Deciding the First-Order Theory of an Algebra of Feature Trees with Updates*

Nicolas Jeannerod and Ralf Treinen

Univ. Paris Diderot, Sorbonne Paris Cité, IRIF, UMR 8243, CNRS, Paris, France
`nicolas.jeannerod@irif.fr` `ralf.treinen@irif.fr`

Abstract. We investigate a logic of an algebra of trees including the update operation, which expresses that a tree is obtained from an input tree by replacing a particular direct subtree of the input tree, while leaving the rest unchanged. This operation improves on the expressivity of existing logics of tree algebras, in our case of feature trees. These allow for an unbounded number of children of a node in a tree.
We show that the first-order theory of this algebra is decidable *via* a weak quantifier elimination procedure which is allowed to swap existential quantifiers for universal quantifiers. This study is motivated by the logical modeling of transformations on UNIX file system trees expressed in a simple programming language.

1 Introduction

Feature trees are trees where nodes have an unbounded number of children, and where edges from nodes to their children carry names such that no node has two different outgoing edges with the same name. Hence, the names on the edges can be used to select the different children of a node. Feature trees have been used in constraint-based formalisms in the field of computational linguistics (e.g. [14]) and constrained logic programming [1, 15]. This work is motivated by a different application of feature trees: they are a quite accurate model of UNIX file system trees. The most important abstraction in viewing a file structure as a tree is that we ignore multiple hard links to files. Our mid-term goal is to derive, using symbolic execution techniques, from a shell script a logical formula that describes the semantics of this script as a relation between the initial file tree and the one that results from execution of the script.

Feature tree logics have at their core basic constraints like $x[f]y$, expressing that y is a subtree of x accessible from the root of x via feature f, and $x[f]\uparrow$, expressing that the tree x does not have a feature f at its root node. This is already sufficient to describe some tree languages that are useful in our context. For instance, the script consisting of the single command `mkdir /home/john`, which creates a directory `john` under the directory `home`, succeeds on a tree if the tree satisfies the formula $\exists d.(r[\texttt{home}]d \wedge d[\texttt{john}]\uparrow)$, which expresses that `home`

* This work has been partially supported by the ANR project CoLiS, contract number ANR-15-CE25-0001.

is a subdirectory of the root, which does itself *not* have a subdirectory `john`. We ignore here the difference between directories and regular files, as well as file permissions.

Update Constraints. In order to describe the effect of executing the above script we need more expressivity. A first idea is to introduce an update constraint $y \doteq x[f \mapsto z]$, which states that the tree y is obtained from the tree x by setting its child f to z, and creating the child when it does not exist. Using this, the semantics of `mkdir /home/john` could be described by

$$\exists d, d', e. \, (in[\texttt{home}]d \wedge d[\texttt{john}]\uparrow \wedge out \doteq in[\texttt{home} \mapsto d'] \wedge d' \doteq d[\texttt{john} \mapsto e] \wedge e[\emptyset])$$

Here, $e[\emptyset]$ expresses that e is an empty directory. Note that this formula, by virtue of the update constraint, expresses that any existing directories under `home` different from `john` are not touched.

Programming constructs translate to combinations of logical formula. For instance, if $\phi_p(in, out)$, resp. $\phi_q(in, out)$ describe the semantics of script fragments p and q, then their composition is described by $\exists t.(\phi_p(in, t) \wedge \phi_q(t, out))$. The reality of our use case is more complex than that due to the hairy details of error handling in shell scripts [10], and is up to future work.

Formulas with more complex quantification structure occur when we express interesting properties of scripts. For instance, p and q are equivalent if

$$\forall in, out. \, (\phi_p(in, out) \leftrightarrow \phi_q(in, out))$$

Debian requires in its policy [7] so-called maintainer scripts to be idempotent, which can be expressed for a script p as

$$\forall in, out. \, (\phi_p(in, out) \leftrightarrow \exists t.(\phi_p(in, t) \wedge \phi_p(t, out))$$

Since we are interested in verifying these kinds of properties on scripts we need a logic of feature trees including update constraints, and which enjoys a decidable first-order logic.

Related Work. The first decidability result of a full first-order theory of *Herbrand trees* (*i.e.*, based on equations $x = f(x_1, \ldots, x_n)$) is due to Malc'ev [13], this result has later been extended by [6, 12]. A first decidability result for the first-order theory of *feature trees* was given for the logic FT [1], which comprises the predicates $x[f]y$ and $x[f]\uparrow$, by [4]. This was later extended to the logic CFT [15], which in addition to FT has an *arity constraint* $x[F]$ for any finite set F of feature symbols, expressing that the root of x has precisely the features F, in [3,5]. Note that in these logics one can only quantify over trees, not over feature symbols. The generalization to a two-sorted logic which allows for quantification over features is undecidable [16], but decidability can be recovered if one restricts the use of feature variables to talk about existence of features only [17]. All these decidable logics of trees have a non-elementary lower bound [18]. The case of a feature logic with update constraints was open up to now.

Choosing the Right Predicates. The difficulty in solving update constraints stems from the fact that an update constraint involves three trees: the original tree, the final tree and the sub-tree that gets grafted on the original tree.

There are no symmetries between these three arguments, and a conjunction of several update constraints may become quite involved. Our approach to handle this rather complex update constraint is to replace it by a more elementary constraint system which is based on the classical $x[f]y$, and the new *similarity constraint* $x \sim_f y$. The latter constraint expresses that x and y have the same children with the same names, except for the name f where they may differ. This system has the same expressive power as update constraints since on the one hand $z \doteq x[f \mapsto y]$ is equivalent to $x \sim_f z \wedge z[f]y$, and on the other hand $x \sim_f y$ is equivalent to $\exists z, v.(z \doteq x[f \mapsto v] \wedge z \doteq y[f \mapsto v])$. In order to simplify these constraints one needs the generalization $x \sim_F y$ where F is a finite set of features. For each set of features F, similarities $\sim_F$ are equivalence relations, which is very useful when designing simplification rules, and these relations have useful properties, like $(x \sim_F y \wedge x \sim_G y) \leftrightarrow x \sim_{F \cap G} y$ and $(x \sim_F y \wedge y \sim_G z) \rightarrow x \sim_{F \cup G} z$.

Eliminating Quantifiers. Our theory of feature trees does not have the property of quantifier elimination in the strict sense [9]. This is already the case without the update (or similarity) constraints, as we can see in the following example: $\exists x.(y[f]x \wedge x[g]\uparrow)$. This formula means that there is a tree denoted by x such that y points to x through the feature f, and that x does not have the feature g. A quantifier elimination procedure would have to conserve this information about the global variable y. This situation is not unusual when designing decision procedures. There are basically two possible remedies: the first one is to extend the logical language by new predicates which express properties which otherwise would need existential quantifiers to express. This approach of achieving the property of quantifier elimination by extension of the logical language is well-known from Presburger arithmetic, it was also used in [3, 4].

However, in the case of feature tree logics, the needed extension of the language is substantial and requires the introduction of *path constraints.* For instance, the above formula would be equivalent to the path constraint $y[f][g]\uparrow$ stating that the variable y has a feature f pointing towards a tree where there is no feature g. Unfortunately, this extension entails the need of quite complex simplification rules for these new predicates.

The alternative solution is to our knowledge due to [13] and consists in exploiting the fact that certain predicates of the logic behave like functions. This solution was also used in [6] for Herbrand trees. When switching to feature trees this solution becomes quite elegant [17], the above formula would be replaced by $\neg y[f]\uparrow \wedge \forall x.(y[f]x \rightarrow x[g]\uparrow)$ stating that y has a feature f (by $\neg y[f]\uparrow$) and that for each variable x such that y points towards x via f (in fact, there is only one), x has no feature g. The price is that existential quantifiers are not completely eliminated but swapped for universal ones. This is, however, sufficient, since one can now apply this transformation to a formula in prenex normal form, and successively reduce the number of quantifier eliminations.

Structure of this Paper. We summarize some notions from logic that will be used in the rest of the paper in Section 2. Our model of trees as well as the syntax and semantics of our logic are defined formally in Section 3. The quantifier elimination procedure in given in Section 4. We conclude in Section 5. Proofs are only sketched, full proofs are to be found in the companion technical report [11].

2 Preliminaries

We assume logical conjunction and disjunction to be associative and commutative, and equality to be symmetric. For instance, we identify the formula $x \doteq y \wedge (x[f] \uparrow \vee x[g]z)$ with $(x[g]z \vee x[f] \uparrow) \wedge y \doteq x$.

The set of free variables of a formula ϕ is written $\mathcal{V}(\phi)$. We write $\phi\{x \mapsto y\}$ for the formula obtained by replacing in ϕ all free occurrences of x by y. We write $\tilde{\exists}\phi$ for the existential closure $\exists\mathcal{V}(\phi).\phi$, and similarly $\tilde{\forall}\phi$ for $\forall\mathcal{V}(\phi).\phi$.

A *conjunctive clause with existential quantifiers*, or in short *clause*, is either $\perp$, or a finite set of literals prefixed by a string of existential quantifiers. Note that such a clause may still contain free variables, that is we do *not* require all its variables to be quantified. If $\exists X.(a_1 \wedge \ldots \wedge a_n)$ is such a clause, then we can partition its set of literals $c = g_c \cup l_c$ such that g_c contains all the literals of c that contain no variable of X, and l_c the set of literals of c that contain at least one variable of X. We have the following logical equivalence:

$$\models (\exists X.c) \leftrightarrow (g_c \wedge \exists X.l_c)$$

We call (g_c, l_c) the *decomposition* of $\exists X.c$. g_c is the *global part* and l_c the *local part* of c, X is the set of *local variables* and $\mathcal{V}(\exists X.c) \supseteq \mathcal{V}(g_c)$ the set of *global variables.*

A *disjunctive normal form* (dnf) is a finite set of clauses, all of which are different from $\perp$.

A formula is in *prenex normal form* (pnf) if it is of the form $Q_1x_1 \ldots Q_nx_n.\phi$ where ϕ is quantifier-free, and where the Q_i are existential or universal quantifiers. If all Q_i are $\exists$ (resp. $\forall$) then the formula is called a Σ_1-formula (resp. Π_1-formula).

$A \rightsquigarrow B$ denotes the set of partial functions from the set A to the set B with a finite domain. The domain of a partial function f is written $\mathtt{dom}(f)$. The complement of a set is written X^c. We write $X \setminus Y$ for $\{x \in X \mid x \notin Y\}$.

3 A Logic for an Algebra of Trees with Similarities

3.1 Decorations

In addition to what has been said in the introduction, our model of feature trees also has information attached to the *nodes* of the trees. In our application to UNIX filesystems, these could be records containing the usual file attributes like various timestamps and access permission bits, owner and group, and so on. This work abstracts from the details of the information attached to tree nodes: we take the definition of node decorations, and the pertaining logic as a parameter. We hence assume given an arbitrary set $\mathcal{D}$ of *decorations*.

We assume given a set D of predicate symbols for decorations, and an interpretation $\mathcal{D}$ for D with universe $\mathcal{D}$. These predicate symbols are of course assumed to be disjoint from the predicate symbols that will be introduced in Subsection 3.3. We also require that D contains a binary predicate $x \not\cong y$ expressing the *disequality* of two information items.

We also assume that we have a quantifier elimination procedure for $\mathcal{D}$: we can compute for any Σ_1 formula ψ over the language D, possibly with free variables, a quantifier-free formula $D\text{-elim}(\psi)$ that is equivalent in $\mathcal{D}$ to ψ and has the same free variables. Furthermore, we can decide for any closed and quantifier-free D-formula whether in holds in $\mathcal{D}$.

3.2 Feature Trees

We assume given an infinite set $\mathcal{F}$ of *features*. The letters f, g, h will always denote features.

The set $\mathcal{FT}$ of *feature trees* is inductively defined as

$$\mathcal{FT} = \mathcal{D} \times (\mathcal{F} \rightsquigarrow \mathcal{FT})$$

Here, the case of a partial function with empty domain serves as base case of the induction. Hence, this amounts to saying that a feature tree is a finite unordered tree where nodes are labeled by decorations, and edges are labeled by features. Each node in a feature tree has a finite number of outgoing edges, and all outgoing edges of a node carry different names. We write $\hat{t}$ for the decoration of the root node of t and we write $\vec{t}$ for its mapping at the root, i.e. $t = (\hat{t}, \vec{t})$. Our notion of equality on trees is *structural equality*, i.e. $t = s$ iff $\hat{t} = \hat{s}$ and $\vec{t} = \vec{s}$, that is $\texttt{dom}(\vec{t}) = \texttt{dom}(\vec{s})$ and $\vec{t}(f) = \vec{s}(f)$ for every $f \in \texttt{dom}(\vec{t})$. Examples of feature trees are given in Figure 1.

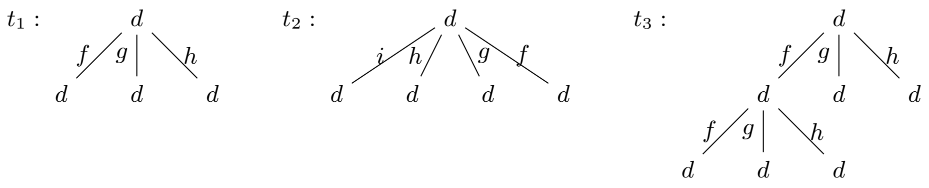

Fig. 1. Examples of Feature Trees. $d \in \mathcal{D}$ is some arbitrary node decoration.

For the reasons explained in the introduction, our logical language does not contain $y \doteq z[x \mapsto f]$ but the simpler $x \sim_F y$ for any *finite* set $F \subseteq \mathcal{F}$. If $F \subseteq \mathcal{F}$ then we say that t *is similar to* s *outside* F, written $t \sim_F s$, if for all $f \in F^c = \mathcal{F} \setminus F$ we have that

– either $f \notin \texttt{dom}(\vec{t}\,) \cup \texttt{dom}(\vec{s}\,)$
– or $f \in \texttt{dom}(\vec{t}\,) \cap \texttt{dom}(\vec{s}\,)$, and $\vec{t}\,(f) = \vec{s}\,(f)$.

In other words, t and s are similar outside F if they have precisely the same children except maybe for the features in F.

3.3 Constraints and their Interpretation

The set of predicate symbols (or atomic constraints) of our logic is

$x \doteq y$	Equality	$A(x_1 \dots x_n)$	Decoration predicate $A \in D$
$x[f]y$	Feature f from x to y	$x[f]\uparrow$	Absence of feature f from x
$x[F]$	Fence constraint	$x \sim_F y$	Similarity outside F

In fences and similarities, the sets F are finite. We will use the usual syntactic sugar and write $x \not\doteq y$ for $\neg(x \doteq y)$, and $x \not\sim_F y$ for $\neg(x \sim_F y)$. As with equality, we consider similarity predicates to be symmetric, that is we identify $x \sim_F y$ with $y \sim_F x$.

We have one model which has as universe the set $\mathcal{FT}$. As usual, we use the same symbol $\mathcal{FT}$ for the model and for its universe. The predicate symbols are interpreted as follows, where ρ is a valuation of the free variables of the formula in the model $\mathcal{FT}$:

$$\begin{array}{ll}
\mathcal{FT}, \rho \models x \doteq y & \text{iff } \rho(x) = \rho(y) \\
\mathcal{FT}, \rho \models x[f]y & \text{iff } f \in \texttt{dom}(\overrightarrow{\rho(x)}) \text{ and } \overrightarrow{\rho(x)}(f) = \rho(y) \\
\mathcal{FT}, \rho \models x[f]\uparrow & \text{iff } f \notin \texttt{dom}(\overrightarrow{\rho(x)}) \\
\mathcal{FT}, \rho \models x[F] & \text{iff } \texttt{dom}(\vec{x}\,) \subseteq F \\
\mathcal{FT}, \rho \models x \sim_F y & \text{iff } \rho(x) \sim_F \rho(y) \\
\mathcal{FT}, \rho \models A(x_1, \dots, x_n) & \text{iff } \mathcal{D}, (\lambda x_i.\widehat{\rho(x_i)}) \models A(x_1, \dots, x_n)
\end{array}$$

Example 1. Let ρ be the valuation $[x \to t_1, y \to t_2, z \to t_3]$ for the trees defined in Figure 1. The following formulas are satisfied in $\mathcal{FT}, \rho$:

$$z[f]x, \quad x[i]\uparrow, \quad x[\{f, g, h, i\}], \quad x \sim_{\{i\}} y$$

Similarity constraints are actually only of interest in case of an infinite set of features. In case of a finite set $\mathcal{F}$, the similarity constraint could already be expressed in the logic FT that was mentioned in Section 1:

$$x \sim_G y \Leftrightarrow \bigwedge_{f \in \mathcal{F} \setminus G} ((x[f]\uparrow \wedge y[f]\uparrow) \vee \exists z(x[f]z \wedge y[f]z))$$

Note the difference between our *fence* constraint, which states an upper bound on the root features of a tree, and the *arity* constraint of [3, 15] which states a precise set of root features of a tree. Both are equivalent, since one can express a fence F as a disjunction of all the arities that are subsets of F. Reciprocally, in our logic, we can express that x has arity F as $x[F] \wedge \bigwedge_{f \in F} \neg x[f]\uparrow$.

Also note that decoration predicates behave in $\mathcal{FT}$ as in $\mathcal{D}$:

Proposition 1. *If ψ is a formula using only symbols of D then*

$$\mathcal{FT}, \alpha \models \psi \qquad \Leftrightarrow \qquad \mathcal{D}, \lambda x.\widehat{\alpha(x)} \models \psi$$

4 Quantifier Elimination

4.1 Clashing Clauses

We say that a clause c that is not $\bot$ *clashes* if one of the patterns of Figure 2 matches (modulo associativity and commutativity of $\wedge$) a sub-clause $c' \subseteq c$.

C-Cycle	$x_1[f_1]x_2 \wedge \ldots \wedge x_n[f_n]x_1$	$(n \geq 1)$
C-Feat-Abs	$x[f]y \wedge x[f]\uparrow$	
C-Feat-Fen	$x[f]y \wedge x[F]$	$(f \notin F)$
C-Neq-Refl	$x \not\doteq x$	
C-NSim-Refl	$x \not\sim_F x$	

Fig. 2. Clash patterns

Remark that C-Cycle is a clash since our model allows for finite feature trees only, the other clash cases should be obvious.

Lemma 1. *If a clause c clashes then $\mathcal{FT} \models (c \rightarrow \bot)$.*

4.2 Positive Clauses with Local Variables

As a preparation for the general case we first consider only one single clause $\exists X.(a_1 \wedge \ldots \wedge a_n)$ containing only positive atoms, prefixed by some existential quantifiers.

S-Eq	$\exists X, x.(x \doteq y \wedge c)$	$\Rightarrow$	$\exists X.c\{x \mapsto y\}$	$(x \neq y)$
S-Feats	$\exists X, z.(x[f]y \wedge x[f]z \wedge c)$	$\Rightarrow$	$\exists X.(x[f]y \wedge c\{z \mapsto y\})$	
				($y \neq z$, and if $z \in \mathcal{V}_o$ then $y \in \mathcal{V}_o$)
S-Feats-Glob	$\exists X, x.(x[f]y \wedge x[f]z \wedge c)$	$\Rightarrow$	$\exists X, x.(x[f]y \wedge y \doteq z \wedge c)$	$(y, z \notin X)$
S-Sims	$x \sim_F y \wedge x \sim_G y \wedge c$	$\Rightarrow$	$x \sim_{F \cap G} y \wedge c$	
P-Feat	$x \sim_F y \wedge x[f]z \wedge c$	$\Rightarrow$	$x \sim_F y \wedge x[f]z \wedge y[f]z \wedge c$	$(f \notin F)$
P-Abs	$x \sim_F y \wedge x[f]\uparrow \wedge c$	$\Rightarrow$	$x \sim_F y \wedge x[f]\uparrow \wedge y[f]\uparrow \wedge c$	$(f \notin F)$
P-Fen	$x \sim_F y \wedge x[G] \wedge c$	$\Rightarrow$	$x \sim_F y \wedge x[G] \wedge y[F \cup G] \wedge c$	
P-Sim	$x \sim_F y \wedge x \sim_G z \wedge c$	$\Rightarrow$	$x \sim_F y \wedge x \sim_G z \wedge y \sim_{F \cup G} z \wedge c$	
				(if $\bigcap_{(y \sim_H z) \in c} H \not\subseteq F \cup G$)

Fig. 3. Transformation rules for the positive case. Existential quantifiers are only written were relevant. Rule S-Feats is parameterized by a set $\mathcal{V}_o$ of variables.

In this subsection and the following, we will use transformation rules as the ones in Figure 3. These rules describe transformations that map a clause to a formula (in this subsection the resulting formula is also a clause, but that will no longer be the case in the next subsection). We say that such a rule *left* $\Rightarrow$ *right* applies to a clause c if:

1. The pattern *left* matches the complete clause c modulo associativity and commutativity of conjunction.
2. The side conditions of the rule, if any, are met.
3. The transformation yields a formula which is *different* from c.

If c is a clause and r a transformation rule then we write $r(c)$ for the formula obtained by applying r to c.

Each of the rules of Figure 3 describes an equivalence transformation in the model $\mathcal{FT}$. Equation elimination (S-EQ) is a logical equivalence. S-FEATS implements the fact that features are functional. This rule is parameterized by a set $\mathcal{V}_o$ of variables that will be the set of variables (local or global) of the input clause. The variable replacement is $\mathcal{V}_o$-oriented in the sense that we never replace a variable in $\mathcal{V}_o$ by a variable outside $\mathcal{V}_o$. S-FEAT-GLOB is similar to S-FEAT for the case that y and z are both global variables. S-SIMS allows us to contract multiple similarities between the same pair of variables into one. P-FEATS, P-ABS and P-FEN propagate constraints along a similarity, taking into account the index of the similarity. Finally, P-SIM is a kind of transitivity of similarity, where we take care not to add a similarity which is subsumed by already existing similarities.

The propagations play two important roles in that system. First, they move information, possibly leading to a clash. This is the case in the following example where a fence moves through similarities to clash with a feature constraint:

$$\begin{array}{lllll}
 & x[f]v & \wedge\ x \sim_{\{g\}} y & \wedge\ \underline{y \sim_{\{h\}} z} \wedge \underline{z[\varnothing]} \\
\text{P-FEN} & x[f]v & \wedge\ \underline{x \sim_{\{g\}} y} \wedge \underline{y[\{h\}]} & \wedge\ y \sim_{\{h\}} z \wedge z[\varnothing] \\
\text{P-FEN} & x[f]v \wedge x[\{g,h\}] & \wedge\ x \sim_{\{g\}} y \wedge y[\{h\}] & \wedge\ y \sim_{\{h\}} z \wedge z[\varnothing]
\end{array}$$

Second, they take information from local variables and move it to global variables. This mechanism is at the core of the elimination of existential quantifications, the idea being that once all the propagations took place, all interesting information is explicit in the global part, and we can hence drop the local part.

$$\begin{array}{lrl}
 & y[h]\uparrow & \wedge\ \exists z.(\underline{x \sim_{\{f\}} z} \wedge \underline{z \sim_{\{g\}} y}) \\
\text{P-SIM} & \underline{y[h]\uparrow} \wedge \underline{x \sim_{\{f,g\}} y} & \wedge\ \exists z.(x \sim_{\{f\}} z \wedge z \sim_{\{g\}} y) \\
\text{P-ABS} & x[h]\uparrow \wedge y[h]\uparrow \wedge x \sim_{\{f,g\}} y & \wedge\ \exists z.(x \sim_{\{f\}} z \wedge z \sim_{\{g\}} y)
\end{array}$$

The following function computes a normal form with respect to the rules of Figure 3:

```
function normalize-positive($c$: positive clause)
  $\mathcal{V}_o$ := $\mathcal{V}(c_1)$ where $c = \exists X.c_1$
  while $c$ does not clash and some rule $r$ of Fig 3 applies to $c$:
    $c$ := $r(c)$
  return($c$)
```

Lemma 2. *For a positive clause c, the function* `normalize-positive` *terminates and yields a positive clause that is equivalent in $\mathcal{FT}$ to c.*

Given a quantifier-free clause c, we define D-part(c) as the conjunction of all D-literals of c.

Lemma 3. *Let the function* `normalize-positive` *return a clause $\exists X.c$ that does not clash and (g_c, l_c) be its decomposition. Let $d = D\text{-}elim(\exists X.D\text{-}part(c))$. If c contains no atom $x[f]y$ with $x \notin X$ and $y \in X$ then*

$$\mathcal{FT} \models \tilde{\forall}((\exists X.c) \leftrightarrow (g_c \wedge d))$$

Actually, both lemmas are special cases of the forthcoming Lemmas 5 and 6 of Section 4.3.

Lemma 3 can serve for quantifier elimination in the positive case, at least when there is no feature constraint from a global variable to a local one. We will see in Section 4.4 what can be done if this is not the case.

4.3 General Clauses with Local Variables

In case of clauses containing both positive and negative literals we have to consider transformation rules that introduce negations or disjunctions. However, our rules will continue to take a single clause as input. As a consequence, we have to transform the result obtained by a transformation into disjunctive normal form. We assume given a function `dnf` that takes a formula without universal quantifiers and containing only positive occurrences of existential quantifiers, and returns an equivalent dnf that does not contain any clashing clauses. This can be achieved by using a standard dnf transformation and then purging all clashing clauses, or alternatively by applying the clash rules on the fly.

Syntactic Sugar. In the transformation rules to be presented below we will use several abbreviations that allow us to write the rules more concisely. First we have

$$x\langle F\rangle \ := \bigvee_{f\in F} \exists z.x[f]z$$

where $F \subset \mathcal{F}$ is a finite set. This formula states that x has *at least one* feature in the set F, it can be seen as a dual to the fence constraint $x[F]$ which states that x has *at most* the features in the set F. Note that $x\langle F\rangle$ introduces a disjunction, so introducing such a formula requires the result to be put into dnf.

The formula $x \neq_f y$ states that x and y differ at feature f, that is either one of them has f and the other one does not, or their children at f are different. The formula $x \neq_F y$ generalizes this to a finite set $F \subset \mathcal{F}$, stating that x and y differ at at least one of the features in F.

$$\begin{aligned} x \neq_f y &:= \exists z'.(x[f]\uparrow \wedge y[f]z') \vee \exists z.(x[f]z \wedge y[f]\uparrow) \\ &\quad \vee \exists z,z'.(x[f]z \wedge y[f]z' \wedge (z \not\approx z' \vee z \not\sim_{\varnothing} z')) \\ x \neq_F y &:= \bigvee_{f\in F} x \neq_f y \end{aligned}$$

We use $(z \ncong z' \vee z \nsim_{\varnothing} z')$ instead of $(z \neq z)$ to denote a difference between two variables in order to avoid problems with the termination. These formulas introduce disjunctions. They also introduce negated similarities at some newly created children of x and y, so we have to take care in the termination proof when these formulas are introduced by a transformation.

$$\begin{array}{llcl}
\textsc{R-NEq} & x \neq y \wedge c & \Rightarrow & (x \ncong y \vee x \nsim_{\varnothing} y) \wedge c \\
\textsc{R-NFeat} & \neg x[f]y \wedge c & \Rightarrow & (x[f]\uparrow \vee \exists z.(x[f]z \wedge (y \ncong z \vee y \nsim_{\varnothing} z))) \wedge c \\
\textsc{R-NAbs} & \neg x[f]\uparrow \wedge c & \Rightarrow & \exists z.x[f]z \wedge c \\
\textsc{R-NFen-Fen} & x[F] \wedge \neg x[G] \wedge c & \Rightarrow & x[F] \wedge x\langle F \setminus G\rangle \wedge c \\
\textsc{R-NSim-Sim} & x \sim_F y \wedge x \nsim_G y \wedge c & \Rightarrow & x \sim_F y \wedge x \not\simeq_{F\setminus G} y \wedge c \\
\textsc{R-NSim-Fen} & x[F] \wedge x \nsim_G y \wedge c & \Rightarrow & x[F] \wedge \big(\neg y[F \cup G] \vee x \not\simeq_{F\setminus G} y\big) \wedge c \\
\textsc{E-NFen} & x \sim_F y \wedge \neg x[G] \wedge c & \Rightarrow & x \sim_F y \wedge (\neg x[F \cup G] \vee x\langle F \setminus G\rangle) \wedge c \\
 & & & \hfill (F \not\subseteq G) \\
\textsc{E-NSim} & x \sim_F y \wedge x \nsim_G z \wedge c & \Rightarrow & x \sim_F y \wedge \big(x \nsim_{F\cup G} z \vee x \not\simeq_{F\setminus G} z\big) \wedge c \\
 & & & \hfill (F \not\subseteq G)
\end{array}$$

Fig. 4. Replacement and Enlargement rules for the general case. $\ncong$ is the disequality of decorations.

New Rules. Figure 4 extends the previously defined set of rules by adding several replacement rules and two enlargement rules. First, we have R-NEq, R-NFeat and R-NAbs that eliminate occurrences of the negated constraints $x \neq y$, $\neg x[f]y$ and $\neg x[f]\uparrow$ respectively. Since no other rule introduces any of these negated constraints we can ignore these two negated constraints in the rest of the section.

Then we have three rules that combine a positive with a negative constraint. R-NFen-Fen applies to the case where we have both a positive fence F and a negated fence G for x. We simplify this by keeping the positive fence F, and replacing the negative fence by saying that x must have a feature that is in F (since that is all it can have), but not in G. Similarly, R-NSim-Sim applies when we have between x and y both a positive similarity except in F, and a negated similarity except in G. We simplify this by keeping the positive similarity, and replacing the negated similarity by stating that x and y differ at a feature that is in F (since these are the only features where they may differ) but not in G. Finally, R-NSim-Fen applies when we have a fence F for x, and a negated similarity with y except in G. Note that for any F and G, $G^c = (F \cup G)^c \cup (F \setminus G)$. Hence, the negated similarity is equivalent to saying that either y has a feature outside $F \cup G$, which is the only possibility to have a difference with x outside $F \cup G$ since x has already fence F, or the difference is in the finite set $F \setminus G$.

Finally, we have the two enlargement rules E-NFen and E-NSim. Their sole purpose is to ensure (by enlarging the negated fence or the index of a negated similarity) that the rules in Figure 5 can be applied when we have a similarity in conjunction with a negated fence or a negated similarity. The correctness proof of these rules is similar to the three previous rules. In fact, the similarity between x and y is not needed for the correctness of these two rules and serves only for the termination proof since the requirement of a context $x \sim_F y$ excludes arbitrary enlargements.

$$\begin{array}{llcll}
\text{P-NFEN} & x \sim_F y \wedge \neg x[G] \wedge c & \Rightarrow & x \sim_F y \wedge \neg x[G] \wedge \neg y[G] \wedge c & (F \subseteq G) \\
\text{P-NSIM} & x \sim_F y \wedge x \not\sim_G z \wedge c & \Rightarrow & x \sim_F y \wedge x \not\sim_G z \wedge y \not\sim_G z \wedge c & (F \subseteq G)
\end{array}$$

Fig. 5. Propagation rules for the general case.

The two rules in Figure 5 may propagate a negated fence or a negated similarity through a similarity. In fact, if x and y coincide outside F and $F \subseteq G$, then x and y also coincide outside G. Hence, if x has a feature outside G then so does y (P-NFEN), and if x differs from z at some feature outside G then so does y (P-NSIM).

We define the set of rules R_1 as the union of all the transformation rules of Figures 3 and 4, and R_2 as the set of the two transformation rules of Figure 5.

```
function normalize(c: clause)
  $d$ := $\{c\}$
  $\mathcal{V}_o$ := $\mathcal{V}(c_1)$ where $c = \exists X.c_1$
  while exists $c \in d$ to which some rule $r \in R_1$ applies
    $d$ := $(d \setminus \{c\})$ ∪ dnf($r(c)$)
  while exists $c \in d$ to which $r \in R_2$ applies
    $d$ := $(d \setminus \{c\})$ ∪ $\{r(c)\}$
  return($d$)
```

The function `normalize` normalizes first by rule set R_1, and then by rule set R_2. This decomposition is necessary to ensure termination. It also makes sense since application of rules R_2 conserves normal forms with respect to R_1.

Lemma 4. *The output of* `normalize` *is a dnf where each conjunction is in normal form for $R_1 \cup R_2$.*

Proof (sketch). We have to prove that the application of one of the rules in R_2 to a normal form with respect to R_1 does not produce a redex for any of the rules in R_1. Assume, for instance that the application of P-NFEN to c introduces a redex of R-NFEN-FEN. This means that the negative fence constraint introduced for y will react with a positive fence constraint (for y) that was already present in c. Since c is in normal form with respect to P-FEN, x must have a fence constraint in c. This yields a contradiction since then c is not in normal form with respect to R-NFEN-FEN. The other cases are similar (details can be found in [11]).

Lemma 5. *The function* `normalize`*, when applied to a clause c, terminates and yields a dnf d such that $\mathcal{FT} \models \tilde{\forall}(c \leftrightarrow d)$.*

Proof (sketch). Equivalence of c and d follows from the fact that each transformation rule is an equivalence in $\mathcal{FT}$. Termination is shown by defining a well-founded order on clauses such that each rule transforms a clause into a set of stricter smaller clauses. The termination order on dnf formulas is the multiset extension [8] of this order.

This order is a lexicographic order over twelve different measures that decrease with the applications of the rules. We can for instance handle the rules R-NEQ, R-NFEAT and R-NABS first by saying that they decrease the number of negated equalities, feature constraints or absences. Since nothing introduces those literals, this is already a good start.

The first main difficulty in finding that order comes from the fact that all the propagation rules are trying to saturate the clause. A good measure that decreases with them is then the set of all possible atoms that are not in the formula. For P-FEAT, for instance: $\{(x[f]y) \mid x, y \in \mathcal{V}(c); f \in \mathcal{F}(c); (x[f]y) \notin c\}$. That would make a good measure if $\mathcal{V}(c)$ could not increase with the application of other rules such as R-NSIM-FEN. We have thus to handle these other rules first, which leads us to another main difficulty.

The second main difficulty comes from the negated similarities. Indeed, while all other literals may only move "horizontally" following the similarities, negated similarities may "descend" in the constraint, creating variables and feature constraints if needed. It is not obvious when it will stop, and in particular to find a bound on the number of variables introduced.

Let us consider the following example constraint and one of its reduction paths (that is, the reduction may create several branches in the dnf, and we take only the one we are interested in):

$$\begin{array}{lll}
 & x_0[f]x_1 \wedge x_1[f]y_0 & \wedge \underline{x_0[\{f\}]} \wedge x_1[\{f\}] \wedge \underline{x_0 \not\sim_{\varnothing} y_0} \\
 & \multicolumn{2}{c}{\text{By R-NSIM-FEN:}} \\
\exists y_1, \underline{z_1}. & \underline{x_0[f]x_1} \wedge x_1[f]y_0 \wedge \underline{x_0[f]z_1} \wedge y_0[f]y_1 & \wedge x_0[\{f\}] \wedge x_1[\{f\}] \wedge z_1 \not\sim_{\varnothing} y_1 \\
 & \multicolumn{2}{c}{\text{By S-FEATS:}} \\
\exists y_1. & x_0[f]x_1 \wedge x_1[f]y_0 \wedge y_0[f]y_1 & \wedge x_0[\{f\}] \wedge x_1[\{f\}] \wedge x_1 \not\sim_{\varnothing} y_1
\end{array}$$

In two rules, we created a new variable y_1, and removed a negated similarity just to put it again somewhere else. Note in particular that R-NSIM-FEN can still apply, because x_1 has now a fence and a negative similarity. In fact, if, instead of two, we take a number n of variables x_i, we can extend that example into one that always doubles the number of variables.

The key to our solution to this problem is that rules that make negative similarities descend, thus introducing feature constraints and new variables, need some "fuel", which is the presence of positive fences or similarities. We define the *original variables* as the variables that were in the clause at the begining of `normalize`. Then, we show that

1. the number of original variables cannot grow;
2. there are never feature constraints from non-original variables towards original ones;
3. the positive fences and similarities can only be present on original variables.

It remains the problem that negative similarities can descend. At some point, they will necessarily go too deep and leave the area where the original variables may live. By doing so, they loose the positive fences and similarities that they need to keep descending, and the process stops.

The full proof, including the lemmas corresponding to the points (1), (2) and (3), the definition of the measures and the technical details can be found in [11].

This is also where we make use of the quantifier elimination procedure for D-formulas. Given a quantifier-free clause c, we define D-part(c) as the conjunction of all D-literals of c.

Lemma 6. *Let the function* `normalize` *return a dnf which contains a clause $\exists X.c$. Let (g_c, l_c) be the decomposition of c, and $d = D\text{-}elim(\exists X.D\text{-}part(c))$. If c contains no atom $x[f]y$ with $x \notin X$ and $y \in X$ then*

$$\mathcal{FT} \models \tilde{\forall}((\exists X.c) \leftrightarrow g_c \wedge d)$$

The proof can be found in [11].

We call a clause *normalized* when it is an element of a dnf returned as result of function `normalize`.

4.4 Quantifier Elimination

In order to eliminate a block of existential quantifiers from a clause we apply iteratively the following rule:

$$\text{F{\scriptsize EAT}-F{\scriptsize UN}} \quad \exists X, x.(y[f]x \wedge c) \quad \Rightarrow \quad \neg y[f]\uparrow \wedge \forall x.\,(y[f]x \rightarrow \exists X.\,(y[f]x \wedge c)) \qquad (y \notin X, y \neq x)$$

This rule follows the idea of [13], and was already applied to feature constraints in [17]. The correctness of this transformation is shown by the following chain of equivalences in the model $\mathcal{FT}$:

$$\begin{array}{ll}
\exists X, x.(y[f]x \wedge c) & \\
\exists x.(y[f]x \wedge \exists X.c) & \text{since } x, y \notin X \\
\neg y[f]\uparrow \wedge \forall x.\,(y[f]x \rightarrow \exists X.c) & \text{since features are functional} \\
\neg y[f]\uparrow \wedge \forall x.\,(y[f]x \rightarrow \exists X.\,(y[f]x \wedge c)) &
\end{array}$$

The last step is very important, because it ensures that, if $y[f]x \wedge c$ is in normal form, then the right part of the implication is also in normal form. This will be important for the function defined below.

The function `switch` defined below iterates this replacement for all local variables x that occur in the form $y[f]x$ where y is not local: the function applies the transformation Feat-Fun, and then recursively applies itself on the result. When there remains no more feature constraint $y[f]x$ from a global variable to a local variable in the normalized clause c, we meet the hypotheses of Lemma 6. We then return the conjunction of the global part g_c of the normalized clause, and of the D-part of the local part l_c from which we have eliminated the block of existential quantifiers.

```
recursive function switch($c$: normalized clause)
  if $\exists X, x.(y[f]x \wedge c')$ matches $c$ and $y \notin X$:
    return($\neg y[f] \uparrow \wedge\ \forall x.(y[f]x \rightarrow$ switch($\exists X.y[f]x \wedge c'$)))
  else:
    ($g_c$,$l_c$) := decomposition($c$)
    $d$ := $D$-elim($\exists X.D$-part($l_c$))
    return($g_c \wedge d$)
```

Example 2. When given the following formula

$$\exists v, w.(y[f]v \wedge v[f]w \wedge w[f]z \wedge w[\{f, g\}] \wedge y \sim_{\varnothing} z)$$

the function `switch` returns

$$\neg y[f] \uparrow \wedge \forall v.(y[f]v \rightarrow (\neg v[f] \uparrow \wedge \forall w.(v[f]w \rightarrow (w[f]z \wedge y \sim_{\varnothing} z))))$$

Lemma 7. *Given a normalized clause c, `switch(c)` terminates and yields a formula ψ such that*

1. $\mathcal{FT} \models \tilde{\forall}(c \leftrightarrow \psi)$;
2. $\mathcal{V}(\psi) \subseteq \mathcal{V}(c)$;
3. *ψ contains no existential quantifiers and only positive occurrences of universal quantifiers;*
4. *If $\mathcal{V}(c) = \emptyset$ then ψ is quantifier-free.*

We can now write a function that transforms a Σ_1 formula into an equivalent Π_1 formula. For this we assume given a function `pnf` that transforms any formula into its prenex normal form.

```
function solve($p$: $\Sigma_1$ formula)
  let $\exists X.q$ = $p$ where $q$ is quantifier-free
  $d$ := dnf($q$)
  $dt$ := $\bigvee_{c \in d}$ normalize($\exists X.c$)
  $u$ := $\bigvee_{c \in dt}$ switch($c$)
  return(pnf($u$))
```

Finally, the function `decide` takes a formula in prenex normal form and returns an equivalent (in $\mathcal{FT}$) formula without any quantifiers. If Q is a string of quantifiers, then $\overline{Q}$ is the string of quantifiers obtained from Q by changing $\exists$ into $\forall$ and vice-versa. For instance, $\overline{\exists x \forall y \exists z} = \forall x \exists y \forall z$.

```
recursive function decide($p$: pnf)
  if $p$ is quantifier-free:
    return($p$)
  else if $p$ is $Q.\exists X.q$
    where $q$ quantifier-free, $Q$ does not end on $\exists$:
    return(decide($Q$. solve($\exists X.q$)))
  else if $p$ is $Q.\forall X.q$
    where $q$ quantifier-free, $Q$ does not end on $\forall$:
    return($\neg$ decide($\overline{Q}$. solve($\exists X.\neg q$)))
```

Theorem 1. *Given a formula p in prenex normal form,* `decide(p)` *terminates and yields a formula q such that*

- $\mathcal{FT} \models \tilde{\forall}(p \leftrightarrow q)$
- $\mathcal{V}(q) \subseteq \mathcal{V}(p)$
- *q is a Π_1 formula, and quantifier-free in case $\mathcal{V}(p) = \emptyset$*

Proof (sketch). Termination follows from the fact that at each call to `decide`, the number of quantifier *alternations* in the pnf decreases.

If we apply `decide` to a closed formula, we hence obtain an equivalent (in $\mathcal{FT}$) formula that contains no free variables and no quantifiers. Since the only tree-terms are variables, we have obtained formula of the language D, for which we can decide by assumption validity in $\mathcal{D}$.

Corollary 1. *The first order theory of $\mathcal{FT}$ is decidable.*

5 Conclusion

We have presented a quantifier elimination procedure for a first-order theory of feature trees with similarity constraints. Since update constraints can be expressed by similarity and feature constraints, this implies in particular that the first-order theory of feature trees with update constraints is decidable.

Our model of feature trees is in several respects an abstraction of UNIX file systems [2]. First, real file systems make a distinction between different kinds of files (directories, regular files, various kinds of device files). This distinction is omitted here just for the sake of presentation. More importantly, real file systems are not really trees as they allow for multiple paths from the root to regular files (which must be sinks), and they provide for symbolic links. Since extending the model by any of these may lead to undecidability of the full first-order theory we might have to look for smaller fragments which are sufficient for our application to the symbolic execution of scripts.

Acknowledgments. The idea of investigating update constraints on feature trees originates from discussions with Gert Smolka a long time ago. We would like to thank the anonymous reviewers for their useful remarks and suggestions, and the members of the CoLiS project for numerous discussions on tree constraints and their use in modeling tree operations, in particular Claude Marché, Kim Nguyen, Joachim Niehren, Yann Régis-Gianas, Sylvain Salvati, and Mihaela Sighireanu.

References

1. Aït-Kaci, H., Podelski, A., Smolka, G.: A feature-based constraint system for logic programming with entailment. Theor. Comput. Sci. **122**(1–2), 263–283 (Jan 1994)
2. Bach, M.: The Design of the UNIX Operating System. Prentice-Hall (1986)
3. Backofen, R.: A complete axiomatization of a theory with feature and arity constraints. Journal of Logic Programming **24**(1&2), 37–71 (Jul/Aug 1995)

4. Backofen, R., Smolka, G.: A complete and recursive feature theory. Theor. Comput. Sci. **146**(1–2), 243–268 (Jul 1995)
5. Backofen, R., Treinen, R.: How to win a game with features. Information and Computation **142**(1), 76–101 (Apr 1998)
6. Comon, H., Lescanne, P.: Equational problems and disunification. Journal of Symbolic Computation **7**, 371–425 (1989)
7. Debian Policy Mailing List: Debian Policy Manual, version 4.1.3. Debian (Dec 2017), `https://www.debian.org/doc/debian-policy/`
8. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Communication of the ACM **22**(8), 465–476 (1979)
9. Hodges, W.: Model Theory, Encyclopedia of Mathematics and its Applications, vol. 42. Cambridge University Press (1993)
10. Jeannerod, N., Marché, C., Treinen, R.: A formally verified interpreter for a shell-like programming language. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments. LNCS, vol. 10712, pp. 1–18. Springer, Heidelberg, Germany (Jul 2017)
11. Jeannerod, N., Treinen, R.: Deciding the first-order theory of an algebra of feature trees with updates (extended version) (Jan 2018), `https://hal.archives-ouvertes.fr/hal-01760575`
12. Maher, M.J.: Complete axiomatizations of the algebras of finite, rational and infinite trees. In: LICS. pp. 348–357. IEEE, Edinburgh, Scotland, UK (Jul 1988)
13. Malc'ev, A.I.: Axiomatizable classes of locally free algebras of various type. In: Benjamin Franklin Wells, I. (ed.) The Metamathematics of Algebraic Systems: Collected Papers 1936–1967, chap. 23, pp. 262–281. North Holland (1971)
14. Smolka, G.: Feature constraint logics for unification grammars. Journal of Logic Programming **12**, 51–87 (1992)
15. Smolka, G., Treinen, R.: Records for logic programming. Journal of Logic Programming **18**(3), 229–258 (Apr 1994)
16. Treinen, R.: Feature constraints with first-class features. In: Borzyszkowski, A.M., Sokołowski, S. (eds.) Mathematical Foundations of Computer Science. LNCS, vol. 711, pp. 734–743. Springer (Aug/Sep 1993)
17. Treinen, R.: Feature trees over arbitrary structures. In: Blackburn, P., de Rijke, M. (eds.) Specifying Syntactic Structures, chap. 7, pp. 185–211. CSLI Publications and FoLLI (1997)
18. Vorobyov, S.: An improved lower bound for the elementary theories of trees. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE'96. LNCS, vol. 1104, pp. 275–287. Springer, New Brunswick, NJ (Jul/Aug 1996)