LISA : A Specification Language Based on WS2S

Abdelwaheb Ayari[1] David Basin[1] Andreas Podelski[2]

[1]Institut für Informatik, Universität Freiburg,
Am Flughafen 17, 79110 Freiburg, Germany.
Phone: (49) (761) 203-8240 Fax: 203-8242
{ayari, basin}@informatik.uni-freiburg.de

[2]Max-Planck-Institut für Informatik
Im Stadtwald, 66123, Saarbrücken, Germany.
podelski@mpi-sb.mpg.de

Abstract

We integrate two concepts from programming languages into a specification language based on WS2S, namely high-level data structures such as records and recursively-defined datatypes (WS2S is the weak second-order monadic logic of two successors). Our integration is based on a new logic whose variables range over record-like trees and an algorithm for translating datatypes into tree automata. We have implemented LISA, a prototype system based on these ideas, which, when coupled with a decision procedure for WS2S like the MONA system, results in a verification tool that supports both high-level specifications and complexity estimations for the running time of the decision procedure.

Keywords: Specification and Verification, Monadic Second Order Logic and Tree Automata, Feature Logic.

1 Introduction

Motivated by the success of [7], a number of research groups [12, 9, 11, 8] have implemented verification tools based on a decision procedure for the weak monadic second-order logic with one (WS1S) or two (WS2S) successors. Experience, *cf.* [4], indicates that although such tools are powerful aids to verification, their usefulness is limited by two major problems. First, the specification language is low-level; writing specifications in WS2S is an experience akin to programming in assembly language. Second, the complexity of verification is very high; WS2S and related monadic logics are amongst the most expressive decidable logics known, but one pays the price that the decision problem requires non-elementary time, which is a strong practical limitation.

In this paper, we propose an approach that addresses both problems. Our contributions are both theoretical and conceptual. Our theoretical contributions are (1) to define a logic whose formulae define relations between record-like trees ("feature trees"). These relations are encoded by WS2S formulae and, thus, recognized by tree automata. This logic forms the kernel of a specification language whose decision procedure is based on that of WS2S. Our logic comes with its own interpretation domain (*i.e.*, the trees) and interpretation function. This distinguishes it from notation (or macros) whose semantics is defined by syntactic translation (or unfolding). (2) We describe explicitly the direct translation of the part of the logic in which one defines datatypes to deterministic tree automata via alternating tree automata with ε-transitions. We show too that in many cases this is polynomial time computable, and doubly exponential in the worst case.

Our conceptual contribution is to propose an approach that simultaneously addresses the two main limitations of WS2S. The base logic of feature trees, combined with recursive types, provides a formalism for high-level abstract specification. In particular, there is direct support for formalizing record-like data-structures, *e.g.*, accessing subtrees via symbolic keywords (as opposed to consecutive numbers, *à la* WS2S), which are supported in all modern programming languages. Moreover, types provide a handle on the complexity of the decision procedure. Types are directly translated to tree automata (as opposed to indirectly via an initial translation to WS2S formulae) and, as noted above, we bound the complexity of this process.

We have motivated our combination by arguing that it alleviates many of the problems of specification and verification with WS2S. An alternative way to approach and understand our proposal is by comparison with standard programming languages. Early programming languages, like the assembly language, Lisp and Fortran, provided little or no support for datatypes. The user encoded data explicitly in memory. This is analogous to WS2S where the only primitive "type" supported is sets of positions in the binary tree. Hence, the user must laboriously encode other kinds of data, say k-ary trees whose nodes are labeled from some finite set, in terms of unlabeled binary trees. As with programming in assembler, this is possible, but not recommended, and the result falls far short of constituting a comprehensible specification. More advanced programming languages, like ML, provide means of abstractly formalizing data using type declarations. This is important also for structuring the program: These declarations are part of the program and integrate a specification language into a programming language (which is also a specification language) in a controlled and natural way. This is analogous to types in our proposal; types structure the specification and interact with defined predicates by restricting the scope of quantification to elements of the defined types (relativized quantification).

One important way the programming analogy breaks down is that datatypes in our specification language (compared to datatypes in, say, ML) have the same expressiveness as the full language. Both define tree automata and hence both are "WS2S complete" in the sense that they can define any WS2S definable relation. However, they do differ from formulae in the full logic in succinctness (with a non-elementary factor). Said the other way round, a specification using types may trade verbosity with a gain in efficiency. Since we have found that one uses types often in a specification, it is important to give the user a means to control the cost of the usage of types (to at least some degree). Therefore we give the translation procedure of types explicitly. Translation procedures have been proposed for various kinds of regular systems of equations over words and trees [2, 5, 6], but none of these are applicable to type systems as rich as ours. The principle distinction is that our type definitions support conjunctions of types, which is natural in our logic where subtree positions (record-fields) are accessed by atomic formulae. We establish a relationship between such type definitions and alternating top-down tree automata with ε-transitions.

Although we see the contributions in this paper as theoretical and conceptual, their ultimate validation must be empirical. We have partially implemented and tested our ideas. The base logic and type system are implemented in a prototype system called LISA, which is currently coupled with the MONA system of [7]. We have used LISA to carry out several case studies, one of which we report on here.

Related work. Our work is directly inspired by the work of Klarlund and Schwartzbach on a system called FIDO [10]; FIDO is based on the idea that one can encode the values of any fixed finite set and write finite-domain constraints in WS2S. FIDO deserves the credit of being the first approach to integrating programming language concepts with MONA. Our work started with the study of FIDO; we observed that the only data of interest are record-modeling trees and that, under this view, the expressiveness of FIDO's datatypes is the full expressiveness of FIDO. Moreover, FIDO was conceived and explicitly described as a "programming notation"; its semantics was defined by compilation into MONA, the assembly language. We felt that this did not take the programming-language point of view all the way. There, one abstracts away from the underlying machine model (be it jumps or sets of positions) and defines a new calculus/logic with its own semantics; then one can prove the compilation correct. The new logic should be small and simple; it forms the kernel of the language, which itself may be rich in notation. Regarding trees and datatypes in FIDO: These were used mainly to define "domains" for position variables. The type declarations for finite-domain values in FIDO are expressed in LISA by non-recursive datatype definitions (denoting finite sets of trees); this is yet one example indicating the advantage (in conceptual simplicity) of having trees as the interpretation domain.

The idea to use feature trees to model records stems from [1]. The first-order logic over feature trees is decidable in non-elementary time [3, 15]; to our knowledge, no decision procedure has been implemented yet. The basic relation in that logic, besides the unary label relation that corresponds to $l(t, \varepsilon)$, is the direct-subtree relation $f(t, t')$. The addition of that relation to our logic would make the validity test undecidable. It is possible the relation $f(t:T, t':T')$ restricted to non-recursive types T and T', although we do not give any details here.

Lisa, informally We now give an intuition for the way one would express relations over records in our specification language. An example of a record (later modeled as a feature tree) might be:

$$\begin{array}{l} \textit{step(status: initial,} \\ \quad \textit{process}_1\textit{: non-critical,} \\ \quad \textit{next: step(process}_1\textit{: critical,} \\ \quad\quad\quad \textit{process}_2\textit{: critical,} \\ \quad\quad\quad \textit{next: stop))} \end{array}$$

The record consists of identifiers (here: *step, stop, initial, critical*) called labels and of field selectors (here: *status*, $process_1$, $process_2$, *next*) called features. It is a nested record: the value in each record field is itself again a record (possibly without further record fields, i.e., a label only). A label does not fix the record fields below it. Records can be graphically represented as trees whose nodes are labeled by labels and whose arcs are labeled by features.

The record above is a solution of the LISA formula $\varphi(t)$ below, which expresses: every sub-record with both $process_1$ and $process_2$ being critical has the value *stop* in its record field *next.*

$$\varphi(t) \;\equiv\; (\forall p\ (\mathit{critical}(t, p.\mathit{process}_1) \wedge \mathit{critical}(t, p.\mathit{process}_2) \Rightarrow \mathit{stop}(t, p.\mathit{next}))) \qquad (1)$$

The record also satisfies a LISA description of a second kind: It belongs to the defined type *Computation* of all those records that have a record of the same type *Computation* in their record field *next*, or their label is *stop.* The type could be declared by the following LISA type definition.

$$\mathit{Computation} = \mathit{next}\colon \mathit{Computation} \mid \mathit{stop}$$

In Section 5 we will see an example of a LISA formula that combines the two kinds of description; it is precisely this combination of a base logic of feature trees with types, that provides us with both a high-level specification language and complexity guarantees.

2 The Logic of LISA

We now introduce the base logic of LISA, without type definitions. We assume a fixed signature $\langle \mathcal{F}, \mathcal{L} \rangle$ of binary symbols $f \in \mathcal{F}$ called *features* and of binary symbols $l \in \mathcal{L}$ called *labels.* LISA is a two-sorted first-order logic; we assume an infinite set of feature-tree variables $t, s, \ldots$ and of position variables $p, q, r, \ldots$. We reserve the term "LISA formula" to formulae of the LISA logic without free position variables; thus, a LISA formula always defines a relation over feature trees. There are two kinds of atomic formulae.

$l(t,p)$	"the tree t has label l at position p"
$f(p_1,p_2)$	"the position p_2 is the feature f down from the position p_1"

The interpretation domain $\mathcal{D}_{\langle\mathcal{T},\mathcal{L}\rangle} = \langle\mathcal{D}_\mathcal{T}, \mathcal{D}_\mathcal{P}\rangle$ consists of the domain $\mathcal{D}_\mathcal{T}$ of feature trees and the domain $\mathcal{D}_\mathcal{P}$ of positions.

A feature tree $t \in \mathcal{D}_\mathcal{T}$ consists of nodes in $\mathcal{D}_\mathcal{P}$ with labels in $\mathcal{L}$; formally, $t \subseteq \mathcal{D}_\mathcal{P} \times \mathcal{L}$ where $(p, l_1) \in t$ and $(p, l_2) \in t$ implies $l_1 = l_2$. The domain $\mathcal{D}_\mathcal{P}$ of positions is a finite subset of $\mathcal{F}^\star$, *i.e.*, consisting of strings over feature symbols. We write the concatenation of strings p and q as $p.q$; the empty string is ε. We assume that the domain $\mathcal{D}_\mathcal{P}$ is prefix-closed: $p.f \in \mathcal{D}_\mathcal{P}$ implies $p \in \mathcal{D}_\mathcal{P}$.[1] We may picture a feature tree as a tree with nodes labeled in $\mathcal{L}$ and edges labeled in $\mathcal{F}$; no node has two outgoing edges with the same label.

Feature symbols f are interpreted as binary relations f over $\mathcal{D}_\mathcal{P} \times \mathcal{D}_\mathcal{P}$, namely $f(p_1, p_2)$ iff $p_1.f = p_2$. Labels l are interpreted as binary relations l over $\mathcal{D}_\mathcal{T} \times \mathcal{D}_\mathcal{P}$, namely $l(t, p)$ iff $(p, l) \in t$, *i.e.*, the node with the path p is labeled with the symbol l in t. In (1) the following abbreviation was used

$$l(t, p.f_1.f_2.\cdots.f_n) \;\equiv\; \exists q_1 \ldots \exists q_n.\, (f_1(p, q_1) \wedge \ldots \wedge f_n(q_{n-1}, q_n) \wedge l(t, q_n)), \text{ for } 1 \leq n.$$

3 Compiling the LISA logic into WS2S

We next describe the compilation of LISA formulae over feature trees into WS2S formulae. Together with the decision procedure for WS2S, this yields, in some sense, the operational semantics of LISA specifications. The idea is very simple: Feature trees are encoded by tuples of position sets, one set for each label, with the restriction that each position $p \in \mathcal{D}_\mathcal{P}$ occurs in at most one of these sets.

The syntax of WS2S formulae φ is as follows (here $x, y, z, \ldots$ are first-order variables and $X, Y, Z, \ldots$ are monadic second-order variables).

$$\varphi ::= s_1(x) = y \mid s_2(x) = y \mid X(x) \mid \exists x\varphi \mid \exists X\varphi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$$

Of course, other connectives and quantifiers can be added, as is standard in classical logic. The model of WS2S has the domain $\mathcal{D} = \{1, 2\}^\star$ of strings (or paths) p over 1 and 2. Valuations α assign strings $p \in \mathcal{D}$ to first-order variables x and finite sets $P \subseteq \mathcal{D}$ of strings to second-order variables. The denotation of s_1 (written S_1) is concatenation with the letter 1; *i.e.*, for $p \in \mathcal{D}$, $S_1(p) = p.1$. The denotation of s_2 is, analogously, concatenation with 2. The symbol $\in$ is interpreted as the membership relation between positions x and sets of positions X. We write $\{1, 2\}^\star \models \varphi$ to say that the WS2S formula φ is valid: $\{1, 2\}^\star, \alpha \models \varphi$ holds for all valuations α under the given interpretation.

In order to clarify the terminology of "logic over trees" for WS2S, we mention that the pair $\langle\mathcal{D}, \alpha\rangle$ can be encoded by a binary tree t. The set of nodes of t is $\mathcal{D}$. The nodes are labeled by n-tuples of Booleans 0 and 1 where n is the number of free variables (first-order or second-order) of φ. The i-th component of the label of the node p is 1 iff p is equal to or contained in the value of the i-th variable, *i.e.*, $p = \alpha(x)$ or $p \in \alpha(X)$, respectively. One sometimes then writes $\underline{t} \models \varphi$ instead of $\mathcal{D}, \alpha \models \varphi$. In contrast, the LISA logic is a logic over trees in the same direct sense that arithmetic is a logic over numbers.

We next define the (effective) bijection $\lceil\cdot\rceil$ between formulae of LISA and WSkS, the (weak) monadic second-order logic over k successors (which has the domain $\{1, \ldots, k\}^\star$). The equivalence of WSkS to WS2S is standard (*cf.* also Section 4.2). Let the set of labels be $\mathcal{L} = \{l_1, \ldots, l_n\}$ and the set of features be $\mathcal{F} = \{f_1, \ldots, f_k\}$. We assign to each tree variable t of φ the n-tuple of the WSkS second-order variables $P_1^t, \ldots, P_n^t$.[2] We assign to each position variable p of φ the WSkS

[1] We may also require that the domain of a feature tree t is prefix-closed, *i.e.*, $(p.f, l) \in t$ implies $(p, l') \in t$, which amounts to giving a dummy label to "non-labeled" nodes.

[2] In practice, we encode labels using bit patterns over $\lceil log_2(k)\rceil$ second-order variables.

first-order variable p. We set

$$\begin{array}{rcll} \lceil f_j(p,q)\rceil & = & s_j(p) = q & \text{for } \ j = 1,\ldots,k, \\ \lceil l_i(t,p)\rceil & = & P_i^t(p) \wedge \bigwedge_{j\neq i} \neg P_j^t(p) & \text{for } \ i = 1,\ldots,n. \\ \lceil \varphi_1 \wedge \varphi_2 \rceil & = & \lceil \varphi_1 \rceil \wedge \lceil \varphi_2 \rceil & \\ \lceil \neg\varphi \rceil & = & \neg\lceil \varphi \rceil & \\ \lceil \exists x \varphi \rceil & = & \exists x \lceil \varphi \rceil & \\ \lceil \exists X \varphi \rceil & = & \exists P_1^X \ldots P_n^X \lceil \varphi \rceil & \end{array}$$

The following statement expresses the correctness of the compilation of the LISA logic that we have defined above.

Theorem 1 *The* LISA *formula φ is valid over the domain of feature trees if and only if the WSkS formula $\lceil\varphi\rceil$ is valid over the domain of strings; formally,*

$$\mathcal{D} \models_{\text{Lisa}} \varphi \text{ iff } \ \{1,\ldots,k\}^\star \models_{\text{WSkS}} \lceil\varphi\rceil.$$

Proof (sketch): Given a LISA valuation α, we assign it the WSkS valuation $\lceil\alpha\rceil$ where

$$\begin{array}{rcl} \lceil\alpha\rceil(P_i^x) & = & \{p \in \mathcal{D}_\mathcal{P} \mid (p, l_i) \in \alpha(x)\}, \\ \lceil\alpha\rceil(p) & = & \alpha(p). \end{array}$$

We can prove by structural induction over LISA formulae φ that for all valuations α,

$$\mathcal{D}, \alpha \models \varphi \ \text{ iff } \ \{1,\ldots,k\}^\star, \lceil\alpha\rceil \models \lceil\varphi\rceil$$

and, moreover, if $\{1,\ldots,k\}^\star, \beta \models_{\text{WSkS}} \lceil\varphi\rceil$, *i.e.*, the WSkS-valuation β is a solution of $\lceil\varphi\rceil$, then β is of the form $\beta = \lceil\alpha\rceil$.

The statement follows directly from the definitions of the mappings $\lceil\cdot\rceil$ for each atomic LISA formula. The induction steps for $\wedge$ and $\neg$ are evident; the one for $\exists$ follows from the bijectivity of the mapping between solutions of formulae φ of LISA logic and solutions of the corresponding WSkS formulae $\lceil\varphi\rceil$. □

In order to show that LISA is as expressive as WSkS, we need to give the translation from WSkS formulae ψ into LISA formulae. This is a simple embedding. The formulae $j(p) = q$ become $s_j(p,q)$, and the formulae $P(p)$ become $l_0(t_P, p)$ where we have a tree variable t_P corresponding to each second-order variable P. We assign each second-order value $P \subseteq \{1,\ldots,k\}^\star$ the feature tree $t = \{(p, l_0) \mid p \in P\}$.

4 LISA Types

We now build upon the kernel Lisa logic by adding a language of types. Let us begin by considering a simple example: binary trees whose labels come from the set $\{a, b, c, d\}$. In a programming language like ML, we might formalize this as:

$$\begin{array}{lcl} \textit{datatype Tag} & = & a \mid b \mid c \mid d; \\ \textit{datatype BinTree} & = & \textit{bin of } (\textit{Tag} \times \textit{BinTree} \times \textit{BinTree}) \mid \textit{leaf}; \end{array}$$

Types specify constraints on the store of the computer; the types above constrain the contents of members *Tag* to have values among the given labels and members of *BinTree* are trees with a given shape and labeling.

Our type system for Lisa formalizes types as systems of recursively defined constraints over feature trees. We formalize the above types as:

$$\begin{array}{lcl} \textit{Tag} & = & a \vee b \vee c \vee d \\ \textit{BinTree} & = & \textit{bin}(\textit{data} : \textit{Tag}, \textit{left} : \textit{BinTree}, \textit{right} : \textit{BinTree}) \vee \textit{leaf} \end{array} \tag{2}$$

Types T denote sets of trees; hence, they are integrated into the kernel LISA logic as unary predicates over trees. The intended use of types is with relativized quantification. As usual, we write $\forall t : T.\, \varphi$ for $\forall t.\, (T(t) \Rightarrow \varphi)$ and $\exists t : T.\, \varphi$ for $\exists t.\, (T(t) \wedge \varphi)$.

Operationally, types can be integrated in a theorem prover system for WSkS as follows. A type definition T is compiled to an equivalent deterministic bottom-up tree automaton A_T, as we describe below. The standard decision procedure for WSkS [14] works by processing formulae bottom up, replacing subformula by tree automata; it can easily be modified such that when encountering the predicate application $T(t)$ in a formulae, the automaton A_T is used. This approach fits, for example, with the already existing library functionality of the MONA system. There, a user can write libraries of predicates $p(t)$ defined in WSkS, and use $p(t)$ as atomic subformulae in subsequent definitions or for theorem statements (i.e., WSkS formulas). Each such definition is compiled into an automaton A_p, once and for all, which is used in the decision procedure of the MONA system like a pre-compiled module. The system can call an automaton A_T stemming from a type definition in exactly the same way as A_p. So, the difference between the two kinds of automata A_T and A_p lies in the ways they are specified (and not in their use). If the automaton is specified by type definitions, then the compilation has a complexity different from the one of the general WSkS decision procedure. As we will see, it is linear in the practically interesting subcase where all types defined in one type system (which can be seen as forming one library module) denote pairwise disjoint sets.

We now explain the details of this integration. We give the semantics of types and their translation into tree automata. We analyze the complexity of the translation and the possible size of the resulting automata.

4.1 Syntax and Semantics

We assume given a finite set $\mathcal{L}$ of labels and a finite set $\mathcal{F}$ of features; we set $\mathcal{F} = \{f_1, \ldots, f_N\}$. A *type system* $\mathcal{T}$ is a conjunction of *type equations*,

$$\mathcal{T} \;\equiv\; \{T_1 = \theta_1, \ldots, T_m = \theta_m\}$$

between pairwise different *declared types* T_j and *type bodies* θ_j. The syntax of the bodies θ_j is given by the grammar

$$\theta ::= a(f_1 : S_1, \ldots, f_n : S_n) \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2\,. \tag{3}$$

where $a \in \mathcal{L}$ and $f_1, \ldots, f_n \in \mathcal{F}$, and $S_1, \ldots, S_n$ are declared type[3]. In addition, we assume given a type λ whose meaning we later define as the set whose sole member is the empty tree, that we also denote with λ. We employ the syntactic shorthand a for $a(f_1 : \lambda, \ldots, f_N : \lambda)$.

The syntax of LISA formulae is extended with quantification relativized to types. The meaning of type membership "$t \in T$" for a feature tree t is intuitively clear. If T is λ then t is the empty tree, $t = \lambda$. If T is defined by $a(f_1 : S_1, \ldots, f_n : S_n)$, then t is labeled with a at the root and it must have a subtree t_i of type S_i at subtree position f_i, for $1 \le i \le n$, and must have arbitrary trees at the other subtree positions.

Formally, the meaning of the types $T_1, \ldots, T_m$ is given by the least solution σ_0 of the formula $\mathcal{T} \;\equiv\; \bigwedge_{j=1,\ldots,m} T_j = \theta_j$; we set $[\![T_j]\!] = \sigma_0(T_j)$. The formula $\mathcal{T}$ is interpreted over the domain $\mathcal{D}$ of sets of feature trees. A *valuation* σ is a mapping $\sigma : \{T_1, \ldots, T_m\} \to 2^{\mathcal{D}_\mathcal{T}}$. For sets $M_1, \ldots, M_n$, the set $a(f_1 : M_1, \ldots, f_n : M_n)$ is the set of all feature trees t whose roots are labeled with a and at position f_i have a subtree lying in M_i for $i = 1, \ldots, n$; i.e.,

$$a(f_1 : M_1, \ldots, f_n : M_n) = \{t \in \mathcal{D}_\mathcal{T} \mid (\varepsilon, a) \in t, \text{ and } t.f_1 \in M_1, \ldots, t.f_n : M_n\}$$

where $t.f$ for a tree t and a feature f is the direct subtree t' at position f (i.e., $t' = \{(p, a) \mid (f.p, a) \in t\}$). The operators $\wedge$ and $\vee$ over type bodies θ are interpreted as intersection and union, respectively. Valuations are ordered by pointwise inclusion. Solutions of $\mathcal{T}$ are *closed under arbitrary intersection*, as one can easily check by case analysis. Hence, the least solution σ_0 always exists.

[3]We will use T, S, S_1, $S_2 \ldots$ as metavariables ranging over the types $T_1, \ldots, T_m$.

Example (continued): Consider the example above defining the binary-tree type. We are given the set of labels $\mathcal{L} = \{a, b, c, d, bin, leaf\}$ and the set of features $\mathcal{F} = \{data, left, right\}$. In order to shorten notation for trees, we write a for the tree $a(\lambda, \lambda, \lambda)$, *leaf* for the tree $leaf(\lambda, \lambda, \lambda)$, etc. We then may write the meaning of the types *Tag* and *BinTree* as follows.

$$\begin{array}{lcl} [\![Tag]\!] & = & \{a,\ b,\ c,\ d\} \\ [\![BinTree]\!] & = & \{leaf,\ bin(data : a, left : leaf, right : leaf),\ \ldots\} \end{array}$$

4.2 Translating LISA Types into Tree-Automata

Given the type system $\mathcal{T}$ declaring the types $T_1, \ldots, T_m$, we show how to construct a family $A_{\mathcal{T}} = (A_{q_{T_i}})_{T_i \in \mathcal{T}}$ of top-down alternating tree automata [5, 16, 13], such that each automaton $A_{q_{T_i}}$ accepts exactly the trees in $[\![T_i]\!]$. The automaton will be defined so that they only differ in their starting states. In a preliminary step, we transform LISA types into *normalized* types. After we can directly translate normalized types into tree-automata.

4.2.1 Normalized Types

The type λ is a normalized type. Type defined by equations are translated to normalized types as follows.

We restrict the syntax of bodies θ in declarations of normalized types by setting the set of features to $\mathcal{F}_0 = \{l,\ r\}$, where l stands for *left* and r stands for *right.* We require that each label a comes with both features; i.e., the general form $a(f_1 : S_1, \ldots, f_n : S_n)$ is restricted to $a(l : S_1, r : S_2)$, which we shorten to $a(S_1, S_2)$. We furthermore apply the operators $\wedge$ and $\vee$ only to types T (as opposed to general type bodies).

$$\theta ::= a(S_1, S_2) \mid S_1 \wedge S_2 \mid S_1 \vee S_2\,. \tag{4}$$

We translate types to normalized types in three steps. First, we flatten the type declarations. We associate recursively with each part of a disjunction (respectively, conjunction) of a declaration body a new type name and then we replace its occurrences with the associated name. For example, for $T = a \vee b(f_1 : S_1)$, we add new type declarations $R_1 = a$ and $R_2 = b(f_1 : S_1)$ and we replace the above declaration with $T = R_1 \vee R_2$.

Second, we transform the type declaration into one over the set $\mathcal{F}_0 = \{l, r\}$ of only two features. Let $\mathcal{F} = \{f_1, \ldots, f_N\}$ and assume that in every disjunct, all N features occur; we can add conjuncts of the form $f : \top$, where $\top$ denotes the set of all trees, or of the form $f : \lambda$. We introduce a new label d (the dummy label). For a formula of the form $a(f_1 : S_1, \ldots, f_N : S_N)$, we add $O(N^2)$ new normalized types, namely for $1 \leq i \leq N$, we replace each occurrence of the expression $f_i : S_i$ with the expression $S_i^{f_i}$, and we add i new normalized type declarations,

$$\begin{array}{lcl} S_i^{f_i} & = & d(\top, S_{i-1}^{f_i}) \\ & \vdots & \\ S_2^{f_i} & = & d(\top, S_1^{f_i}) \\ S_1^{f_i} & = & d(S_i, \top). \end{array}$$

Thus, we translated $a(f_1 : S_1, \ldots, f_N : S_N)$ into the conjunction $a \wedge S_1^{f_1} \ldots \wedge S_N^{f_N}$. We flatten the conjunction; i.e, we introduce a sequence of type declarations whose body is a binary conjunction.

We must next say in what sense the above translation preserves the meaning of types. We associate general trees in $\mathcal{D}_{\mathcal{T}}$ with binary trees (and vice versa) in the way given above. Let $\tilde{T}_j$ be the normalized version of type T_j, for $j = 1, \ldots, m$. Note that the trees in $[\![\tilde{T}]\!]$ are binary trees whose nodes can have the dummy label. Let $trans([\![\tilde{T}_j]\!])$ be the set of general trees associated with the binary trees that are obtained from the trees in $[\![T_j]\!]$ by replacing the dummy label with any original label. We can prove the following statement by induction over the structure of types.

Lemma 1 *The translation of* LISA *types into normalized types preserves the meaning of types in that* $trans([\![\tilde{T_j}]\!]) = [\![T_j]\!]$.

When we prove later the correctness of the compilation of normalized types into top-down alternating tree automaton, we need the following facts.

Lemma 2

1. $[\![a(S_1, S_2)]\!] = \{a(t_1, t_2) \mid t_1 \in [\![S_1]\!],\ t_2 \in [\![S_2]\!]\}$.
2. $[\![S_1 \vee S_2]\!] = [\![S_1]\!] \cup [\![S_2]\!]$.
3. $[\![S_1 \wedge S_2]\!] = [\![S_1]\!] \cap [\![S_2]\!]$.

Example (continued): Consider the type *BinTree* of example (2). The flattening step leads to:

$$T_1 = bin(data : Tag,\ left : BinTree,\ right : BinTree),\ T_2 = leaf(\lambda, \lambda), \quad \text{and} \quad BinTree = T_1 \vee T_2.$$

In order to transform T_1 into a normalized type, we introduce the new types B, Q, R, U, S_1, S_2, E_1, E_2 and E_3.

$$\begin{aligned}
&T_1 = B \wedge R, \quad B = bin(\top, \top), \quad R = Q \wedge U,\\
&Q = d(Tag, \top), \quad U = S_2 \wedge E_3,\\
&S_2 = d(\top, S_1), \quad S_1 = d(BinTree, \top),\\
&E_3 = d(\top, E_2), \quad E_2 = d(\top, E_1), \quad E_1 = (BinTree, \top).
\end{aligned}$$

4.2.2 Alternating Top-down Tree Automata

Our presentation of alternating top-down tree automata extends Vardi's presentation in [16] to a setting that is more general in two aspects: (1) we have (binary) trees instead of strings, and (2) we allow $\varepsilon-$transitions. Furthermore, we formulate an original acceptance condition which is simpler than the existing ones (for several kinds of automata, alternating or nondeterministic, over trees or over strings).

For a set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulae over X, built using the connectives $\wedge$ and $\vee$. We use $\oplus$ as a symbol that stands for either $\wedge$ or $\vee$. Below we will instantiate X with both the set Q of states and the set Q^2 of pairs of states.

Definition 1 *An* alternating top-down tree automaton with $\varepsilon-$transitions *is a tuple* $A = (\Sigma, Q, q_0, \delta, \delta_\varepsilon, F)$*, where* Σ *is the alphabet set,* Q *is the set of states,* q_0 *the initial state,* F *the set of final states,* δ *is the transition function*

$$\delta : Q \times \Sigma \to \mathcal{B}^+(Q^2)$$

and δ_ε *is the* ε*-transition function*

$$\delta_\varepsilon : Q \to \mathcal{B}^+(Q).$$

Each *unary state expression* $u \in \mathcal{B}^+(Q)$ is a Boolean unary function on trees,

$$u : T_\Sigma \to \{true,\ false\},$$

and each *binary state expression* $b \in \mathcal{B}^+(Q^2)$ is a Boolean binary function on trees,

$$b : T_\Sigma \times T_\Sigma \to \{true,\ false\},$$

defined in the following mutually recursive way (note that a state $q \in Q$ is a special case of a unary state expression).

$$
\begin{array}{rcll}
(u_1 \oplus u_2)(t) & = & u_1(t) \oplus u_2(t) & \\
q(\lambda) & = & \mathit{true} & \text{if } q \in F \\
q(\lambda) & = & \mathit{false} & \text{if } q \notin F \\
q(t) & = & u(t) & \text{if } \delta_\varepsilon(q) = u \\
q(a(t_1,t_2)) & = & b(t_1,t_2) & \text{if } \delta(q,a) = b \\
\langle q_1, q_2\rangle(t_1,t_2) & = & q_1(t_1) \wedge q_2(t_2) & \\
(b_1 \oplus b_2)(t_1,t_2) & = & b_1(t_1,t_2) \oplus b_2(t_1,t_2) &
\end{array}
$$

We say that *an automaton A starting from a state q accepts a tree t* if $q(t)$ evaluates to *true*; i.e.,

$$L_A(q) = \{t \in T_\Sigma \mid q(t) = \mathit{true}\}.$$

In case q is the initial state q_0, we simply say that A *accepts* t; we set $L_A = L_A(q_0)$. Sometimes we say that an automaton recognizes a tuples of sets of trees, referring (explicitly or implicitly) to a tuple of initial states.

4.2.3 From Normalized Types to Alternating Top-down Tree Automata

Given a type system $\mathcal{T}$ defining the types $T_1, \ldots, T_m$, we define a family $A_{\mathcal{T}}$ of top-down alternating tree automata $A_q = (\Sigma,\ Q,\ q,\ \delta,\ \delta_\varepsilon,\ F)$, where the set of states is $Q = \{q_\lambda,\ q_{T_1}, \ldots, q_{T_m}\}$, *i.e.*, to each type T, including λ, we associate the state q_T. The starting state is $q \in Q$, the alphabet Σ is the label set $\mathcal{L}$ and the set of final states is $F = \{q_\lambda\}$. The transition functions are given by:

$$
\begin{array}{ll}
\delta(q_T, a) = \langle q_{S_1}, q_{S_2}\rangle & \text{if } T = a(S_1, S_2) \\
\delta_\varepsilon(q_T) \ = q_{S_1} \vee q_{S_2} & \text{if } T = S_1 \vee S_2 \\
\delta_\varepsilon(q_T) \ = q_{S_1} \wedge q_{S_2} & \text{if } T = S_1 \wedge S_2
\end{array}
$$

The construction is correct in the following sense.

Lemma 3 *The automaton $A_{\mathcal{T}}$ given above recognizes $\mathcal{T}$; i.e., $L_{A_{q_T}} = [\![T]\!]$, for every T in $\mathcal{T}$.*

Proof: by induction over the structure of normalized types. If $T \equiv \lambda$ then $[\![T]\!] = \{\lambda\}$. By construction of $A_{\mathcal{T}}$, T is associated with the state q_λ. By Definition 1, $q_\lambda(\lambda) = \mathit{true}$. Thus, $\lambda \in L_{A_{q_\lambda}}$. Since no transition leads to the state q_λ, λ is the only tree in $L_{A_{q_\lambda}}$.

Suppose $T = a(S_1, S_2)$. We first prove $L_{A_{q_T}} \subseteq [\![T]\!]$. If $t \in L_{A_{q_T}}$, then $q_T(t) = \mathit{true}$. Since q_T can be reached only by applying the transition $\delta(q,a) = \langle q_{S_1}, q_{S_2}\rangle$, t is of the form $a(t_1,t_2)$. Thus, $q_{S_i}(t_i) = \mathit{true}$, and hence $t_i \in L_{A_{q_{S_i}}}$. By induction hypothesis $t_i \in [\![S_i]\!]$ and thus, by Lemma 2, we have $t \in [\![T]\!]$. Conversely, if $t \in [\![T]\!]$, then again by Lemma 2, t is of the form $a(t_1,t_2)$ where $t_i \in [\![S_i]\!]$. By induction hypothesis, $t_i \in L_{A_{q_{S_i}}}$, which means that $q_{S_i}(t_i) = \mathit{true}$. Since $\delta(q_T, a) = \langle q_{S_1},\ q_{S_2}\rangle$, $q_T(t) = q_{S_1}(t_1) \wedge q_{S_2}(t_2) = \mathit{true}$ and thus $t \in L_{A_{q_T}}$.

The cases $T = S_1 \wedge S_2$ and $T = S_1 \vee S_2$ are similar. □

4.2.4 Alternating Top-down into Bottom-up

A nondeterministic top-down tree automaton is a special case of an alternating top-down tree automaton where all state expressions e (occurring in the range of the transition functions) are built up with disjunction $\vee$ only (i.e., without conjunction $\wedge$). This observation forms the basis of our transformation.

Let $A = (\Sigma, Q, q_0, \delta, \delta_\varepsilon, F)$ be a top-down alternating tree automaton with ε-transitions. We will now gradually transform A into an equivalent nondeterministic top-down tree automaton

$A' = (\Sigma, Q', F', \delta', \delta'_\varepsilon, q'_o)$ with ε-transitions. Let $\mathcal{B}^\vee(X)$ and $\mathcal{B}^\wedge(X)$ be the set of disjuncts and conjuncts over the set X, respectively. The main idea is that $Q' = \mathcal{B}^\wedge(Q)$. Let, for a state q and a label a,

$$\bigvee_i \bigwedge_j \langle q^l_{ij}, q^r_{ij} \rangle = DNF(\delta(q, a))$$

where $DNF(e)$ refers to the disjunctive normal form of an expression $e \in \mathcal{B}^+(X)$. We obtain an equivalent automaton if we modify the transition function δ to the function $\delta' : Q \times \Sigma \to \mathcal{B}^\vee((\mathcal{B}^\wedge(Q))^2)$, where

$$\delta'(q, a) = \bigvee_i \langle \bigwedge_j q^l_{ij}, \bigwedge_j q^r_{ij} \rangle.$$

Similarly, we can translate δ_ε into a function $\delta'_\varepsilon : Q \to \mathcal{B}^\vee(\mathcal{B}^\wedge(Q))$ mapping states to disjunctions of conjunctions of states and again obtain an equivalent automaton. We set

$$\delta'_\varepsilon(q) = DNF(\delta_\varepsilon(q)).$$

We next extend δ' and δ'_ε to functions on the set $\mathcal{B}^\wedge(Q)$ of conjunctions of states in the canonical way; i.e.,

$$\delta'(\bigwedge_k q_k, a) = \bigvee_i \langle \bigwedge_j q^l_{ij}, \bigwedge_j q^r_{ij} \rangle$$

where

$$\bigvee_i \bigwedge_j \langle q^l_{ij}, q^r_{ij} \rangle = DNF(\bigwedge_k \delta(q_k, a))$$

and similarly

$$\delta'_\varepsilon(\bigwedge_k q_k) = DNF(\bigwedge_k \delta_\varepsilon(q_k)).$$

We hence, define the nondeterministic top-down tree automaton $A' = (\Sigma, Q', F', \delta', \delta'_\varepsilon, q'_o)$ where $Q' = \mathcal{B}^\wedge(Q)$ is the set of all conjuncts of states $q \in Q$ (thus, if we identify a conjunction with the set of its conjuncts, $Q' = 2^Q$), δ' and δ'_ε are defined above, F' is the set of all conjuncts of states $q \in F$ (or, $F' = 2^F$), and q'_0 is the conjunction q_0 (or, $q_0 = \{q_0\}$).

Theorem 2 *The alternating top-down tree automaton A with ε-transitions and the nondeterministic top-down tree automaton A' with ε-transitions are equivalent; i.e., $L_A = L_{A'}$.*

Proof: by the definition of the acceptance condition; the only ingredient is the fact that $DNF(e)$ and e are logically equivalent for expressions $e \in \mathcal{B}^+(\{true, false\})$. □

The transformation of a nondeterministic top-down tree automaton A' with ε-transitions into a deterministic bottom-up tree automaton is almost standard (see, for example, [6]; we only have to extend the ε-elimination procedure for the string case to the tree case).

Example (continued): Consider the type system $\mathcal{T}$ containing the types *BinTree* and *Tag* of example (2). The family of bottom-up tree automaton $A_\mathcal{T}$ that recognizes $\mathcal{T}$ is $A_\mathcal{T} = (A_q)_{q \in \{Tag, BinTree\}}$ where $A_q = (\Sigma, Q, \{\lambda\}, \delta, \{q\})$ with $\Sigma = \{a, b, c, d, bin, leaf, dummy\}$, $Q = \{Tag, BinTree, \lambda, \top, U, S_1\}$ and δ is defined by the following transitions.

$\langle \lambda, \lambda \rangle \xrightarrow{a,b,c,d} Tag$, $\langle \lambda, \lambda \rangle \xrightarrow{leaf} BinTree$, $\langle Tag, U \rangle \xrightarrow{bin} BinTree$, $\langle BinTree, S_1 \rangle \xrightarrow{dummy} U$, $\langle BinTree, \top \rangle \xrightarrow{dummy} S_1$, $\langle \lambda, \lambda \rangle \xrightarrow{x \in \Sigma} \top$ and $\langle \top, \top \rangle \xrightarrow{x \in \Sigma} \top$.

4.2.5 The Size of the Automaton

How big are the automata obtained by our construction? By the construction given above, for a types system $\mathcal{T}$ with n equations we get a bottom-up tree automaton of the size $2^{0(n)}$. This automaton may be nondeterministic though and determinizing it may give rise to another exponential factor.

This is the upper bound for the general case of LISA types. One can now look for natural syntactic restrictions on type definitions with better bounds. We will define a semantic property that is sufficient to guarantee that there exists a tree automaton recognizing the declared types in which the number of states is linear in the number of types. This property encompasses a large and natural class of types, including those given in this paper and those typically encountered in type declarations.

Let $\mathcal{T}$ be a system of nonempty types and $k = |\mathcal{F}|$. We say $\mathcal{T}$ is *disjoint*, if all its types are pairwise disjoint, *i.e.*, $[\![T]\!] \cap [\![T']\!] = \emptyset$, for all distinct T and T' in $\mathcal{T}$. For any such type system the following holds:

Lemma 4 *If $\mathcal{T}$ is disjoint, then each type $T \in \mathcal{T}$ has a body that can be transformed into a finite disjunction of formulae θ of the form either λ or $a(f_1 : S_1, \ldots, f_n : S_n)$.*

We omit the straightforward but tedious proof of this lemma. A consequence of lemma 4, is the following

Lemma 5 *If $\mathcal{T}$ is disjoint, then there is a deterministic bottom-up k-ary tree automaton $A_{\mathcal{T}}$ that recognizes $\mathcal{T}$ and it has at most $|\mathcal{T}|$ states.*

A deterministic bottom-up k-ary tree automaton A can be transformed into a deterministic bottom-up binary tree automaton A' whose number of states is linear in the number of the states of A. Hence, for a disjoint type system $\mathcal{T}$ we can build a deterministic bottom-up binary tree automaton A that recognizes $\mathcal{T}$ and whose number of states is linear in the size of $\mathcal{T}$.

Although disjointness is semantically defined, we define below a sufficient syntactic characterization for disjointness, that can be checked in linear time.

Definition 2 *Let T be a type with body θ. We define $root(T)$ as the set of all labels that can label the root of an element of T. For $T = \theta$, $root(T) = root(\theta)$ and $root(\theta)$ is defined by,*

$$root(\theta) = \begin{cases} \emptyset & \text{if } \theta \equiv \lambda \\ \{a\} & \text{if } \theta \equiv a \\ \mathcal{L} & \text{if } \theta \equiv f : S \\ root(\theta_1) \cup root(\theta_2) & \text{if } \theta \equiv \theta_1 \vee \theta_2 \\ root(\theta_1) \cap root(\theta_2) & \text{if } \theta \equiv \theta_1 \wedge \theta_2 \end{cases}$$

Lemma 6 *If the types in $\mathcal{T}$ have disjoint roots then $\mathcal{T}$ is disjoint.*

Example (continued): Consider the type system $\mathcal{T}$ containing the types *BinTree* and *Tag* of example (2). The root of *Tag* is the set $root(Tag) = \{a, b, c, d\}$ and the root of the type *BinTree* is the set $root(BinTree) = \{bin, leaf\}$. The roots of both types are disjoint, it follows that $\mathcal{T}$ is disjoint (this fact is obvious for this example). In $\mathcal{T}$ we have three features *tag, left* and *right* that we associate with the projections on the first, second and third component respectively. The deterministic 3-ary bottom-up tree automaton $A = (A_q)_{q \in \{BinTree,\, Tag\}}$ that recognizes $\mathcal{T}$ is defined by $A_q = (\{a, b, c, d, bin, leaf\}, \{\lambda, BinTree, Tag\}, q, \delta, \{\lambda\})$ with

$$\delta = \{\langle \lambda, \lambda, \lambda\rangle \xrightarrow{a,b,c,d} Tag,\ \langle \lambda, \lambda, \lambda\rangle \xrightarrow{leaft} BinTree,\ \langle Tag, BinTree, BinTree\rangle \xrightarrow{bin} BinTree\}.$$

5 Example

We conclude by illustrating some of the features of LISA with an example, which is also considered by Klarlund and Schwartzbach: the correctness of the following toy mutual exclusion algorithm.

```
Turn:  Integer range 1..2 := 1;
Proc(i) is: loop
            a:  Non_critical_section_i
            b:  loop exit when Turn = i; end loop;
            c:  Critical_section_i;
            d:  Turn := i + 1 mod 2
         end loop;
Proc(0) || Proc(1)
```

This algorithm consists of two processes that execute the program, whose lines are numbered a through d. The variable Turn resides in shared memory. We begin our specification by declaring the following datatypes[4].

```
Turn = 1 | 2;
Pc    = a | b | c | d;
State   = state(pc1:Pc, pc2:Pc, turn:Turn);
Comp = node(val: State, next:Comp) | done;
```

The type Comp formalizes trees that represent sequences of states. Each state has features pointing to the program counters (each storing a program line) and the value of the Turn variable.

Not all elements of Comp represent valid executions. Hence we define Lisa predicates[5] that further constrain the members of Comp.

The predicate Init describes the start state of both processes.

```
pred  Init(X::State)  = a(X,pc1) & a(X,pc2) & 1(X,turn);
```

The state X satisfies Init, if 1) it is a tree of type State and 2) its subtrees at the positions pc1 and pc2 are labeled with the program line a and the subtree at the position turn is labeled with the turn value 1.

For the transition function, we first declare how a process can execute. The predicate Step1 is a large conjunction that describes the possible transitions.

```
pred Step1(X::State, Y::State) =
   a(X,pc1) => b(Y,pc1) & X.turn == Y.turn &
   b(X,pc1) => ((1(X,turn)=>c(Y,pc1))&(2(X,turn)=>b(Y,pc1))&(X.turn==Y.turn))&
   c(X,pc1) => d(Y,pc1) & X.turn == Y.turn &
   d(X,pc1) => a(Y,pc1) & 2(Y,turn) &
   X.pc2 == Y.pc2;
```

For example, the first conjunct states that the first process can advance from state X where pc1 is at line a, to state Y where pc1 is at line b, and the value of the turn variable remains unchanged. The predicate Step2 is declared similarly.

```
pred Step2(X::State, Y::State) =
   a(X,pc2) => b(Y,pc2) & X.turn == Y.turn &
   b(X,pc2) => ((1(X,turn)=>b(Y,pc2))&(2(X,turn)=>c(Y,pc2))&(X.turn==Y.turn))&
   c(X,pc2) => d(Y,pc2)  & X.turn == Y.turn &
   d(X,pc2) => a(Y,pc2) & 1(Y,turn) &
   X.pc1 == Y.pc1;
```

[4]We use the ASCII syntax: | for $\vee$, & for $\wedge$, => for the implication and == for the equality.

[5]A predicate (defined with pred in LISA) is equivalent to (*i.e.*, syntactically interchangeable with) a formula in one or several free variables (tree variables in the case of LISA).

A transition of the system is a step by either process (*i.e.*, we assume an interleaving semantics).

```
pred Trans(X::State, Y::State) =  Step1(X,Y)  | Step2(X,Y) ;
```

Finally, a computation is `Valid` when the first state satisfies `Init` and all pairs stand in the transition relation.[6]

```
pred Valid(X::Comp) = Init(X.val) &
          all p: (node(X,p) & node(X,next.p)) =>Trans(X.p.val, X.p.next.val);
```

Given these definitions, we now define what mutual exclusion means: no two processes are simultaneously in their critical section, line `c`.

```
pred Mutex(X::Comp) =  all p: ~(c(X,p.val.pc1)& c(X,p.val.pc2));
```

With this, we can formalize the question of whether all valid computations have the mutual exclusion property.

```
  all  X::Comp: Valid(X) => Mutex(X);
```

LISA is an implemented system and is integrated with MONA. For this example, LISA translates the defined predicates and types and produces about 2 pages of formulae in WS2S. These are input to MONA, which takes 2 seconds to process the result, and to report

```
  Formula with free variables X0 X1 X2 X3 X4 X5 is a tautology
  Dfa has 1 states and 1 BDD-nodes
```

thereby verifying that the program does indeed enforce mutual exclusion.

This example illustrates how types and LISA formulae interact. There is a natural decomposition of specifications into types, which express simple properties about the shapes and values of data, and predicates (e.g., `Mutex`), which express more complicated constraints. Again, recall that both are equally expressive; the tradeoff is one of conciseness and complexity of the translation. We can also express predicates like `Valid` and `Mutex` as types, but the results would be rather cumbersome (a blowup in size traded in for a better complexity bound with respect to the larger types) and not very natural.

6 Discussion and Further Work

Up to now, it has been taken for granted that a specification and verification tool based on WS2S also algorithmically had to be based on WS2S semantically. This is the first time that a programming language (*i.e.*, with its own semantics) is built on top of WS2S.

Our contribution is also to provide a new way for users to estimate the complexity of their specification, namely the size of the resulting compiled tree automaton. Type compilation has a double-exponential upper bound that in many practical cases is linear. Hence, the more a system can be specified with types, the more accurately one can bound the complexity. Of course, for the part of the system specified in the kernel logic, non-elementary blowups (an exponential blowup with each quantifier alternation) are, of course, still possible. There are interesting practical tradeoffs here: as noted in the introduction, all tree-automata can be described using types and hence it follows that there are certain problems that can be more naturally (and, in particular, with a non-elementary savings of space) described in the kernel Lisa logic.

LISA may be the first programming language compiled into tree automata via WS2S, but it is certainly not the last. The language is still primitive, and one can think of extensions that further help in structuring specifications. For example, we plan to replace the primitive `pred` construct with more powerful means of decomposing specifications into modules that support abstraction and specification reuse.

[6]`X.f` is the tree such that `a(X.f,p)` *iff* `a(X,p.f)` for all labels a and for all positions p.

Acknowledgments. This work would not have been possible without the preliminary work by Nils Klarlund and Michael Schwartzbach on FIDO. We thank both of them for fruitful discussions. Part of the implementation of LISA was done while the first author was invited by BRICS where he received technical support and encouragement. We thank Harald Ganzinger and Luca Viganò for comments on this draft as well as the anonymous CSL referees.

References

[1] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science (TCS)*, 122:263–283, 1994.

[2] D. Arden. Delayed logic and finite state machines. In *Proceedings of the 2nd Annual Symposium on Switching Circuit Theory and Logical Design*, pages 133–151, 1961.

[3] R. Backofen and G. Smolka. A complete and recursive feature theory. Research Report RR-92-30, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany, July 1992.

[4] D. A. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Computer-Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 31–41. Springer-Verlag, 1995.

[5] J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.

[6] F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, Budapest, 1984.

[7] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996.

[8] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. Mosel: A flexible toolset for monadic second–order logic. In *Proc. CAV'97, 9th Int. Conference on Computer Aided Verification*, Haifa (Israel), June 1997. Springer Verlag.

[9] Kesten, Maler, Marcus, Pnueli, and Shahar. Symbolic model checking with rich assertional languages. In *Proc. CAV'97, 9th Int. Conference on Computer Aided Verification*, Haifa (Israel), June 1997. Springer Verlag.

[10] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in automated verification based on trace abstractions. In *Formal Systems Specification; The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*. Springer, 1996.

[11] Z. Manna and the STeP group. Step: The stanford temporal prover. In *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *LCNS*, pages 793–794. Springer-Verlag, 1995.

[12] F. Morawietz and T. Cornell. On the recognizibility of relations over a tree definable in a monadic second order tree description language. Research Report SFB 340-Report 85, Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft, Februar 1997.

[13] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.

[14] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, August 1967. Published by Springer-Verlag NY Inc.

[15] S. Vorobyov. An improved lower bound for the elementary theories of trees. In *Automated Deduction – CADE-13*, volume 1104 of *LNAI*, pages 275–287. Springer-Verlag, 1996.

[16] M. Y.Vardi. An automata-theoretic approach to linear-temporal logic. *In Logics for Concurrency: Structure versus Automata. LNCS*, 1043:238–266, 1996.