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 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, to our knowledge, there are no serious implementations—research has focused on issues of expressiveness and decidability. In this work, we investigate implementation techniques aimed at good performance in practice.

Type systems and automata are closely related; indeed, over certain structures, including trees, the two are equivalent. Type systems are usually syntactically simpler, but automata provide direct algorithms for membership and emptiness testing. For this reason, automata often serve as the back-ends of type system implementations, providing the machinery to realize the fundamental operations on types. Classical tree automata, however, are poorly suited to processing unordered trees, because their operation depends on a correspondence between the position of children in a tree and transitions to specific states. For example, to decide if an ordered tree is accepted, an automaton traverses the tree—either top-down or bottom-up—and tests, at each node, if every child is accepted by the corresponding successor state in the transition graph.

Many formal systems corresponding to automata extended with arithmetic constraints have been proposed; the list includes Courcelle's counting monadic second-order logic [5]; Seidl, Schwentick, Muscholl, and Habermehl's extensions of monadic second-order logic and unranked automata with Presburger constraints [18]; and Dal Zilio, Lugiez, and Meyssonnier's sheaves automata [7], a similar extension of unranked automata with Presburger constraints. An excellent study by Boneva and Talbot, relates the expressive power of these systems [2]. The operation of a tree automaton with arithmetic constraints is rather different from standard tree automata. Instead of matching children and states by position, the automaton counts the number of children accepted by each successor state and then tests if the resulting tallies satisfy a given arithmetic constraint. Counting automata are also useful for describing structure in ordered trees, where they succinctly express order-agnostic properties (or “numeric document queries” [18]).

It is reasonable straightforward to build a naive implementation of counting automata that correctly realizes their semantics. One only needs to know how to count the number of children accepted by each successor state and how to test if a list 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; specifically, 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 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. In this paper, we describe an efficient implementation of a type system for unordered trees based on sheaves automata. Our implementation achieves good performance using 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. The transitions from a given automaton state are described by a list of pairs of regular expressions and states: if the label of a given child in the tree matches the regular expression, then the corresponding state is used to process its subtree. When compiling a type to a sheaves automaton, we often need to take two such lists and compute a common list. This can be done using an all-pairs intersection of the elements of each list, resulting in a quadratic blowup. However, elements in the intersection that are empty need not be included in the result, avoiding the quadratic blowup in many cases. Similarly, the standard 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 from 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 variables 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. As a motivating example, consider the problem of representing and deciding the satisfiability of Presburger formulas. These optimizations turn out to be critical, for many reasons. For example, rather than implementing a decision procedure directly, we decided to use the excellent MONA tool [14] 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 a few tens of thousands, in our test cases. Thus, formulas are a good candidate for hash-consing. Whenever we create a formula, we first check if the same formula has already been allocated. We then cache the results returned by MONA’s satisfiability procedure, ensuring that the emptiness of each formula is computed at most once. We also hash-cons trees and automata states and memoize the results of the incremental member algorithm at several levels.

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 great-sounding optimizations that fail to make much difference in practice. We are using our implementation in two components of a larger project called Harmony [12]: a static type checker for a bi-directional tree transformation language [10], and a type-aware generic data synchronizer [9]. In Section 8, we evaluate the effectiveness of our techniques by presenting running times for some real-world examples with various optimizations turned on and off.

Section 9 discusses related work. Several appendices present optional material that may be of interest to expert readers.

2. Preliminaries

We begin by defining notation for trees, types, and automata and reviewing the translation from types into automata. The notions of tree types and counting automata used here are based on the types and automata invented by Dal Zilio et al [6]. 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 structure in our data model, however, are slightly different than the “information trees” used in previous studies: we work with unordered trees, in which 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, we believe that our results should carry over to information trees with only minor modifications.

**Data Model** Let \( \mathcal{N} \) denote a set of labels. We work with the set \( T \) 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 \( \text{"n} \mapsto t_n\text{"} \) 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 \( \text{dom}(t) \) 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 \( n \mapsto t_t(n) \) for \( n \in \text{dom}(t) \) and to \( t'(n) \) for \( n \notin \text{dom}(t) \). The variables \( n \) and \( m \) range over labels; \( t \) ranges over trees; and \( r, s \) range over string regular expressions over labels—i.e., labels closed under union, concatenation, and Kleene-star. In writing regular expressions, we also use the negation operator—e.g., \( \neg \{t\} \) denotes for the set of all labels.

We say that a label \( \text{matches} \) a regular expression if it belongs to the set it denotes.

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

\[
T ::= \{\} \mid \{\} T \mid T \cdot T \mid T + T \mid \neg T \mid T^* \mid X
\]

The type \( \{\} \) denotes the singleton set containing just the empty tree; a type atom \( r(T) \) denotes the set of trees with a single child whose label belongs to the regular set denoted by \( r \) and subtree belongs to the set denoted by \( T \); repeated atoms \( r(T) \) represent the Kleene closure of the same set; types \( T_1 + T_2 \) and \( T_1 \cdot T_2 \) denote concatenation and union lifted element-wise to sets of trees respectively; \( \neg T \) denotes the relative complement of the set denoted by \( T \) in \( T \); 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 \( \Delta = \{X_1 = T_1, ..., X_n = T_n\} \). Type definitions in \( \Delta \) may be mutually-recursive, but must obey a strong contractiveness constraint discussed below: recursion variables may only appear below atoms and repeated atoms. DTT terms closely resemble formulas in the Tree Logic, as described by Dal Zilio et al [6], 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 along an axis that is of both theoretical—see Appendix C—and practical interest—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 \( (a[T] \cdot b[T]) \) (where \( T \) denotes the set of all trees and can be defined as \( T = r(T_1[T] \cdot T_2[T]) \) and the regular expression \( r(T) \) is \( \{\} \) ) is semantically equivalent to \( \{\} \cdot (a[T] \cdot b[T]) \), 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. Following Dal Zilio et al, we use formulas in Presburger arithmetic, the 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:

\[
e ::= i \mid x \mid e + e \mid e \cdot e \mid \neg e \mid \neg e \mid e \lor e \mid e \land e
\]

We use a de Bruijn representation—a variable \( x_i \) within the scope of \( k \) quantifiers represents the \((i-k)\)th free variable if \( j \geq k \), and otherwise is bound by the \( j \)th enclosing quantifier, counting from the inside-out. 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 external solver we use only directly represents 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 \( \phi[e_1 \mid \ldots \mid e_n] \) for the set of free variables of \( \phi \); when \( \phi[e_1 \mid \ldots \mid e_n] \) for the instantiation of the first \( j = \min(i, k) \) free variables of \( \phi \) 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: \( \mathbf{v} = (n_0, \ldots, n_k) \). Projection is defined in the usual way: \((n_0, \ldots, n_i, \ldots, n_k)(i) = n_i \). We write \( \mathbf{v} \models e \) if \( e \) satisfies \( \phi \), and \( \models e \) if \( \mathbf{v} \models e \) for some \( \mathbf{v} \). Appendix A gives the standard definition of the satisfaction relation.

**Sheaves Automata** Next we review sheaves automata, which are finite state machines that describe sets of trees using Presburger constraints. A sheaves automaton comprises a finite set of states, and a mapping \( \Gamma \) from states to sheaves formulas. The transition behavior from a state is given by the sheaves formula associated to it in \( \Gamma \). Each sheaves formula includes 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 similar to a classical bottom-up regular tree automaton. Let \( t \) be a tree and \( S \) be an automaton state with \( \Gamma(S) = (\phi, \{r_i[S_i] \mid r_i \in R(S)\}) \). For each \( i \) in the range \( 0 \) to \( k \), let \( c_i \) be the number of children \( n < \text{dom}(t) \) for which \( n \in r_i \), and \( t(n) \) is accepted by \( S_i \). Then \( t \) is accepted by \( S \) if \( (c_0, \ldots, c_k) \models \phi \).

Sheaves automata and sheaves formula are subject to certain well-formedness conditions. A sheaves formula \( (\phi, [E]) \) with \( |E| = k \) is well-formed iff the free variables of \( \phi \) are \( \{x_0, \ldots, x_{k-1}\} \); 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 $\Gamma$ is well-formed. (Although bases are characterized semantically, for the sheaves formulas compiled from DTTs, they will be easy to verify syntactically.) These well-formed 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, ($\phi, E$) and ($\neg \phi, E$) accept complementary sets of trees.

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

$$\left( \begin{array}{c} (x_0 = 1) \land (x_1 + x_2 = 0), \\
(a[T], b[T], (\neg (a, b))[T]) \end{array} \right)$$

and $T$ is a state that accepts all of $T$. To see that the two are equivalent, observe that the constraints on $x_0$ and $x_2$ force the number of children described the elements $a[T]$ and $b[T]$ 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: $(x_0 = 0, (\neg ())[T])$. The single element in the basis, $(\{ \})[T]$, 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:

$$\left( \begin{array}{c} (x_0 = 1) \land (x_1 + x_2 = 0), \\
[r[S_T], r[\neg S_T], (r)(T)] \end{array} \right)$$

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 $T_1 \cdot T_2$, we first recursively compile $T_1$ and $T_2$ states characterized by sheaves formulas $(\phi_1, E_1)$ and $(\phi_2, E_2)$. We then use a refinement operator to compute from $E_1$ and $E_2$ a common basis $E$ and substitutions on variables $\sigma_1$ and $\sigma_2$, such that $(\sigma_1(\phi_1), E)$ and $(\sigma_2(E_2))$ are equivalent, and likewise for $\phi_2$. The basis calculated by the refinement operation is obtained by intersecting every pair of elements in the input bases; $\sigma_1$ maps each $x_i$ to the sum of variables corresponding to elements in $E$ obtained by intersecting the $E_1(i)$ with an element of $E_2$, and similarly for $\sigma_2$:

$$E_1(i) = k \quad E_2(l) = l$$

$$\forall i \in 0..(l - 1). E(i) = E_1(i \mod l) \land E_2(i \mod l)$$

$$\forall i \in 0..(l - 1). \sigma_1(x_i) = \sum_{j=0}^{i} x_{i-j} \land \sigma_2(x_i) = \sum_{j=0}^{i} x_{i-j+1}$$

refine($(\phi_1, E_1), (\phi_2, E_2)) = \sigma_1, \sigma_2, E'$$

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

$$\phi + \psi = \exists_1, \exists_2. \bigwedge_{i=0..n-1} (x_{2n+i} = x_i + x_{n+i})$$

The final sheaves formula is the sum of the rewritten formulas over the common basic: $(\sigma_1(\phi_1) + \sigma_2(\phi_2), E')$. 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. 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. Appendix C gives a reduction from the halting problem for two-counter machines to the emptiness problem for DTTs with horizontal recursion; it is adapted from a similar reduction using information trees by Boneva and Talbot, but uses deterministic trees instead of information trees. 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 horizontal 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 such boolean operations.

During compilation, an automaton state is introduced for each top-level type definition in $\Delta$, and for each syntax node in a type. Because $\Delta$ 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 XDue, 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 $X_i$s and $Y_i$s are all simple states, then complex states are given by $C$ and compound states by $S$:

$$C ::= \{X_1, \ldots, X_k\} \quad S ::= \{C_1, \ldots, C_k\}$$

1 Our implementation enforces a more relaxed condition: after the types in a 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 $C_j$s.

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:
\[ \neg \{C_1, \ldots, C_k\} = C_1 \cap \cdots \cap C_k. \]
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 formula for compound states as needed by mem-
bership and emptiness tests.

3. Basic Algorithm

The rest of the paper is devoted to developing and evalu-
ating an efficient membership implementation for sheaves
automata. Rather than implementing the bottom-up strategy
that correctly realizes the semantics of sheaves automata,
or simply all-pairs refinement operator produces
bases that have many empty elements. Consider re-
fining the following bases: $\{a[S], a[-S], (\neg \{a\})[T]\}$ and
$\{b[S], b[-S], (\neg \{b\})[T]\}$. The all-pairs intersection yields a basis
with nine elements
\[
\begin{align*}
(a \land b)[S \land S], (a \land b)[S \land -S], (a \land b)[S \land T], \\
(a \land b)[S \land S], (a \land b)[S \land -S], (a \land b)[S \land T], \\
(\neg \{a \land b\})[T \land S], (\neg \{a \land b\})[T \land -S], (\neg \{a \land b\})[T \land T]
\end{align*}
\]
but only five of the nine are non-empty:
\[
\emptyset[S], \emptyset[L], a[S], \emptyset[L], \emptyset[-S], a[-S], b[S], b[-S], (\neg \{a, b\})[T]
\]
It is simple to identify and eliminate many empty elements
by identifying empty regular expressions and obvi-
ously 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 ba-
sis with five elements on the above input. Eliminating use-
less 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 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 ex-
ample, the type of individual entries in a tree type represent-
ing address books might have atoms \texttt{tel-cell}, \texttt{tel-home},
and \texttt{tel-work}, all pointing to the same type one level down:

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

There is a sheaves formula over a three-element basis:
\[
\begin{aligned}
\{x_0 = 3 \land x_1 = x_2 = 0\}, \\
\left(\langle \texttt{tel-cell + tel-home + tel-work} \rangle[X]\right), \\
\left(\langle \texttt{tel-cell + tel-home + tel-work} \rangle[T]\right)
\end{aligned}
\]
but the compilation function described in Section 2 produces
one with a seven-element basis:
\[
\begin{aligned}
\{x_0 = 1 \land x_2 = 1 \land x_4 = 1 \land x_1 = x_3 = x_5 = x_6 = 0\}, \\
\left(\langle \texttt{tel-cell[X]}\rangle, \langle \texttt{tel-cell[X]}\rangle, \langle \texttt{tel-home[X]}\rangle, \\
\langle \texttt{tel-work[X]}\rangle, \langle \texttt{tel-work[X]}\rangle\right), \\
\left(\langle \texttt{tel-cell + tel-home + tel-work} \rangle[T]\right)
\end{aligned}
\]
To make it simple to compile to the compact formula, we extend DTTs with two new forms:
\[
\{r_1, \ldots, r_k\}[T] \quad \text{and} \quad \{r_1, \ldots, r_k\}[T]
\]
The first describes the set of trees with $k$ children, one
belonging to each of $r_1, \ldots, r_k$, 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 $x_0 = 0$ is replaced by
$x_0 = k$ and $x_0 \leq k$ respectively. If we rewrite the example
from the address book as
\[
\{\texttt{tel-cell.tel-home.tel-work} \rangle[X]
\]
then the compiler produces the first, compact sheaves for-

Compact Sums The next optimization streamlines
the Presburger formulas produced by the addition operator.
Recall that for formulas $\phi$ and $\psi$ with $n$ free variables, the
construction of $\phi + \psi$ adds $n$ conjuncts and $2n$ existential quantifiers. If, however, we can determine that $\phi$ forces $x_j = 0$, then the constraints on the $j$th component of every vector satisfying $\phi + \psi$ are just those expressed by $\psi$. Instead of existentially quantifying the values used to instantiate the $j$th variable of each formula and setting $x_j$ to the sum of these quantified variables, we can simply instantiate the $j$th variable of $\phi$ with 0, and the $j$th variable of $\psi$ 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 atoms produces a formula in which exactly one variable is non-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 $T$ 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,

$$T \equiv (x_0 \geq 0, \psi)$$

$$Q \equiv (x_0 = 0, \psi)$$

due to 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 $T$ 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 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 composition of the tree over the elements before it calculates

```
Phase I: */
let $\phi, [r_0[X_0],..,r_l[X_l]] = \Gamma(X)$
allocate fresh vars $y_{(i,j)}$ for each $i,j$ such that $n_i \in r_j$
let $z = \sum x_j$ in $\nuv()$ \exists $m_i$ in $\text{dom}(t)$, $n_i \in r_j$
let $\phi' = \phi \land (z = 0)$
if $\not\exists \phi'$ then return false
```

```
Phase II: */
for $i=0$ to $k$ do
for each $j$ such that $n_i \in r_j$
do
if $\not\exists \phi'$ then return false
end do
```

```
return true
```

The algorithm begins by picking fresh variables $y_{(i,j)}$ for every label $n_i \in \text{dom}(t)$ that matches the tag of an element $r_j[S_j]$. It then augments $\phi$ with constraints forcing the sums of the variables associated to each label to be equal to 1, the sum of the variables associated to $x_j[S_j]$ to be equal to $x_j$, and the sum of variables not matching any label to be equal to 0. The first group of constraints expresses the conditions that each subtree belongs to exactly one element; the second group connects the constraints on the free variables in $\phi$ to the sums of fresh variables for that element; the constraints on $z$ force the count of every element
whose tag does not match a label to 0. Intuitively, each fresh variable represents the situation where a particular subtree belongs to a particular element and the sum of all the fresh variables for a given label represents all of the possible ways of assigning its subtrees to elements. In the second phase, when the algorithm examines the subtree below each child, it determines which element the subtree belongs to, and then refines the formula further by assigning the corresponding fresh variable to 1. Because the sum of all the fresh variables for a given label is equal to 1, this constraints the rest of the fresh variables to be equal to 0. If the algorithm successfully processes all of the children in this way, then it returns $true$.

**Partial Evaluation** We implement the incremental algorithm using a partial evaluation technique. Notice that most of the tricky calculations on the indices of free variables and elements are performed in the first phase, and that these calculations only depend on the set of immediate children of $t$ and the sheaves formula. Thus, given a tree domain $d$ and a state $S$, we can compute a specialized membership function for trees having that domain. We first construct the formula $\phi'$ 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 function that takes a subtree $t(n)$ and performs the corresponding step of the second phase for $n$. 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 takes 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, if 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 sets of 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. First, as we have already seen, Presburger formulas compiled from DTTs often force many of their free variables to be equal to 0, and these constraints can often be discovered using simple syntactic analyses. In cases where a variable $x_i$ is statically determined to be equal to 0, we can avoid allocating a fresh variable $y_{i,j}$ for each label $n_i$ that matches the tag of $r_i[S_i]$, because for any tree $t$ accepted by $S$, the constraint on $x_i$ forces the subtree $t(n_i)$ to belong to a different element. Second, if a given label $n_i$ only matches the tag of a single element $r_i[S_i]$, then the constraints added to the Presburger formula in the first phase are $y_{i,j} = x_i$ and $y_{i,j} = 1$. These constraints are equivalent to the single constraint $x_i = 1$, 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 1, there is no need to test the Presburger formula again; we can just test that $t(n_i)$ 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 give intuitive arguments why these choices are sensible; statistics backing up 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 $(x_0 = 1), ([x_0 = 1] \land [x_1 = 0] \land [x_2 = 0]), \ldots, \text{etc.}$, 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 an 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.

**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 two lists of elements requires comparing the elements of the list by absolute position.

Merging identical states is an important 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 translation that merges redundant elements—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. 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 Function 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 D.

Emptiness Our algorithm for deciding emptiness follows the co-inductive approach used in XDuce [7]. 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 \( (\phi, E) \) is empty, we would determine the states in \( E \) that are empty and then test that \( \phi \) is not satisfiable when those elements are constrained to be zero. For deterministic trees, we need to ensure that the 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 \( E \) denotes either a singleton or an infinite set. We then add to \( \phi \) 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 [9]). 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 that the replicas belong to the type.

8. Experimental Results

In this section we present timing data and statistics from experiments run using our system. The first example is a program that takes trees representing address book and validates them against a tree schema loosely based on the vCard standard [8]. The second example is a parser that takes ASCII text decorated with headings and subheadings and transforms it into a tree structure where the nested structure described by the headings is made explicit. 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 is implemented in a tree transformation language [11] where conditionals are evaluated by testing membership of a tree in a type.

We ran the experiments on a 1.4GHz SuSE Linux machine with 2GB of memory; system and user running times were collected using the standard shell utility; statistics about the number of calls to various functions and the hit rates of caches were collected by the caching functions. We ran experiments on a range of inputs, varying in size from a single address book entry or line of text up to several thousand entries or lines, and 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 compiled to a sheaves automaton with 302 states and 1783 total elements; for the text parser, the types produced a sheaves automaton with 105 states and 706 total elements. The inputs to each program were generated by interleave snippets of Joyce’s *Ulysses* into the appropriate syntax: we created address book entries by randomly choosing the fields present in each entry and populating each field with text drawn from the text; for the text parser, we chose lines 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 system is completely unusable with non-trivial inputs.

As the graphs show, neither the incremental algorithm alone nor memoization does as well as all of the optimizations do together. The performance of the basic algorithm alone is predictably bad—on address book entries it plots is nearly vertical; for the text parser, performance was so poor—200s even for very small inputs—that we could not reasonably fit it in the graph. 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.

<table>
<thead>
<tr>
<th>Formulas</th>
<th>Sat</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>7605</td>
<td>98.4%</td>
</tr>
<tr>
<td>Txt</td>
<td>5227</td>
<td>99.6%</td>
</tr>
</tbody>
</table>

These numbers validate the hash-consing and memoization strategies we chose in Section 6. We believe that the low hit rate for the tree hash-cons table in the text parser stems from the way that the system processes the ASCII input into trees: it first explodes the document into a list containing single characters, encoded as trees using the cons-cell representation described previously, and then performs the parsing actions on the tree structure. For a large document—e.g., one with 5,000 80-character lines—the number of distinct trees needed to represent the cons cells represents a significant percentage of the total number of live trees in the heap.

9. Related Work

Automata with counting have been proposed numerous times. Courcelle [5] noticed that the discriminating power of monadic second order logic is weak on unordered trees and suggested adding counting constraints. Later, Dal Zilio, Lugiez, and Meyssonnier [7] and Seidl, Schwentick, Muscholl, and Habermehl [18] 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 [4], 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].

Other formalisms for unordered trees include Ohsaki’s study of AC-closure of regular tree languages [16] and a number of papers on feature logics and automata from the Oz group (for example, [15]). Rounds [17] surveys a range of work on feature logics.

Hague [13] describes an implementation of a membership checker along lines broadly similar to ours. His checker handles multirecursion, as opposed to our 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.

References


A. Presburger Semantics

The value of an expression in a vector is:

\[
\begin{align*}
\text{val}(v, i) & \triangleq i \\
\text{val}(v, x_j) & \triangleq v(j) \\
\text{val}(v, c_1 + e_2) & \triangleq \text{val}(v, c_1) + \text{val}(v, e_2)
\end{align*}
\]

The satisfaction relation is:

\[
\begin{align*}
v \models e_1 = e_2 & \iff \text{val}(v, e_1) = \text{val}(v, e_2) \\
v \models \neg\phi & \iff v \models \exists \phi \\
v \models \phi_1 \land \phi_2 & \iff v \models \phi_1 \land v \models \phi_2 \\
v \models \neg\phi_1 & \iff v \neq \phi_1 \\
v \models \exists_i \phi & \iff \exists n. \langle n, v(0), ..., v(k-1) \rangle \models \phi \text{ where } k = |v|
\end{align*}
\]

B. State Operations

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

\[
\neg\{(X_1, ..., X_k) \setminus \{Y_1, ..., Y_l\}\} \triangleq \{Y_1, ..., Y_l, X_1, ..., X_k\}
\]

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

\[
\{(X_1, ..., X_k) \setminus \{Y_1, ..., Y_l\}\} \cap \{(X_1, ..., X_m) \setminus \{Y_1, ..., Y_n\}\} \triangleq \{(X_1, ..., X_k, X_1, ..., X_m) \setminus \{Y_1, ..., Y_l, Y_1, ..., Y_n\}\}
\]

Negating a compound state intersects the negations of each component 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.

\[
\{C_1, ..., C_k\} \cap \{D_1, ..., D_l\} \triangleq \left\{ (C_1 \cap D_1), ..., (C_1 \cap D_l), ..., (C_k \cap D_1), ..., (C_k \cap D_l) \right\}
\]

C. Horizontal Recursion

This section describes a reduction from the halting problems for two-counter machines to the emptiness problem for DTFs with full horizontal recursion. The reduction closely follows Boneva’s reduction for information trees [3].

A two-counter machine \(M = (Q, q, q_f, \Delta)\) where \(Q\) is a finite set of states, \(q, q_f\) are distinguished initial and final states, the transition relation \(\Delta \subseteq Q \times \mathbb{B} \times \mathbb{B} \times Q\) is a finite set of triples of the form \((q, o, q')\) where \(q\) and \(q'\) are the predecessor and successor states, and \(o\) is an instruction drawn from \(\mathbb{B} = \{\triangleright, \triangleleft, =\}\), which increments or decrements one of the two counters, or tests whether the value of the counter is equal to zero.

A configuration of a two-counter machine is a triple \(q, c_1, c_2\) where \(q \in Q\) and \(c_1, c_2 \in \mathbb{N}\) contain the values of the two registers.

The single-step transition relation, written \(\rightarrow_{M}\) is the binary relation on configurations defined by \((q, c_1, c_2) \rightarrow_{M} (q', c'_1, c'_2)\) iff there exist \((q, o, q') \in \Delta\) and \(i \neq j\) such that \(c'_j = c_j + o\) and either

1. \(o = \triangleright\) and \(c'_j = c_j + 1\);  
2. \(o = \triangleleft\) and \(c'_j = c_j - 1\);  
3. \(o = =\) and \(c'_j = c_j = 0\).

The reflexive and transitive closure of \(\rightarrow_{M}\) is written \(\lambda_{M}\).

The language accepted by a two-counter machine \(M\) is the set of naturals \(L(M) \triangleq \{n \in \mathbb{N} | \exists \Delta \subseteq Q \times \mathbb{B} \times \mathbb{B} \times Q \text{ s.t. } (q, q_f, \Delta) \}\). It is well known that every recursively enumerable set \(S\) one can effectively construct a two-counter machine \(M\) such that \(S = \lambda_{M}\); it follows that the emptiness problem for two-counter machines is undecidable.

Before showing how to encode the transition behavior of a two-counter machine using FTTs, we introduce some refined notation for configurations. In the refined notation, each counter \(n_i\) is represented, implicitly, as the difference of two naturals \(m_i\) and \(x_i\). This allows us to simulate the behavior of \(\triangleright\) and \(\triangleleft\) by incrementing \(n_i\) and \(x_i\) respectively and, in the final construction, to simulate natural number addition by tree concatenation.

We consider the subset of quadruples of naturals where each element \(m_0, x_0, m_1, x_1\) satisfies \(m_0 \geq x_0 \land m_1 \geq x_1\); we write \(\mathbb{N}_2^4\) for the set of all quadruples that satisfy this property. We build configurations \(q, m_0, x_0, m_1, x_1\) and define a single-step transition relation \(\Rightarrow_{M}\) as \((q, m_0, x_0, m_1, x_1) \Rightarrow_{M} (q', m_0', x_0', m_1', x_1')\) if there exists \((q, o, q') \in \Delta\) and \(i \neq j\) such that \(m'_j = m_j + o\) and \(x'_j = x_j\), and either

1. \(o = \triangleright\) and \(m'_j = m_j + 1\);  
2. \(o = \triangleleft\) and \(m'_j = m_j - 1\);  
3. or \(o = =\) and \(m'_j = m_j\) and \(x'_j = x_j\).

By construction, we have that \((q, c_0, c_1) \rightarrow_{M} (q', c'_0, c'_1)\) iff there exist naturals \(x_0, x_1, x'_0, x'_1\) such that \((q, c_0 + x_0, x_0, x_1 + c_1, x_1) \Rightarrow_{M} (q', c'_0 + x'_0, x_0, c'_1 + x'_1, x_1)\). Hence, \(L(M) \neq \emptyset\) iff there exist \(m_0, x_0, m_1, x_1\) such that \((q, m_0, x_0, m_1, x_1) \Rightarrow_{M} (q', m_0', x_0', m_1', x_1')\).

We next will show how to construct a FTT \(T\) such that \(T \not\equiv 0\) iff \(L(M) \neq \emptyset\). For every quadruple \((m_0, x_0, m_1, x_1)\) in \(\mathbb{N}_2^4\) let \(((m_0, x_0, m_1, x_1))\) be the set of trees with \(m_0\) children belonging to \(a*\{\}\), \(x_0\) belonging to \(b*\{\}\), \(m_1\) belonging to \(c*\{\}\), and \(x_1\) belonging to \(d*\{\}\).

Next, let us write down some types that describe the well-formedness constraints on elements of \(\mathbb{N}_2^4\) when encoded as trees. Let

\[
\begin{align*}
T_{ok} & = (a*\{\} + b*\{\} + c*\{\} + d*\{\})* \\
T_{ok1} & = (a*\{\} + b*\{\} + c*\{\} + d*\{\})* \\
T & = T_{ok} + T_{ok1} \\
T_{zero0} & = (a*\{\} + b*\{\} + d*\{\})* + T_{zero0} | T_{ok1} \\
T_{zero1} & = (a*\{\} + b*\{\} + c*\{\} + d*\{\})* + T_{zero0}
\end{align*}
\]

The following equalities hold:

\[
\begin{align*}
T_{ok} & = \{(m_0, x_0, m_1, x_1) | (m_0 \geq x_0 \land m_1 \geq x_1)\} \\
T_{zero0} & = \{(m_0, x_0, m_1, x_1) | (m_0 = x_0 \land m_1 \geq x_1)\} \\
T_{zero1} & = \{(m_0, x_0, m_1, x_1) | (m_0 \geq x_0 \land m_1 = x_1)\}
\end{align*}
\]

For any state \(q \in Q\) let \(\text{Acc}(q) \triangleq \{s \in \mathbb{N}_2^4 | \exists s' \in \mathbb{N}_2^4 (q, s) \Rightarrow_{M} (q, s')\}\). Next we will define a type \(T_q\), for every \(q \in Q\) not equal to \(q_f\), such that \([T_q] = \{\text{Acc}(q)\}\). Let \(T_q = \{(q, r, q') \in \Delta | T_{ok} \cup T_{zero0} \} \) where

\[
\begin{align*}
P_{T_{ok}} & = T_{ok} + T_{ok1} \\
P_{T_{zero0}} & = T_{zero0} + T_{zero1} \\
P_{T_{zero1}} & = T_{zero1} + T_{zero0} \\
P_{T_{ok1}} & = T_{ok1} + T_{zero1}
\end{align*}
\]

Finally, let \(T_{q_f} = T_{ok}\). Then \(L(M) = \emptyset\) iff \(T_{q_f} \cup T_{zero1}\) is empty.
D. Additional Operations

The mutually-recursive functions `empty` and `check` are defined as follows:

```
empty(S, mts, nmts) =
  if S ∈ mts then true
  else if S ∈ nmts then false
  else let mts' = (mts ∪ S) in
    let (is mt, mts'', nmts'') = check(Γ(S), mts', nmts') in
    if is mt then (true, mts'', nmts'') else (false, mts, nmts)
```

```
check((φ, [r_0[S_0], ..., r_k[S_k]]), mts, nmts) =
  let φ' = φ \bigwedge_{i \in 0..k} \{x_j | \text{singleton}(r_j) \land r_j = r_i\} \leq 1
  in for i = 0 to k do
    let (mts', nmts') = (mts, nmts) in
    for each j such that n_i ∈ r_j do
      if mem'(t_i, X_j) then (φ' := φ' \land (y(i,j) = 1)); break
    done
  if \not\models φ' then return false
  if \models φ' ∧ s_i = 1; then return false
```

... 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 1:

```
  let s_i = 0 in
  for each j such that n_i ∈ r_j do
    if empty(X_j, mts, nmts) then s_i := s_i + y(i,j);
  done
  φ' := φ' ∧ s_i = 1;
  if \not\models φ' ∧ 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`.)