top]])$. The single element in the basis, $\langle\tilde{}\{\}\rangle[\top]$, describes every tree slice, and the Presburger formula forces the number of children to be 0.

For type atoms, if T compiles to a state S_T then r`[`T`]` compiles to a state S with $\Gamma(S)$ as follows:

$$\left(\begin{array}{l} x_0=1 \wedge x_1=0 \wedge x_2=0, \\ [r[S_T]\,, r[\neg S_T]\,, \langle \tilde{}r\rangle[\top]] \end{array} \right)$$

It is easy to verify that the elements form a basis. Note that, in writing $\neg S_T$, we assume that it is possible to negate states; the details of this operation are discussed below. The compilation of repeated atoms r`[`T`]*` is similar, except that the constraint $x_0=1$ is replaced with $x_0 \geq 0$.

To compile a concatenation T_1`+`T_2, we first recursively compile T_1 and T_2 to states characterized by sheaves formulas $(\phi_1, \mathbf{E}_1)$ and $(\phi_2, \mathbf{E}_2)$. We then use a *refinement* operator to compute from $\mathbf{E}_1$ and $\mathbf{E}_2$ a common basis $\mathbf{E}$ and substitutions on variables σ_1 and σ_2, such that $(\sigma_1(\phi_1), \mathbf{E})$ and $(\phi_1, \mathbf{E}_1)$ are equivalent, and likewise for ϕ_2. The basis calculated by the refinement operation is obtained by intersecting every pair of elements in the input bases; σ_1 maps each x_i to the sum of variables corresponding to elements in $\mathbf{E}$ obtained by intersecting the $\mathbf{E}_1(i)$ with an element of $\mathbf{E}_2$, and similarly for σ_2:

$$\frac{\begin{array}{c} |\mathbf{E}_1| = k \qquad |\mathbf{E}_2| = l \\ \forall i \in 0..(k \times l - 1).\ \mathbf{E}'(i) = \mathbf{E}_1(i \div l) \wedge \mathbf{E}_2(i \mod l) \\ \forall i \in 0..(k-1).\ \sigma_1(x_i) = \sum_{j=0}^{l-1} x_{i \times l + j} \\ \forall i \in 0..(l-1).\ \sigma_2(x_i) = \sum_{j=0}^{k-1} x_{i + j \times l} \end{array}}{\mathsf{refine}((\phi_1, \mathbf{E}_1)\,, (\phi_2, \mathbf{E}_2)) = \sigma_1, \sigma_2, \mathbf{E}'}$$

Element intersection is calculated component-wise: $r[S] \wedge r'[S'] = \langle r \wedge r'\rangle[S \cap S']$ (intersections of states are discussed below). To finish the compilation of the original concatenation, we use an addition operator on Presburger formulas with the property that $\mathbf{v} \models \phi + \psi$ iff there exist vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ such that $\mathbf{v} = \mathbf{v_1} + \mathbf{v_2}$, with $\mathbf{v_1} \models \phi_1$ and $\mathbf{v_2} \models \phi_2$. The sum formula $\phi + \psi$ can be calculated by existentially quantifying the values of the vectors satisfying ϕ and ψ, and then adding the constraint that each free variable is the sum of the corresponding quantified variables:

$$\phi + \psi = \underbrace{\exists.,..,\exists.}_{2n} \left(\begin{array}{l} \bigwedge_{i \in 0..n-1} (x_{2n+i}=x_i+x_{n+i}) \\ \wedge\phi\wedge\psi[x_n,..,x_{2n-1}] \end{array} \right)$$

The final sheaves formula for the concatenation is the sum of the rewritten formulas over the common basis: $(\sigma_1(\phi_1) + \sigma_2(\phi_2), \mathbf{E}')$. Unions are compiled similarly, except that we use the union operator on the Presburger formulas instead of addition. To compile a negated type `~`T, we first compile T to a sheaves automata, and then negate its Presburger formula.

The compilation of recursion variables depends on a syntactic restriction—each recursion variable must appear below a type atom or repeated atom.[1] This restriction ensures that when we encounter a recursion variable X during compilation, we can simply use the state already compiled for X. There exist more elaborate compilation strategies where recursion variables may appear in non-contractive positions, as long as they do not appear below negations or intersections (which are equivalent to negations of unions of negations). However, the system with unrestricted recursion has an undecidable emptiness problem (the reduction with deterministic trees is similar to the proof given by Boneva and Talbot for information trees [3]). As we noted in the discussion surrounding Kleene-star, since our automata implementation handles arbitrary sheaves formulas, adding support for decidable fragments of DTTs with more general forms of recursion would only involve changing the syntax and compilation.

Compound States In describing the compilation from DTTs to sheaves automata, we have assumed that the space of automata states is closed under negation (in the atom cases) and intersection (in the refinement operator). We now show how to identify a set of automata states that is closed under these operations.

During compilation, an automaton state is introduced for each top-level type definition in Δ, and for each syntax node in a type. Because Δ is finite, the set of such *simple states* is finite. It follows that the set of boolean combinations of simple states is also finite. Therefore, we can take the set of automata states to be arbitrary boolean combinations of simple states.

In order to define algorithms that operate on states, it is helpful to have some canonical syntax for writing them down. We borrow a notational device from XDuce [16], and describe states as *compound states*. A compound state is a finite union of *complex states*, which represent sets of intersections and differences of simple states. Formally, if the X_is and Y_js are all simple states, then complex states are given by C and compound states by S:

$$C ::= (\{X_1,..,X_k\} \setminus \{Y_1,..,Y_l\}) \qquad S ::= \{C_1,..,C_k\}$$

[1] In our implementation, the restriction is implemented as a slightly more liberal check on recursive definitions: after the types in a single recursion group have been compiled, its variables may be used in non-contractive positions in subsequent groups. This facilitates compact descriptions of DTTs using type definitions.

The semantics of a complex state is the set of trees accepted by every X_i and no Y_j; the semantics of a compound state is the union of the sets denoted by the C_is.

With this notation fixed, it is simple to write down algorithms that symbolically compute boolean operations on states. For example, the negation of a compound state is the intersection of the negation of each complex state: $\neg\{C_1, .., C_k\} \triangleq \neg C_1 \cap .. \cap \neg C_k$. The other boolean operations on complex and compound states are straightforward; their definitions are given in Appendix B.

While the number of compound states is exponentially larger than the number of simple states, most compound states are never encountered during an execution run. In our implementation we exploit this fact and lazily expand the sheaves formulas for compound states as needed by membership and emptiness tests.

3. Basic Algorithm

The rest of the paper is devoted to developing and evaluating an efficient membership implementation for sheaves automata. As a starting point, we describe a baseline algorithm that correctly realizes the semantics of sheaves automata. Rather than implementing the bottom-up strategy directly, which would entail computing the set of accepting states for every node in the tree, it uses a top-down traversal that non-deterministically explores paths in the transition graph of the automaton until it finds the unique element accepting each internal node:

```
$mem(\{\!\!|n_0 \mapsto t_0, .., n_k \mapsto t_k|\!\!\}\,, S) =$
  let $(\phi, [r_0[S_0]\,, .., r_l[S_l]])) = \Gamma(S)$
  let $a =$ new int$[l + 1]$
  for $i$=0 to $l$ do $a$[i] := 0 done;
  for $i$=0 to $k$ do
    for $j$=0 to $l$ do
      if $n_i \in r_j$ and $mem(t_i, S_j)$ then
        ($a$[j] := $a$[j] + 1; break)
    done;
  done;
  $\langle a[0], .., a[l]\rangle \overset{?}{\models} \phi$
```

The algorithm uses two nested loops to traverse the immediate children of the tree and the elements, and to increment the count of the element that accepts each child and its subtree. It breaks out of the inner loop as soon as it identifies an accepting element—as the elements form a basis, there exists exactly one such element. The final result is computed by testing if the counts represent a vector satisfying the Presburger formula.

4. Syntactic Optimizations

The basic algorithm suffers from exponential costs stemming from two sources: the non-deterministic traversal of the automaton's state space, and satisfiability testing for Presburger formulas. In this section, we describe some simple syntactic optimizations that reduce the number of automata states and the sizes of formulas. Although these optimizations do not address the sources of exponential behavior, they have significant practical impact because they apply often and reduce the size of the state space and length of formulas dramatically. Studying these simple optimizations first also warms us up for the more complicated ones described in later sections.

Compact Bases The first optimization reduces the size of sheaves formulas by compacting useless elements produced by the refinement operator. As observed by Dal Zilio et al, the simple all-pairs refinement operator produces bases that have many empty elements. Consider refining the following bases: $[\texttt{a}[S]\,, \texttt{a}[\neg S]\,, \langle \texttt{\~{}\{a\}}\rangle[\top]]$ and $[\texttt{b}[S]\,, \texttt{b}[\neg S]\,, \langle \texttt{\~{}\{b\}}\rangle[\top]]$. The all-pairs intersection yields a basis with nine elements

$$\left[\begin{array}{l} \langle \texttt{a}\wedge\texttt{b}\rangle[S\wedge S]\,, \langle \texttt{a}\wedge\texttt{b}\rangle[S\wedge\neg S]\,, \langle \texttt{a}\wedge\texttt{\~{}\{b\}}\rangle[S\wedge\top]\,, \\ \langle \texttt{a}\wedge\texttt{b}\rangle[\neg S\wedge S]\,, \langle \texttt{a}\wedge\texttt{b}\rangle[\neg S\wedge\neg S]\,, \langle \texttt{a}\wedge\texttt{\~{}\{b\}}\rangle[\neg S\wedge\top]\,, \\ \langle \texttt{\~{}\{a\}}\wedge\texttt{b}\rangle[\top\wedge S]\,, \langle \texttt{\~{}\{a\}}\wedge\texttt{b}\rangle[\top\wedge\neg S]\,, \\ \langle \texttt{\~{}\{a\}}\wedge\texttt{\~{}\{b\}}\rangle[\top\wedge\top] \end{array}\right]$$

but only five of the nine are non-empty:

$$\left[\begin{array}{l} \langle\{\}\rangle[S]\,, \langle\{\}\rangle[\bot]\,, \texttt{a}[S]\,, \langle\{\}\rangle[\bot]\,, \langle\{\}\rangle[\neg S]\,, \\ \texttt{a}[\neg S]\,, \texttt{b}[S]\,, \texttt{b}[\neg S]\,, \langle \texttt{\~{}\{a, b\}}\rangle[\top] \end{array}\right]$$

It is simple to eliminate many empty elements by identifying empty regular expressions and obviously empty states—e.g., the intersection of a state with its negation—as refinements are calculated. Indeed, with this optimization enabled, our implementation produces the basis with five elements on the above input. Eliminating useless elements has a direct effect on the running-time of the member algorithm by reducing the number of iterations of the inner loop over the elements and, more critically, the number of free variables of Presburger formulas expressed over the basis.

Extended Syntax In many applications the same type is used to describe the structure below several labels. For example, the type of individual entries in a tree type representing address books might have atoms `tel-cell`, `tel-home`, and `tel-work`, all pointing to the same type one level down:

$$\texttt{tel-cell}[X]\texttt{+tel-home}[X]\texttt{+tel-work}[X]$$

There is a sheaves formula over a three-element basis:

$$\left(\begin{array}{l} (x_0 = 3 \wedge x_1 = x_2 = 0), \\ \left[\begin{array}{l} \langle\texttt{tel-cell}, \texttt{tel-home}, \texttt{tel-work}\rangle[X]\,, \\ \langle\texttt{tel-cell}, \texttt{tel-home}, \texttt{tel-work}\rangle[\neg X]\,, \\ \langle\texttt{\~{}tel-cell}, \texttt{tel-home}, \texttt{tel-work}\rangle[\top] \end{array}\right] \end{array}\right)$$

but the compilation function described in Section 2 produces one with a seven-element basis:

$$\left(\begin{array}{l} (x_0 = 1 \wedge x_2 = 1 \wedge x_4 = 1 \wedge x_1 = x_3 = x_5 = x_6 = 0), \\ \left[\begin{array}{l} \texttt{tel-cell}[X]\,, \texttt{tel-cell}[\neg X]\,, \\ \texttt{tel-home}[X]\,, \texttt{tel-home}[\neg X]\,, \\ \texttt{tel-work}[X]\,, \texttt{tel-work}[\neg X]\,, \\ \langle\texttt{\~{}tel-cell}, \texttt{tel-home}, \texttt{tel-work}\rangle[\top] \end{array}\right] \end{array}\right)$$

To make it simple to compile to the compact formula, we extend DTTs with two new forms:

$$\{r_1, .., r_k\}\texttt{[}T\texttt{]} \qquad \text{and} \qquad \{r_1, .., r_k\}\texttt{[}T\texttt{]?}$$

The first describes the set of trees with k children, one belonging to each of $r_1, .., r_k$, and with subtrees all belonging to T; the second type describes trees with at most k children subject to the same constraints. These types are compiled like atoms, except that the constraint $x_0 = 1$ is replaced by $x_0 = k$ and $x_0 \leq k$ respectively. If we rewrite the example from the address book as

$$\{\texttt{tel-cell}, \texttt{tel-home}, \texttt{tel-work}\}\texttt{[}X\texttt{]}$$

then the compiler produces the first, compact sheaves formula.

Compact Sums The next optimization streamlines the Presburger formulas produced by the addition operator.

Recall that for formulas ϕ and ψ with n free variables, the construction of $\phi + \psi$ adds $2n$ existential quantifiers. If, however, we detect that a variable x_j in ϕ is explicitly equal to zero, then the constraints on the jth component of every vector satisfying $\phi+\psi$ are just those expressed by ψ. Instead of existentially quantifying the values used to instantiate the jth variable of each formula and setting x_j to the sum of these quantified variables, we can simply instantiate the jth variable of ϕ with zero, and the jth variable of ψ with x_j (shifted up by the number of quantifiers used in the final construction). Because of the way that types are compiled—e.g., the compilation of an atom produces a formula in which exactly one variable is not zero—this optimization can often be applied in practice. Moreover, as it reduces both the size and complexity of Presburger formulas, it simplifies the satisfiability problems passed off to the external solver.

State Constants There are a handful of types that are encountered many times in applications. For example, in our examples, the types representing the universal type $\top$ and the singleton type containing only the empty tree `{}` represent a significant percentage of the total member tests. Although these types can be compiled to sheaves formulas,

$$\top \equiv (x_0 \geq 0, [\langle \tilde{}\{\} \rangle [\top]])$$
$$\texttt{\{\}} \equiv (x_0 = 0, [\langle \tilde{}\{\} \rangle [\top]])$$

because of their ubiquity, it makes sense to introduce constant states for them, so that membership can be calculated immediately, without examining bases or Presburger formulas. For example, by making $\top$ a constant, the membership function returns true immediately, but with the sheaves formula, we would have to traverse the whole tree, testing that the number of children satisfying the trivial formula $x_0 \geq 0$ at each node.

5. Incremental Algorithm

In many situations, it is possible to determine whether a tree is accepted by a state after examining only part of the tree. The basic algorithm always determines a complete decomposition of the tree over the elements before it calculates a result; along the way, it calculates the solutions to many irrelevant membership problems. In this section, we present an algorithm that avoids these irrelevant problems by processing trees incrementally.

The pathological behavior of the basic algorithm can be demonstrated by considering its behavior on list structures. Suppose that lists $[t_1, .., t_k]$ are represented in trees as cons cells: $\{\!|\texttt{hd} \mapsto t_1, \texttt{tl} \mapsto \{\!| .. \mapsto \{\!|\texttt{hd} \mapsto t_k, \texttt{tl} \mapsto \{\!|\!\}\,|\!\} .. |\!\}\,|\!\}$. The type of lists of T is $X = (\texttt{\{\}|hd[}T\texttt{]+tl[}X\texttt{]})$. If T compiles to a state S_T, then X compiles to S_X where $\Gamma(S_X)$ is

$$\left(\begin{bmatrix} (x_1 = x_3 = x_4 = 0) \wedge \\ (x_0 = x_2 = 0) \vee \\ (x_0 = x_2 = 1) \end{bmatrix}, \begin{bmatrix} \texttt{hd}[S_T], \texttt{hd}[\neg S_T], \\ \texttt{tl}[S_X], \texttt{tl}[\neg S_X], \\ \langle \tilde{}\{\texttt{hd}, \texttt{tl}\} \rangle [\top] \end{bmatrix} \right).$$

Now consider a run of the basic algorithm on a tree $[t_1, .., t_k]$ where none of the t_is belong to T. Assuming that the algorithm examines the children in alphabetical order, it will first determine that t_1, the subtree under `hd`, is accepted by $\neg S_T$. At this point, it already has enough information to see that l does not belong to T, since x_1 is constrained to be zero in the Presburger formula. The algorithm, however, does not stop; it proceeds to the other child, `tl`, and churns away, evaluating many recursive calls, until it finally determines that $[t_2, .., t_k]$ belongs to $\texttt{tl}[\neg S_X]$ and that the corresponding vector $\langle 0, 1, 1, 0, 0 \rangle$ does not satisfy the Presburger formula.

Incremental Algorithm The incremental algorithm turns the membership test on its head: rather than collecting all of the counts needed to decide whether a tree is accepted by a state, it collects information piecemeal and tries, at each step, to *refute* the assertion that the tree is accepted by the state. For example, when presented with the list above, it determines that the result is false after examining just the child labeled `hd`. (This might appear to help only in cases where the whole tree is not accepted at a state; actually, because it also helps quickly determine when subtrees do not belong to particular elements, it also improves performance in cases where the tree is accepted.)

The algorithm is divided into two phases. In the first phase, given a tree t and a state S, it constructs a Presburger formula ϕ' such that if ϕ' is unsatisfiable, then no tree with domain $\mathsf{dom}(t)$ is accepted by S. The second phase loops over the children. On the ith iteration, having already examined the children n_0 through n_{i-1}, it constructs a Presburger formula ϕ' such that if ϕ' is unsatisfiable then no tree with the same domain $\mathsf{dom}(t)$ and the subtrees below n_0 through n_i is accepted by S. Formally, the algorithm is defined as follows ($\sum$ denotes the addition operator lifted to Presburger formulas):

```
$mem'(\{\!| n_0 \mapsto t_0, .., n_k \mapsto t_k |\!\}, S) =$

  $/*$ Phase I: $*/$
  let $(\phi, [r_0[S_0], .., r_l[S_l]])) = \Gamma(S)$
  allocate fresh vars $y_{(i,j)}$ for each $i, j$ such that $n_i \in r_j$
  let $\phi' = \phi \wedge \bigwedge_i (\sum_j y_{(i,j)} = 1)$
                $\wedge \bigwedge_j (x_j = \sum_i y_{(i,j)}))$
                $\wedge \bigwedge \{(x_j = 0) \mid \neg\exists n_i \in \mathsf{dom}(t).\, n_i \in r_j\}$
  if $\not\models \phi'$ then return false

  $/*$ Phase II: $*/$
  for $i$=0 to $k$ do
    for each $j$ such that $n_i \in r_j$ do
      if $mem'(t_i, S_j)$ then $(\phi' := \phi' \wedge (y_{(i,j)} = 1)$; break)
    done
    if $\not\models \phi'$ then return false
  done
  return true
```

Intuitively, each fresh variable $y_{(i,j)}$ represents the possibility that the subtree t_i is accepted by S_j. In the first phase, the constraints added to ϕ expresses the conditions that every subtree belongs to exactly one element—i.e., that for every i there is exactly one j such that $y_{(i,j)} = 1$, that the fresh variables allocated for each element sum to the value of the corresponding free variable—i.e., that for every j we have $x_j = \sum_i y_{(i,j)}$, and that free variables for elements that do not match any child are equal to zero. If this formula is not satisfiable, then no tree with the same domain is accepted.

In the second phase, the algorithm checks the membership of subtrees. It examines each subtree t_i, and finds the unique element $r_j[S_j]$ such that n_i matches r_i and S_i accepts t_i. It then incorporates this information into a refined formula, by adding the constraint $(y_{(i,j)} = 1)$. If this formula is not satisfiable, then no tree with the same domain and children n_0 through n_i leading to subtrees t_0 through t_i is accepted.

Partial Evaluation Our implementation of the incremental algorithm uses partial evaluation. Most of the tricky calculations—on indices of free variables and on elements—are performed in the first phase, and these calculations only depend on the set of immediate children of t and the sheaves formula itself. Thus, given a tree domain d and a state S, we can construct a specialized membership function for S that is correct for all trees with domain d. We first create the formula ϕ' as above and test $\models \phi'$. If $\not\models \phi'$, then we return a constant member function that always returns *false* (since no tree with domain d is accepted by S). Otherwise, we return a specialized function that implements the second phase—it expects the subtree t_i and performs the corresponding step of the second phase. This function in turn either returns *false*, if it disproves that t is accepted by S, *true*, if every subtree has been processed, or else another function that expects a different subtree and performs the next step of the second phase.

For a single input, explicitly separating the first and second phases in this way does not improve performance. However, since we cache the specialized function (see Section 6), subsequent membership tests for trees with domain d can skip the first phase entirely and jump straight to the specialized membership function without examining the elements, manipulating free variables, or testing if labels match regular expressions.

Further Optimizations Finally, using a few more simple syntactic optimizations, we can reduce the number of fresh variables as well as the number of satisfiability tests we have to perform in the incremental algorithm. As we have already seen, Presburger formulas compiled from DTTs often force many of their free variables to be equal to zero, and these constraints can often be discovered using simple syntactic analyses. In cases where a variable x_j is explicitly zero, we can avoid allocating a fresh variable $y_{(i,j)}$ for each label n_i that matches the tag of $r_j[S_j]$, because for any tree t accepted by S, the constraint on x_j forces the subtree t_i to belong to a different element.

Additionally, if a given label n_i only matches the tag of a single element $r_j[S_j]$, then the constraints added to the Presburger formula in the first phase are $y_{(i,j)} = x_j$ and $y_{(i,j)} = 1$. These constraints are equivalent to the single constraint $x_j = 1$, which uses one less variable. Moreover, when the child n_i is processed during the second phase, since there is only a single element that could match and we have already tested that it can take the value one, there is no need to test the Presburger formula again; we can just test that t_i is accepted by S_j.

6. Hash-Consing and Memoization

The final collection of optimizations focuses on strategies for sharing common structures and caching the results of expensive computations for later reuse. We make extensive use of hash-consing and memoization throughout our system. Hash-consed structures have the property that only one copy of a given structure is ever live in the system. Memoized functions look up their arguments in a table of already-computed results and only perform their actual computation when the lookup misses.

Hash-consing and memoization work well together. In particular, looking up hash-consed structures in a memo table can be fast, even if they are large, because structural equality and pointer equality coincide. These benefits, however, do not come for free. Hash-consing adds overhead to every allocation. Memoized functions require additional memory to store the memo tables. And every call to a memoized function requires computing a hash code plus a table lookup, which might be more costly than recomputing the function.

In our implementation, we hash-cons Presburger formulas, automata states, and trees, and we memoize the compilation of Presburger formulas to MONA structures, the satisfiability test, and both phases of the member function. In this section, we suggest why these choices are sensible; experimental results supporting these claims are given in Section 8.

Presburger Formulas Presburger formulas are an obvious candidate for hash-consing. Even for fairly large types, the number of distinct formulas produced in the compilation to sheaves automata is small in comparison, because the same formulas appear many times. For example, the formulas $(x_0 = 1)$, $(x_0 = 1 \wedge x_1 = 0 \wedge x_2 = 0)$, etc., are produced in the compilation of every type atom. Moreover, because we use an external solver to decide the satisfiability of formulas, the concrete representation of each formula contains both a value representing its syntax and a reference to a C-structure allocated by the MONA back-end. These MONA representations can be quite large—formulas are themselves encoded as tree automata—so there are significant benefits to be gained by sharing representations among copies of the same formula. (Actually, the story is slightly more complicated: the automata realizing a specific Presburger formula have a notion of "width" that corresponds to the set of free variables in the formula—the automaton realizing the formula $(x_0 = 1)$ with a single free variable is different than the one with two free variables. The function that takes a width and a formula and compiles the MONA representation at that width uses a memo table where already-constructed representations are indexed by width. When possible it uses projection and inverse projection operations to narrow and widen an existing automaton to a new width if possible; only the very first automaton representing a formula is constructed from scratch.) The satisfiability function is also memoized so that the satisfiability of each formula is tested at most once.

Automata States The translation from DTTs to sheaves automata described in Section 2 introduces a fresh state for every syntax node appearing in the type. However, if the same type has already been compiled, then we can reuse its state in the automaton. To realize this optimization, we hash-cons the allocation of fresh simple states and only generate new states when a given pair of Presburger formula and basis have not yet been bound to a state. Because Presburger formulas are themselves hash-consed, checking the structural—i.e., syntactic—equality of two formulas is simple; checking the equality of the lists of elements requires comparing the lists using a dictionary ordering on their elements.

Merging identical states is an effective optimization for several reasons. First, since the size of the set of automata states is exponential in the number of simple states, reducing the number of simple states dramatically decreases the size of the automaton. Second, optimizations that identify states syntactically—such as the syntactic optimization that eliminates empty states during refinement—are enabled more often when there are fewer redundant states.

Trees We also hash-cons the structures representing trees. In this case, the memory saved by sharing common structure is less critical—unlike Presburger formulas, the representation of trees is relatively compact, and, unlike states, none of our algorithms perform worse when there are more representations of trees live in the system. Instead, the benefits of hash-consing trees become apparent when we memoize functions that take trees as arguments, such as the member function. Because of their size, it would be impractical to use structural equality on trees for every lookup in a memo table—the system would spend all its time comparing trees!

Member Functions Each state maintains a pointer to its own member function, and each function is memoized at two levels. The outer memo tables associates trees to the boolean results of the membership function for that state. If a lookup in this table hits, then the answer is returned immediately. Misses are passed off to the inner memo table, which associates tree domains to partially-evaluated membership functions. A hit in this table returns a function, which can then be used to run the incremental algorithm on the children of the tree. A miss causes the system to construct (and remember) a specialized member function from the domain of the tree and the state.

7. Additional Operations

Until now, our discussion has focused on deciding membership efficiently. In this section we briefly describe our implementations of two additional operations: *emptiness testing*, which, given a state S determines if there is some tree accepted by S; and an operation called *domain membership*, which, given a set of names d and a state S determines if d is the domain of some tree accepted by S. To save space, we defer the formal definitions to Appendix C.

Emptiness Our algorithm for deciding emptiness follows the co-inductive approach used in XDuce [17]. Given a state S we assume that it is empty and then check that $G(S)$ is empty under that assumption. To avoid computing the same emptiness tests many times, we maintain a cache of empty and non-empty types; these caches provide the initial values of the assumptions *mts* and *nmts* that are threaded through the co-inductive calculation.

Because we work with deterministic trees, checking that a sheaves formula is empty requires a little more work than for information trees. For information trees, to check that $(\phi, \mathbf{E})$ is empty, we would determine the states in $\mathbf{E}$ that are empty and then test that ϕ is not satisfiable when those elements are constrained to be zero. For deterministic trees, we need to ensure that each label is used at most once. To do this, we assume that the sheaves formulas have been preprocessed, using the refinement operator with a dummy basis, so that each of the tags in $\mathbf{E}$ denotes either a singleton or an infinite set. We then add to ϕ the constraint that the sum of elements with identical singleton tags is at most one.

Domain Membership Harmony's synchronization algorithm requires a somewhat unusual operation on types: testing whether a set of names is exactly the domain of some tree belonging to the type. Since there are no subtrees to deal with, it might seem that domain membership is simpler than full membership testing. Conversely, since there no subtrees available, it might seem that domain membership is a more difficult problem—we need to determine whether there are *any* subtrees that could be combined with the tree domain to form a tree accepted by the automaton. In fact, by combining the emptiness and membership algorithms, we obtain an algorithm for deciding domain membership.

Emptiness Counterexamples This last algorithm requires one final optimization. It is among the simplest in our implementation, but it has a big impact on the performance of our synchronization algorithm (described in detail in [10]). The inputs to the synchronizer are three trees—two current replicas and a common ancestor—plus a type. The algorithm first checks that both replicas belong to the type, and then walks down all three input trees recursively, identifying the changes in each replica with respect to the ancestor and assembling those changes into updated replicas as long as it is possible to do so without breaking the invariants expressed by the type. The key property of the algorithm is that, for types that express only "local" properties of trees, it only needs to test that the domains of the assembled result belong to the set of domains of trees in the type to guarantee that the whole result will belong to the type. Thus, the key operations on types are membership and domain membership; the emptiness test is invoked indirectly by the domain membership algorithm.

The optimization that makes synchronization perform well is dead simple: whenever we determine that a tree or a domain is accepted by a state, we have also found a counterexample demonstrating that the state is not empty. We can safely add it to the non-empty cache of types. In the synchronization algorithm, this small optimization is a huge win because most of the emptiness queries generated by the domain membership algorithm are for states that we will have already seen when we tested the membership of the replicas in the top-level schema.

8. Experimental Results

In this section we present timing data and statistics from several experiments run using our system. The first experiment takes trees representing address book data and validates them against a tree schema loosely based on the vCard standard [9]. The second is a parser that takes ASCII text where some lines are decorated as headings and subheadings and transforms it into a tree structure where the nested structure described by the headings is made explicit. The third is a bi-directional lens program that maps between calendar data represented as iCalendar files and a simplified tree form suitable for synchronization. In each program, performance depends critically on the behavior of the member algorithm. For the validator, this is obvious—it *is* a membership tester. The text parser and iCalendar lens are both implemented in a tree transformation language [12] where conditionals and assertions are evaluated by testing membership of a tree in a type. Several aspects of these experiments are artificial, but they emphasize the effect of type operations on performance: the structured text parser operates by exploding the each line into a list of characters rather than processing entire lines; we ran the iCalendar example with run-time assertions enabled at the entry and exit points of each function.

We ran the experiments on a 1.4GHz Intel Pentium III machine equipped with 2GB of memory and running the SuSE operating system with Linux kernel version 2.6.16. System and user running times were collected using the standard POSIX timing functions; statistics about the number of calls to various functions and the hit rates of caches were collected by instrumented versions of the functions. We halted any experiment whose running time was more than four minutes; several experiments exceeded the memory limits imposed by the operating system and were also terminated. Each experiment was run on a range of inputs, varying in size from a single address book or calendar entry or line of text up to several thousand entries or lines, and

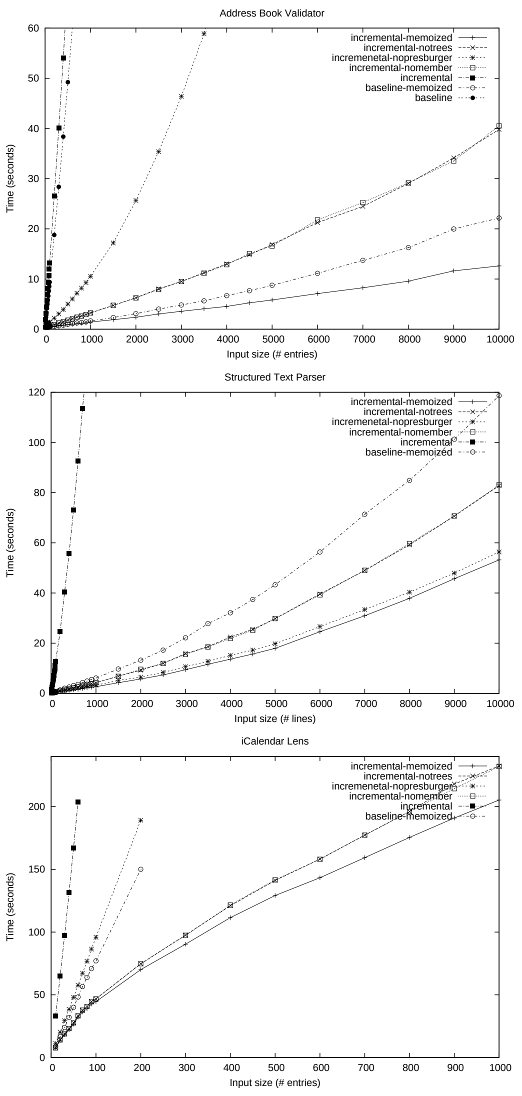

Figure 1. Experimental Results

in a variety of configurations, including the basic algorithm with and without memoization, and the incremental algorithm with full, selective, and no memoization. The types used to validate address book entries, parse text, and transform calendars compiled to sheaves automata with 312, 105 361 states respectively. The inputs to each program were generated by interleaving snippets of Joyce's *Ulysses* into the appropriate format—e.g., we created address book and iCalendar entries by randomly selecting the fields present in each entry and populating each field with text drawn from the text; for the text parser, we chose lines of text at random to be headings or subheadings.

Graphs of the timing results of the experiments in each configuration are shown in Figure 1. The lines labeled "baseline" refer to the basic algorithm and similarly for "incremental." Labels containing "notrees," "nopresburger," and "nomember" refer to experiments where the hash-consing and memoization optimizations for those structures were disabled. We did not perform tests on configurations with the simple syntactic algorithms turned off—the bases and formulas grow so quickly that the implementation is unusable on non-trivial inputs.

As the graphs show, neither the incremental algorithm alone nor memoization alone do as well as all of the optimizations do together. The performance of the basic algorithm alone is predictably bad—on address book and iCalendar entries its plot is nearly vertical; for the text parser, memory usage for the smallest input exceeded operating system limits. The basic algorithm performs much better in both examples when hash-consing and memoization are enabled, but the incremental algorithm outperforms it when the same optimizations are available. Interestingly, the incremental algorithm depends critically on memoization and hash-consing. Intuitively this makes sense—e.g., we would expect the memoization of Presburger results to be critical since it performs a less aggressive traversal of the tree and automaton but solves many more formulas at each node.

The following table gives some simple statistics collected from the experiments. In order, the columns are as follows: the total number of Presburger formulas allocated in the system and the hit rate in the hash-cons table; the total number of satisfiability queries and the hit rate in the memo table; and the total number of trees allocated and the hit rate in the hash table.

	Formulas		Sat		Trees	
Addr	107711	99.8%	25744	99.9%	107711	42.1%
Txt	12580	99.1%	222	92.8%	3507706	81.4%
Cal	116939	97.4%	17600	87.8%	407652	76.5%

The high hit rates validate the hash-consing and memoization strategies we chose in Section 6.

9. Related Work

Automata with counting have been proposed numerous times. Courcelle [6] noticed that the discriminating power of monadic second-order logic is weak on unordered trees and proposed adding counting constraints. Later, Dal Zilio, Lugiez, and Meyssonnier [8] and Seidl, Schwentick, Muscholl, and Habermehl [22] independently proposed equipping automata with Presburger constraints; Dal Zilio *et al.*'s starting point was a static (i.e., non modal) fragment of ambient logic [5], while Seidl *et al.* were interested in numeric document queries—order-agnostic queries over ordered tree structures. Boneva and Talbot survey the expressive power of all three systems [2] and (with Tison) show that full horizontal recursion makes satisfiability undecidable [3]. They describe several decidable fragments.

Other formalisms for unordered trees include Ohsaki's study of AC-closure of regular tree languages [20] and a number of papers on feature logics and automata from the Oz group (for example, [19]). Rounds [21] surveys a range of work on feature logics.

Recent work by Buneman, Cong, Fan, and Kementsietsidis [4] applies partial evaluation to XML query processing in a distributed setting.

Hosoya and Murata [15] describe an implementation of a type system for attribute-element constraints that handles ordered and unordered structures. Their implementation operates directly on the syntax of types.

Hague [14] describes an implementation of a membership checker along lines broadly similar to ours. His checker handles information trees, as opposed to deterministic trees, and uses OMEGA as its external solver. Lacking some of our optimizations, and largely due to the limitations of the

OMEGA tool, his implementation is more limited in the size of examples it can handle.

10. Future Work

There are many possible directions for future study following from this work. One obvious direction is to implement our own solver for Presburger arithmetic. This direct approach would alleviate some of the overhead of maintaining shadow MONA structures for every formula and could potentially admit more compact automata representations and optimized operations, such as refinement of formulas with new constraints. Another area we have not yet explored is determinization of automata. The incremental algorithm determinizes certain transitions, when it can determine that a single element matches a given child, but a more global analysis of automata would certainly do better. Finally, we would like to investigate extending our implementation techniques—in particular the incremental algorithm and partial evaluation—to other settings including tree automata operating on ordered trees.

Acknowledgments

We would like to thank Haruo Hosoya, Christian Kirkegaard, Stéphane Lescuyer, and the members of the Penn PL Club for helpful discussions, as well as the anonymous referees for many helpful suggestions. Harmony is supported by the National Science Foundation under grants ITR-0113226, *Principles and Practice of Synchronization* and IIS-0534592 *Linguistic Foundations for XML View Update*. This work was begun while Nathan Foster was visiting INRIA Rhône-Alpes; he wishes to thank the members of Projet Sardes for their generous hospitality during his stay, and the additional support of an NSF GRF.

References

[1] S. Bardin, J. Leroux, and G. Point. FAST Extended Release. In *International Conference on Computer Aided Verification (CAV), Seattle, WA*, volume 4144 of *Lecture Notes in Computer Science*, pages 63–66. Springer-Verlag, Aug. 2006.

[2] I. Boneva and J.-M. Talbot. Automata and logics for unranked and unordered trees. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 500–515. Springer, 2005.

[3] I. Boneva, J.-M. Talbot, and S. Tison. Expressiveness of a spatial logic for trees. In *LICS*, pages 280–289. IEEE Computer Society, 2005.

[4] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *VLDB*, pages 211–222. ACM, 2006.

[5] L. Cardelli and G. Ghelli. Tql: a query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004.

[6] B. Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[7] S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A Logic You Can Count On. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy*, pages 135–146. ACM Press, Jan. 2004.

[8] S. Dal-Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In N. D. Jones and X. Leroy, editors, *POPL*, pages 135–146. ACM, 2004.

[9] F. Dawson and T. Howes. RFC 2426: vCard MIME directory profile, Sept. 1998.

[10] J. N. Foster, M. B. Greenwald, C. Kirkegaard, B. C. Pierce, and A. Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 2006. To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.

[11] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, pages 233–246, 2005.

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 2006. To appear. Extended version available as University of Pennsylvania technical report MS-CIS-03-08. Preliminary version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004; extended abstract presented at *Principles of Programming Languages (POPL)*, 2005.

[13] J. N. Foster, B. C. Pierce, and A. Schmitt. *Harmony Programmer's Manual*, 2006. Available from `http://www.seas.upenn.edu/~harmony/`.

[14] M. Hague. Static checkers for tree structures and heaps, 2004. Final year project report, Imperial College.

[15] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *International Conference on Implementation and Application of Automata, Santa Barbara, CA*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003.

[16] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[17] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, Jan. 2005. Preliminary version in ICFP 2000.

[18] N. Klarlund and A. Møller. The MONA project, 2001.

[19] J. Niehren and A. Podelski. Feature automata and recognizable sets of feature trees. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT*, volume 668 of *Lecture Notes in Computer Science*, pages 356–375. Springer, 1993.

[20] H. Ohsaki. Beyond regularity: Equational tree automata for associative and commutative theories. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 539–553. Springer, 2001.

[21] W. C. Rounds. Feature logics. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 475–533. Elsevier, Amsterdam, 1996.

[22] H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in trees for free. In *International Colloquium on Automata, Languages and Programming (ICALP), Turku, Finland*, volume 3142 of *Lecture Notes in Computer Science*, pages 1136–1149. Springer-Verlag, July 2004.

A. Presburger Semantics

The value of an expression in a vector is:

$$\begin{aligned}
\mathsf{val}(\mathbf{v}, i) &\triangleq i \\
\mathsf{val}(\mathbf{v}, x_j) &\triangleq \mathbf{v}(j) \\
\mathsf{val}(\mathbf{v}, e_1 + e_2) &\triangleq \mathsf{val}(\mathbf{v}, e_1) + \mathsf{val}(\mathbf{v}, e_2)
\end{aligned}$$

The satisfaction relation is:

$$\begin{array}{lll}
\mathbf{v} \models e_1 = e_2 & \text{iff} & \mathsf{val}(\mathbf{v}, e_1) = \mathsf{val}(\mathbf{v}, e_2) \\
\mathbf{v} \models \phi_1 \vee \phi_2 & \text{iff} & \mathbf{v} \models \phi_1 \text{ or } \mathbf{v} \models \phi_2 \\
\mathbf{v} \models \phi_1 \wedge \phi_2 & \text{iff} & \mathbf{v} \models \phi_1 \text{ and } \mathbf{v} \models \phi_2 \\
\mathbf{v} \models \neg\phi_1 & \text{iff} & \mathbf{v} \not\models \phi_1 \\
\mathbf{v} \models \exists.\ \phi_1 & \text{iff} & \exists n.\ \langle n, \mathbf{v}(0), .., \mathbf{v}(k-1)\rangle \models \phi \text{ where } k = |\mathbf{v}|
\end{array}$$

B. State Operations

This appendix gives the definitions of boolean operators on compound and complex states. Negating a complex state yields a compound state:

$$\neg\left(\{X_1, .., X_k\} \setminus \{Y_1, .., Y_l\}\right) \triangleq \{Y_1, .., Y_l, \neg X_1, .., \neg X_k\}$$

We write $\neg X_i$ as an abbreviation for $(\{\top\} \setminus \{X_i\})$. Intersecting a complex state by another complex state compound state simply combines their intersections and differences:

$$\begin{aligned}
&(\{V_1, .., V_k\} \setminus \{W_1, .., W_l\}) \cap (\{X_1, .., X_m\} \setminus \{Y_1, .., Y_n\}) \\
\triangleq\ &(\{V_1, .., V_k, X_1, .., X_m\} \setminus \{W_1, .., W_l, Y_1, .., Y_n\})
\end{aligned}$$

Negating a compound state intersects the negations of each component complex state:

$$\neg\{C_1, .., C_k\} \triangleq \neg C_1 \cap .. \cap \neg C_k$$

Intersecting two compound states intersects each complex state from the first with every complex state from the second.

$$\{C_1, .., C_k\} \cap \{D_1, .., D_l\} \triangleq \left\{ \begin{array}{c} (C_1 \cap D_1), .., (C_1 \cap D_l), \\ .. \\ (C_k \cap D_1), .., (C_k \cap D_l) \end{array} \right\}$$

C. Additional Operations

The mutually-recursive functions *empty* and *check* are defined as follows (the parameters *mts* and *nmts* represent assumptions about empty and non-empty sets of types respectively):

```
$empty(S, mts, nmts)$ =
  if $S \in mts$ then true
  else if $S \in nmts$ then false
  else
    let $mts' = (mts \cup S)$ in
    let $(is\_mt, mts'', nmts'') = check(\Gamma(S), mts', nmts')$ in
    if $is\_mt$ then $(true, mts'', nmts'')$ else $(false, mts, nmts)$

$check((\phi, [r_0[S_0], .., r_k[S_k]]), mts, nmts)$ =
  let $\phi' = \phi \bigwedge_{i \in 0..k} \sum\{x_j \mid \mathsf{singleton}(r_j) \wedge r_j = r_i\} \leq 1$
  let $(mts', nmts') = (mts, nmts)$ in
  for $i$=0 to k do
    let $(is\_empty, mts'', nmts'') = empty(S_i, mts', nmts')$ in
    $mts' := mts''$;
    $nmts' := nmts''$;
    if $is\_mt$ then $\phi' := \phi' \wedge (x_j = 0)$;
  done
  if $\overset{?}{\models} \phi$ then $(false, mts, nmts)$ else $(true, mts', nmts')$
```

The domain membership algorithm is obtained by replacing the inner loop of the incremental algorithm, which determines the element that the given subtree belongs to and then adds the constraint that the corresponding fresh variable is one:

```
    for each $j$ such that $n_i \in r_j$ do
      if $mem'(t_i, X_j)$ then $(\phi' := \phi' \wedge (y_{(i,j)} = 1)$; break)
    done
    if $\not\models \phi'$ then return $false$
  done
```

with a loop that determines *all* of the non-empty elements (with tags matching the given label), and then adds the constraint that the sum of fresh variables corresponding to those elements is one:

```
  let $s_i = 0$ in
    for each $j$ such that $n_i \in r_j$ do
    if not $empty(X_j, mts, nmts)$ then $s_i := s_i + y_{(i,j)}$;
  done
  $\phi' := \phi' \wedge s_i = 1$;
  if $\not\models \phi' \wedge$ then return false
```

(For the sake of readability, we elide the statements that update the global empty / non-empty caches with the sets returned by *empty*.)