Université Paris Diderot - Paris 7
Master Parisien de Recherche en Informatique

Constraint Solving and Decision Problems of First-Order Theories of Concrete Domains

Ralf Treinen

Version 2.5 of December 10, 2014

Author's address:

Ralf Treinen
Laboratoire Preuves, Programmes et Systèmes (PPS)
Université Paris Diderot - Paris 7
Case 7014
75205 PARIS Cedex 13
France

Email: `treinen@pps.jussieu.fr`
WWW: `http://www.pps.jussieu.fr/~treinen`

Contents

Chapter 1

Preface

The present lecture notes are interested in the following kind of problems: Given a first-order language L and a first-order structure A over L, is there an algorithm for deciding the validity in A of a first-order formula over L? If yes, what is its complexity? If not, can we find a decision algorithm for a certain fragment of the theory? In particular we are interested in structures whose domains are the elementary data structures of computer science: numbers, strings and trees.

Besides giving some important results in the field (a summary of the results mentioned in these lectures notes is given in Appendix A) our aim is to present techniques to construct decision algorithms for first-order theories on the one hand and techniques to prove undecidability of first-order theories on the other hand. The next chapter gives an introduction to the use of constraints in computer science and to the different constraint systems that we are interested in. Chapter 3 gives some preliminaries on notations and basic results, and Chapter 4 defines precisely the different constraint structures that will be investigated in the succeeding chapters. These two chapters should be consulted as needed. With Chapter 5 we start the exposition of constraint solving methods with syntactical and deterministic methods for simplifying quantifier-free constraints. This is extended in Chapter 6 to non-deterministic methods which rely on the exploitation of a search tree. In Chapter 7 we investigate how the methods of syntactical constraint simplification can be extended to the elimination of quantifiers in order to decide complete first-order theories. Chapter 8 addresses how constraint resolution methods based on the syntactical simplification of formulas can cope with possible non-termination of the simplification process. An important alternative to syntactic simplification of formulas is the translation of constraints into automata, this allows us to employ known techniques and results from automata theory for the resolution of constraints (Chapter 9). Chapter 10 discusses the special case of monadic structures, and in Chapter 11 we sum up the two big classes of decision methods we have presented, and attempt to compare the two approaches. Finally, Chapter 12 presents methods for proving undecidability of first-order theories.

The first version of these lecture notes have been prepared for the *DEA Programmation: Sémantique, Preuves et Langages*, which was a one-year preparatory program for prospective PhD students jointly organized by several universities in the Paris region. Edition of these notes continues while the former DEA program has evolved into a second-year master course of the *Master Parisien de Recherche en Informatique*. It is currently integrated in the course on *Automated Deduction*.

My thanks go to all colleagues and students who have spotted errors in earlier versions or have otherwise made suggestions for improvements: Hubert Comon, Stéphanie Delaune, Felix Klaedtke, Étienne Lozes, Assia Mahboubi, Claude Marché, Arthur Milchior, and Fabien Renaud.

Chapter 2

Introduction

This chapter aims to explain what constraints are in general, what they are used for, and what the interesting questions about constraints are. This chapter is heavily based on [JT01].

2.1 Constraints

Tuples of values are central to many concepts of computer science. For instance, the fundamental object of the semantics of an imperative programming language is a tuple of values the components of which are indexed by the names of variables. Such a tuple represents the *state* of the memory at a certain point of the execution of a program, for instance $\{x \mapsto 2, y \mapsto 3\}$ represents the state of the memory that assigns the value 2 to the variable x and the value 3 to the variable y. Imperative programs contain *instructions* that describe in a schematic way the passage from one state to another. For instance, an instruction like

```
if x > 0 then y := y-1 else x := x+y
```

describes a mapping on states. This mapping maps the state $(x \mapsto 1, y \mapsto 42)$ to $(x \mapsto 1, y \mapsto 41)$, and the state $(x \mapsto -5, y \mapsto 3)$ to $(x \mapsto -2, y \mapsto 3)$. See for instance [Win93, Gun92, LS87] for a formal definition of the semantics of imperative programming languages.

Tuples of values are also at the base of the semantics of *logic programming*, a programming paradigm proposed in the early 70s by Alain Colmerauer [CKRP73] et Robert Kowalski [Kow74]. In the case of logic programming the tuples are labeled by so-called *predicates*. The tuple of the n values $t_1, \ldots, t_n$ and labeled with the predicate P is usually written $P(t_1, \ldots, t_n)$, and called a *ground atom*. The intended meaning of such an atom is the assertion that the tuple $(t_1, \ldots, t_n)$ has the property P. In the case of classical logic programming, values are terms without variables, so-called *ground* terms. In general, terms may contain variables. An expression $P(t_1, \ldots, t_n)$ in which the terms t_i are not necessarily ground is called an *atom*. A (not necessarily ground) atom A can be seen as a description of all the ground atoms that are an instance of A. Logic programs contain *rules* which describe how to deduce new ground atoms from already known atoms. For instance,

$$P(f(x, y), g(x, y)) \leftarrow R(y, x), Q(h(x), g(x, y))$$

allows us to deduce, for instance, the ground atom $P(f(a, b), g(a, b))$ from the ground atoms $R(b, a)$ and $Q(h(a), g(a, b)$. A very important difference with imperative programming is that the semantics of logic programming is about the (potentially infinite) *set* of all ground atoms that can be deduced using the rules of a program. There are different but equivalent ways of defining formally the semantics of a logic program (see, for instance, [Llo84, Apt90]).

Generally speaking *constraints* are expressions of a formal language designed to talk about sets of tuples of values. Sets of tuples of values are central to many formal systems of computer science: We have already seen above that set of tuples of values (that is, sets of closed atoms) are at the heart of the semantics of logic programming. In the context of imperative programming, sets of tuples of values (that is, sets of states) are used when *analyzing* imperative programs since in this case one has to consider the set of all states that can be attained by the execution of a program. These and other applications of constraints will be explained in some more detail in Section 2.2.

Since constraints denote sets of tuples of values, computing with constraints gives us a means to effectively manipulate sets of tuples of values. We will see in Section 2.2 which are the computational services required for important applications.

One important idea that emerged in the 1990 decade was to factor out the constraints from computational mechanisms that so far had used constraints implicitly. This leads to a two-tired architecture, consisting of

- A language of data descriptions, and means to compute with data descriptions. Predicate logic is the natural choice as a framework for expressing data descriptions, that is *constraints*.
- A computation formalism which operates on constraints by a well-defined set of operations. The choice of this set of operations typically is a compromise between what is desirable for the computation mechanism, and what is feasible for the constraint system.

Predicate Logic is the natural framework for expressing constraints since it provides for the following features which are important for most of the constraint-based formalisms:

1. first-order variables to denote values; these variables will be shared with the formalism which uses the constraints (for instance a programming language),
2. conjunction as the primary means to combine constraints, since conjunction of formulas corresponds exactly to the intersection of their respective denotations,
3. existential quantification to express locality of variables. It is existential, in contrast to universal, quantification that matters here since most of the constraint-based mechanisms are based on *satisfiability* of descriptions.

2.2 A Quick Tour of Constraint Based Calculi

2.2.1 From Logic Programming to Constrained Logic Programming

Historically, the need of constraints arose from *Logic Programming* (LP), one of the classical formalisms following the paradigm of *declarative programming*. Let us recall the basics of logic programming before we show how constraints make LP an even more natural and efficient declarative language. A *program* is a sequence of *definite* Horn clauses, that is, Horn clauses with a non-empty head. The declarative semantics of a logic program P is the least set M_P of ground atoms such that if $B \leftarrow A_1, \ldots, A_n$ is a ground instance of a program clause of P and if $A_1, \ldots, A_n \in M_P$ then $B \in M_P$. Such a least set does always exist for Horn clauses (but not for arbitrary clauses).

Here is an example of a logic program:

```
arc(a,b).
arc(b,c).
arc(c,d).
arc(c,e).
path(X,X).
path(X,Y) ← arc(X,Z),path(Z,Y).
```

The idea of this program is that ground terms a, b, c, etc. are nodes of a graph, *arc* defines the edges of the graph, and $path(X, Y)$ whether there is a directed path from X to Y.

While functional or imperative programs execute when provided with input data, logic programs execute when provided with *queries*, that is, a conjunction of atoms.

- Asking whether there is an arc from c to e is done by the query $arc(c, e)$. The answer to this query should be affirmative, and this can simply be found by searching the program facts.
- In contrast, the answer to the query $arc(c, X)$ should be all possible nodes X to which there is an arc originating in c. Here, we need to non-deterministically unify the query with the program facts.
- Finally, the answer to the query $path(X, e)$ should be all nodes X from which there is a path ending in e. Here, we need to reflect the declarative semantics of our program in the search mechanism by transforming the query until we fall into the previous case.

This computation mechanism is expressed by the following rule called *resolution*:

$$\text{(Resolution)} \quad \frac{P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots P_n(\bar{t}_n)}{Q_1(\bar{r}_1\sigma), \ldots, Q_m(\bar{r}_m\sigma), P_2(\bar{t}_2\sigma), \ldots, P_n(\bar{t}_n\sigma)}$$

where $P_1(\bar{s}) \leftarrow Q_1(\bar{r}_1), \ldots, Q_m(\bar{r}_m)$ is a program clause not sharing any variable with the query $P_1(\bar{t}_1), \ldots, P_n(\bar{t}_n)$, and σ is the most general unifier of $\bar{s}$ and $\bar{t}_1$.

This rule contains two fundamentally different non-determinisms: the choice of the atom to which resolution is applied is don't-care (it does not matter for the completeness of the search procedure which atom we choose to resolve on, provided we do a breadth-first search), while the choice of the program clause is don't-know (we have to try all possible clauses). This non-determinism is dealt with externally: choosing an atom in the query, we have in principle to execute all possible sequences of resolutions steps. Along every execution sequence we construct the sequence of substitutions computed so far. If such a computation sequence terminates in an empty goal, then we say that the computation *succeeds* and that the computed substitution is an *answer substitution*, otherwise it *fails*.

In pure LP, all data items are represented as ground terms, and unification is the fundamental method of combining descriptions of data items. If data types like natural numbers are to be used in LP they have to be encoded as terms, and operations on numbers have to be expressed as LP predicates. Let us, for example, extend our example of paths in graphs to weighted graphs:

```
plus(0,X,X).
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```

```
arc(a,b,s(0)).
arc(b,c,s(s(0))).
arc(c,d,s(0)).
arc(c,e,s(0)).
path(X,X,0).
path(X,Y,L) ← arc(X,Z,M),path(Z,Y,N),plus(M,N,L).
```

With this program, the goal $path(a, c, L)$ will succeed with answer substitution $\{L \mapsto s(s(s(0)))\}$. However, computation of the goal $plus(X, s(0), X)$ will not terminate. The problem is that, although searching solutions by enumerating all possible computation paths is complete in the limit, this gives us in general not a complete method for detecting the absence of solutions (not to speak of its terrible inefficiency). What we would like here is to have numbers as a primitive data type in LP.

Another problem with pure LP is that, even with terms as data structures, we might want to have more expressiveness for describing terms, and be able, for instance, to formulate the query "is there a path of weight 10 starting in a and ending in a node different from e"? Our approach to use terms with variables as descriptions of ground terms is obviously not sufficient here.

Constraint Logic Programming (CLP) offers a solution to these deficiencies of LP. The basic idea is to disentangle the resolution rule, by separating the process of replacing an atom of the query by the body of a program clause from the other process of unifying the data descriptions. For this, we need to consider a new syntactic entity, namely *equations* between terms. These equalities must be distinguished from possibly already existing equality atoms, since their use will be quite different. In addition, this will allow us to restrict our syntax for atoms to predicate symbols applied to *different* variables, the value of these variables being recorded via the equalities. A Horn clause is now of the form

$$B \leftarrow A_1, \ldots, A_n, c$$

where B and the A_i are atoms and c is a conjunction of equations usually written again as a set. Our example of weighted graphs could now look like this:

```
plus(X,Y,Z) ← X=0,  Y=Z.
plus(X,Y,Z) ← plus(X',Y,Z'), X=s(X'), Z=s(Z').
arc(X,Y,Z)  ← X=a, Y=b, Z=s(0).
arc(X,Y,Z)  ← X=b, Y=c, Z=s(s(0)).
arc(X,Y,Z)  ← X=c, Y=d, Z=s(0).
arc(X,Y,Z)  ← X=c, Y=e, Z=s(0).
path(X,Y,L) ← X=Y, L=0.
path(X,Y,L) ← arc(X,Z,M), path(Z,Y,N), plus(M,N,L).
```

In this new context, queries are called *configurations*. A configuration consists of a list of atoms *followed by* a list of equations. We can rewrite our resolution rule as:

$$\text{(Resolution-2)} \quad \frac{P_1(\bar{X}_1), P_2(\bar{X}_2), \ldots, P_n(\bar{X}_n), c}{Q_1(\bar{Y}_1), \ldots, Q_m(\bar{Y}_m), P_2(\bar{X}_2), \ldots, P_n(\bar{X}_n), c, d}$$

where $P_1(\bar{X}_1) \leftarrow Q_1(\bar{Y}_1), \ldots, Q_m(\bar{Y}_m), d$ is a program clause not sharing any variable with the configuration $P_1(\bar{X}_1), \ldots, P_n(\bar{X}_n), c$ except $\bar{X}_1$, and $c \wedge d$ is satisfiable.

Unification has disappeared from the resolution rule: it is replaced by constructing the conjunction of the two equation systems and checking their satisfiability. Note that in the constrained resolution rule the parameters $\bar{X}_i$ of the predicate P_i are required to be the same lists of variables in the query and in the program clause, this will have to be achieved by renaming the variables in the program clause when necessary.

Semantically, an equation $X = s$ must be understood as the set of values (for the variable X) which are the ground instances of the term s. When making a conjunction $X = s \wedge X = t$, we intersect these two sets of values, and therefore, the set of solutions is the same as that of $X = \textit{most-general-instance}(s, t)$, provided s and t are unifiable. This is why we need to check for satisfiability in our new rule. This satisfiability test will of course be delegated to the old unification algorithm, which is seen here as an "equation solver" put aside. Furthermore, we could allow more freedom in the rule by writing in the conclusion of the rule any equations system that is equivalent to $c \wedge d$. This gives the freedom of whether to leave the constraint $c \wedge d$ as it is, or to replace it by some simpler "solved form".

Execution of our program is now essentially the same as before, we just have separated matters. An important observation is that we have only used the following properties of descriptions of terms:

- we can intersect their denotation (by building the conjunction of the associated equalities)
- we can decide the emptiness of the intersection of their denotations (by unification).

The crucial abstraction step is now: We obtain CLP when we replace equation systems of terms by any system of description of data with the same two properties as above. For example, we could replace the Presburger language for integers used in our last example by the usual integer arithmetic:

```
arc(X,Y,Z)  ← X=a, Y=b, Z=1.
arc(X,Y,Z)  ← X=b, Y=c, Z=2.
arc(X,Y,Z)  ← X=c, Y=d, Z=2.
arc(X,Y,Z)  ← X=c, Y=e, Z=2.
path(X,Y,L) ← X=Y, L=0.
path(X,Y,L) ← arc(X,Z,M), path(Z,Y,N), L = M + N.
```

Some authors have used the notation $CLP(X)$ fro this programming paradigm, emphesizing the fact that it is parameterized by a constraint system X. In this terminology, a concrete constraint logic programming system is obtained when instantiating X by some constraint system.

The theory of $CLP(X)$ has been developed in [JL87]. More about Constraint Logic Programming can be found for instance in [MS98].

2.2.2 Constraint Programming

The CLP calculus presented in Section 2.2.1 is an non-deterministic calculus. To define the semantics of of a CLP program it is necessary to talk about *all* possible deductions in the calculus. This non-determinism is not reflected in the calculus itself. For a concrete implementation of CLP this means that the machine must be able to explore systematically all computation paths, this is usually implemented with some kind of backtracking.

In this section we want to present another constraint-based programming formalism that

- represents indeterminism *explicitly* in the calculus, by using disjunctions, and
- allows both deterministic and non-deterministic computation steps.

Constraint Programming (CP) in its pure form follows a paradigm that is fundamentally different from the philosophy of CLP as seen in Section 2.2.1. CP is based on *deterministic* computation while the latter follows the tradition of *declarative programming*, that is based on *non-deterministic* computation that can be realized using an additional enumeration procedure. In the following we present a simple calculus that reunites CLP and pure CP. Constraint programming has been introduced by V. Saraswat [SR90, SR91]. The presentation in this section describes a fragment of the ultraconcurrent calculus of the Oz 1 programming language [Smo95], in contrast to the Oz 2 language and the MOZART programming system which are based on a sequential programming calculus with an explicit construct for concurrency [vRH04].

A *program* is a sequence of procedure definitions

$$\begin{array}{rcl} P_1(x_1,\ldots,x_{n_1}) & \Leftrightarrow & A_1 \\ & \vdots & \\ P_m(x_1,\ldots,x_{n_m}) & \Leftrightarrow & A_m \end{array}$$

where the A_i are *agent expressions* to be defined below. The variables x_i are considered *universally quantified*. We will rename these bound variables whenever necessary. The set of agent expressions is defined by the following grammar:

$$\begin{array}{rcll} A & \rightarrow & P(x_1,\ldots,x_n) & \textit{procedure call} \\ & \rightarrow & c & \textit{constraint} \\ & \rightarrow & A \wedge A & \textit{conjunction} \\ & \rightarrow & \exists \bar{x}\, A & \textit{scoped expression} \\ & \rightarrow & \texttt{ if } D \texttt{ fi} & \textit{conditional} \\ & \rightarrow & \texttt{ or } D \texttt{ ro} & \textit{disjunction} \\ D & \rightarrow & & \textit{empty clause list} \\ & \rightarrow & C \,[]\, D & \textit{non-empty clause list} \\ C & \rightarrow & \exists \bar{x} c \texttt{ then } A & \textit{clause} \end{array}$$

In a clause list we usually omit the last []. In a clause $\exists \bar{x} c$ `then` A, the scope of the existential quantifier extends over the constraint c (the *guard* of the clause) and the agent A (the *protected part* of the clause).

A CP-configuration is either FAIL or a pair $c \mid As$ where c is a constraint and As is a set of agent expressions. The reduction rules of the calculus are given in Figure 2.1. In the Local-rule it is understood that $\bar{y}$ and $\bar{x}$ are both vectors of pairwise different variables and are of the same length. The notation $A\{\bar{x} \mapsto \bar{y}\}$ denotes the agent expression obtained by replacing in A every free occurrence of a free variable from $\bar{x}$ by its counterpart from $\bar{y}$.

A computation is called *successful* when it terminates in a configuration with an empty set of agents, it *fails* when it terminates in FAIL, and it is *stuck* when it terminates in a configuration where no further rule application is possible and where the agent space is non-empty.

The disjunctive and conditional agents are at the heart of our CP calculus. Roughly, disjunctive agents express a *don't-know* choice while conditional agents express a *don't-care* choice. More precisely:

There are three ways how a disjunctive agent can evolve:

$$\text{(Unfold)} \qquad \frac{c \mid P(x_1,\ldots,x_n),\psi}{c \mid A,\psi} \qquad P(x_1,\ldots,x_n) \Leftrightarrow A \text{ is a program clause}$$

$$\text{(Failure)} \qquad \frac{c \mid c',\psi}{\textsc{Fail}} \qquad c \wedge c' \text{ unsatisfiable}$$

$$\text{(Simplify)} \qquad \frac{c \mid c',\psi}{c'' \mid \psi} \qquad c \wedge c' \text{ satisfiable and } c \wedge c' \equiv c''$$

$$\text{(Unpack)} \qquad \frac{c \mid A_1 \wedge A_2,\psi}{c \mid A_1, A_2,\psi}$$

$$\text{(Local)} \qquad \frac{c \mid (\exists \bar{x}\, A),\psi}{c \mid A\{\bar{x} \mapsto \bar{y}\},\psi} \qquad \bar{y} \text{ fresh variables}$$

$$\text{(DisDrop)} \qquad \frac{c \mid \ \texttt{or}\ D \,[]\, \exists \bar{x}\, c' \ \texttt{then}\ A \,[]\, D' \ \texttt{ro}\ ,\psi}{c \mid \ \texttt{or}\ D \,[]\, D' \ \texttt{ro}\ ,\psi} \qquad c \wedge \exists \bar{x} c' \text{ unsatisfiable}$$

$$\text{(DisSingle)} \qquad \frac{c \mid \ \texttt{or}\ \exists \bar{x}\, c' \ \texttt{then}\ A \ \texttt{ro}\ ,\psi}{c \mid (\exists \bar{x}\,(c' \wedge A)),\psi}$$

$$\text{(DisEmpty)} \qquad \frac{c \mid \ \texttt{or}\ \ \texttt{ro}}{\textsc{Fail}}$$

$$\text{(CondDrop)} \qquad \frac{c \mid \ \texttt{if}\ D \,[]\, \exists \bar{x}\, c' \ \texttt{then}\ A \,[]\, D' \ \texttt{fi}\ ,\psi}{c \mid \ \texttt{if}\ D \,[]\, D' \ \texttt{fi}\ ,\psi} \qquad c \wedge \exists \bar{x} c' \text{ unsatisfiable}$$

$$\text{(CondFire)} \qquad \frac{c \mid \ \texttt{if}\ D \,[]\, \exists \bar{x}\, c' \ \texttt{then}\ A \,[]\, D' \ \texttt{fi}\ ,\psi}{c \mid (\exists \bar{x}(c' \wedge A)),\psi} \qquad c \models \exists \bar{c}',$$

$$\text{(CondEmpty)} \qquad \frac{c \mid \ \texttt{if}\ \ \texttt{fi}\ ,\psi}{c \mid \psi}$$

Figure 2.1: Computation rules for CP.

1. A clause is dropped by the rule DisDrop if its guard is not consistent with the constraint store.

2. If the set of clauses is empty then the agent fails. This is consistent with the logical view of an empty disjunction being equivalent to $false$.

3. If there is only one clause in the agent then the disjunction disappears and sets free the guard together with its protected part. Application of the Local-rule will eliminate the existential quantifier, allowing for further consideration of the agent expressions.

Note that, in presence of several disjunctive clauses, we can not commit to a clause when its guard is entailed by the constraint of the current configuration (since that would amount to a *don't-care* choice whereas we want disjunctive agents to model *don't-know* choices).

This treatment of disjunction is very similar to the *Andorra* rule proposed by D. H. Warren for logic programming to allow determinate reduction steps (the names comes from And-Or).

The conditional agent behaves differently:

1. A conditional *fires* its clause if its guard is entailed by the constraint store. Entailment is necessary for firing the clause even when there is only one clause left in the conditional expression.

2. A conditional clause is dropped when its guard is inconsistent with the constraint of the current configuration.

3. The conditional vanishes when no clause is left; as an alternative to our definition, one could allow a conditional expression to have an *otherwise* expression which fires when all regular clauses are eliminated.

A dramatic consequence of the rules for the conditional expression is that, a priori, logical semantics is lost. For a given constraint store it might well be the case that the guards of several clauses are entailed at a time. In this case, our calculus makes a non-deterministic *don't-care* choice of the clause to be fired. It is hence the responsibility of the programmer to make the guards of the conditional clauses strong enough such that a *don't care* commitment to a clause causes no harm. This choice is fundamentally different from the choice seen in Section 2.2.1 when selecting a clause since now all non-determinism is explicit in the expressions. There are, however, ways to ensure by a careful analysis that a conditional agent still behaves in a logically meaningful way (see, for instance, [AFM96]).

The constraint store increases monotonically during the course of a computation, that is if $c \mid \psi \rightarrow^* c' \mid \psi'$ then $c \models c'$ holds. The consequence is that if in a certain configuration a conditional agent can fire one of its clauses, then he will still be able to fire this clause after further progress of the other agents (provided that the clause has not disappeared due to inconsistency). This is one key to consider the calculus as a calculus of *concurrent computations*.

The agents co-exist independently from each other in the agent space, the only way to have some "communication" among agents is by having them impose constraints on the constraint store. Formally, if $c \mid A, \psi \rightarrow c \mid \phi, \psi$, then for all sets ψ' of agent expressions we have as well that $c \mid A, \psi' \rightarrow c \mid \phi, \psi'$. This is another key to concurrency.

Let us now demonstrate the difference between disjunctions and conditionals with an example: the concatenation of lists. First we define a concatenation predicate as a disjunction (the binary constructor symbol ":" is here written in infix notation):

```
conc(X,Y,Z) ↔
   or
       X=nil ∧ Y=Z
    [] ∃ H,X',Z' X=H:X' ∧ Z=H:Z' then conc(X',Y,Z')
   ro
```

Here is a query leading to a successful computation:

$$
\begin{aligned}
& X = a\!: nil \wedge Y = b\!: c\!: nil \mid conc(X,Y,Z) \\
& X = a\!: nil \wedge Y = b\!: c\!: nil \mid \texttt{ or } X = nil \wedge Y = Z \,[]\, \exists H, X', Z' \ldots \texttt{ ro} \\
& X = a\!: nil \wedge Y = b\!: c\!: nil \mid \\
& \qquad \texttt{or } \exists\, H, X', Z'\ X = H\!: X' \wedge Z = H\!: Z' \texttt{ then } conc(X', Y, Z') \texttt{ ro} \\
& X = a\!: nil \wedge Y = b\!: c\!: nil \mid \\
& \qquad \exists H, X', Z' \,(X = H\!: X' \wedge Z = H\!: Z' \wedge conc(X', Y, Z')) \\
& X = a\!: nil \wedge Y = b\!: c\!: nil \mid X = H\!: X' \wedge Z = H\!: Z', conc(X', Y, Z') \\
& X = a\!: nil \wedge Y = b\!: c\!: nil \wedge H = a \wedge X' = nil \wedge Z = a\!: Z' \mid conc(X', Y, Z') \\
& \ldots \\
& X = a\!: nil \wedge Y = b\!: c\!: nil \wedge H = a \wedge X' = nil \wedge Z = a\!: b\!: c\!: nil \wedge \\
& \qquad Z' = b\!: c\!: nil \mid \emptyset
\end{aligned}
$$

The following computation, however, is not successful since it gets stuck with a disjunction that can not reduce any further:

$$
\begin{aligned}
& Y = nil \mid conc(X,Y,Z) \\
& Y = nil \mid \texttt{ or } X = nil \wedge Y = Z \,[]\, \exists H, X', Z'\ X = H\!: X' \wedge \ldots \texttt{ ro}
\end{aligned}
$$

This situation can always occur when we use disjunctive agents. One solution is to consider a *disjunction of configurations* and to have an additional enumeration rule in the calculus:

$$
\text{(Enumerate)} \quad \frac{c \mid \texttt{ or } C \,[]\, D \texttt{ ro } , \psi}{\Big(c \mid C, \psi\Big) \vee \Big(c \mid \texttt{ or } D \texttt{ ro } , \psi\Big)}
$$

This rule should only be used when no other reduction is possible. In some cases, in particular for constraint satisfaction problems as seen in Section 2.2.4, such an *enumeration rule* is necessary. In other cases, however, it is possible to overcome the weakness of disjunctions by using conditional expressions, as in our example. Here is a version of the concatenation predicate using a conditional agent expression:

```
conc(X,Y,Z) ↔
  if X=nil then Y=Z
   [] Y=nil then X=Z
   [] Z=nil then X=nil ∧ Y=nil
   [] ∃ H,X',Z' X=H:X'  then  Z=H:Z' ∧ conc(X',Y,Z')
  fi
```

With this conditional version of conc, both example queries seen above lead to successful computations:

$$
\begin{aligned}
& X = a\colon nil \wedge Y = b\colon c\colon nil \mid conc(X, Y, Z) \\
& X = a\colon nil \wedge Y = b\colon c\colon nil \mid \\
& \qquad \texttt{if} \ \ldots [] \, \exists H, X', Z' \ X = H\colon X' \ \texttt{then} \ Z = H\colon Z' \wedge conc(X', Y, Z') \ \texttt{fi} \\
& X = a\colon nil \wedge Y = b\colon c\colon nil \mid \\
& \qquad \exists H, X', Z'(X = H\colon X' \wedge Z = H\colon Z' \wedge conc(X', Y, Z')) \\
& X = a\colon nil \wedge Y = b\colon c\colon nil \mid X = H\colon X' \wedge Z = H\colon Z', conc(X', Y, Z') \\
& X = a\colon nil \wedge Y = b\colon c\colon nil \wedge Z = a\colon Z' \wedge H = a \wedge X' = nil \mid conc(X', Y, Z') \\
& X = a\colon nil \wedge Y = b\colon c\colon nil \wedge Z = a\colon Z' \wedge H = a \wedge X' = nil \mid \\
& \qquad \texttt{if} \ X' = nil \ \texttt{then} \ Y = Z' \ \texttt{fi} \\
& X = a\colon nil \wedge Y = b\colon c\colon nil \wedge Z = a\colon b\colon c\colon nil \wedge H = a \wedge \\
& \qquad X' = nil \wedge Z' = b\colon c\colon nil \mid \emptyset
\end{aligned}
$$

$$
\begin{aligned}
& Y = nil \mid conc(X, Y, Z) \\
& Y = nil \mid \ \texttt{if} \ \ldots [] \, Y = nil \ \texttt{then} \ X = Z \, [] \ldots \ \texttt{fi} \\
& Y = nil \mid Y = nil, X = Z \\
& Y = nil \wedge X = Z \mid \emptyset
\end{aligned}
$$

2.2.3 Constraints and Theorem Proving

Let E be a finite set of equations, for example the classical three equations defining group theory:

$$
\left\{
\begin{array}{rcl}
(x \times y) \times z & = & x \times (y \times z) \\
x \times 1 & = & x \\
x \times x^{-1} & = & 1
\end{array}
\right.
$$

A classical problem is to decide whether a given equation, for example $(x \times y)^{-1} = y^{-1} \times x^{-1}$ in group theory, is a logical consequence of E. This problem, also known as the *word problem*, has been subject to intensive research. The brute force search for a proof using the replacement of equals by equals, although complete, rarely leads to an actual solution. One of the most successful approaches due to Knuth and Bendix is to use *ordered strategies*, that is to use the equations in one way only as so-called *rewrite rules*. Of course, such a strategy is incomplete in general, but completeness can be regained using a completion mechanism based on the computation of some particular equational consequences called *critical pairs*. One requirement of the original method was the *termination* of the rewrite system: the replacement of equals by equals using the ordered strategies should always end up after a finite number of replacement steps.

In the above example of group theory, it is quite easy to fulfill this termination requirement by choosing carefully the way in which to orient the equations. The situation changes if we consider *commutative groups*, adding the equation $x \times y = y \times x$ to the above system. Now the completion procedure fails because commutativity cannot be oriented in either way without loosing termination. Several solutions have been studied to overcome this problem, the solution that interests us here is called *ordered rewriting*.

The idea is very simple: use every equation in one way or the other, depending on the ordering on *the instances on which it is applied*. For example consider the commutativity axiom and assume a total ordering on terms, e.g compare lexicographically the arguments of $\times$, from left to right. Then if $a > b$, $a \times b$ rewrites to $b \times a$ using $x \times y = y \times x$, but not the other way around, since $a \times b > b \times a$, but $b \times a \not> a \times b$.

The equations that we are trying to prove are (implicitly) universally quantified, that is an equation as $x \times y = y \times x$ should be read as "for all ground terms s and t, $s \times t = t \times s$". Hence, as in the case of (C)LP, terms with variables represent the sets of their respective ground instances. In order to describe ordered rewriting we need a total ordering relation on ground terms, denoted $>$. Now, an equation $l = r$ can be replaced by the two constrained rules $l \to r \mid l > r$ and $r \to l \mid r > l$. Since we required the order to be total, we know that for any ground instance of l and r, one of the two rules applies (except in the case $l = r$ when there is of course no need to rewrite). When we require in addition the ordering to be monotonic, that is to satisfy

$$s > t \quad \text{implies} \quad f(\ldots, s, \ldots) > f(\ldots, t, \ldots)$$

for all function symbols f and ground terms s, t, then the ordering on the rule instances propagates to the rewritten terms. Hence, rewriting always decreases terms in the ordering. Assuming then the ordering to be well-founded implies that rewriting a term always ends up in a *normal form*. This latter property is the key to completeness.

Once the equations are split into ordered rewriting rules as described above, we have to complete the rule system in order to obtain a rewrite system that allows to derive all valid equations $s = t$ by rewriting from s and t until a same term u is found. Without going into the technical details here it suffices to say that the completion procedure searches for so-called *critical pairs*, obtained when two rewrite rules $l \to r \mid l > r$ and $g \to d \mid g > d$ can overlap in the same ground term v. For this to happen, say at the top of v to make it simple, a ground substitution σ must exist such that $v = l\sigma = g\sigma$, $l\sigma > r\sigma$ and $g\sigma > d\sigma$. Starting from v, rewriting can yield the two different terms $r\sigma$ and $d\sigma$, and we have to repair this divergence by adding additional rewrite rules. But since the substitution σ is a solution of the *ordering constraint* $l = g \wedge l > r \wedge g > d$, the divergence can be repaired by adding the equation (called critical pair) $r = d \mid l = g \wedge l > r \wedge g > d$, more precisely, the two rules $r \to d \mid l = g \wedge l > r \wedge g > d \wedge r > d$ and $d \to r \mid l = g \wedge l > r \wedge g > d \wedge d > r$. Since we know that the ordering is total, one of the two rules will rewrite one -we don't know which one, of course- of $(r\sigma, d\sigma)$ into the other. This process of adding critical pairs may terminate with a so-called *canonical* rewrite system, that is one which can prove all valid equations by rewriting, or run forever, in which case any valid equation will be provable by rewriting after some time.

Here, we have seen an application of constraints in theorem proving in which constraints play already the role of *defining* the notion of proof, via the definition of ordered rewriting. Constraints can also be used in a completely different way: to restrict the set of possible inferences. In the above discussion, we have overlapped two rules at the top. In practice, we must actually overlap left-hand sides of rules at arbitrary non-variable subterms. Assume now that we can solve the constraints, and replace the ordered rules $l \to r \mid l > r$ by plain rewrite rules $l\sigma \to r\sigma$, where σ is a kind of mgu for the constraint $l > r$. Then overlapping these new rules does not have the same effect at all than overlapping the constrained one they originate from. In the first case, the overlap may happen *inside* the substitution σ, which is impossible for the constrained rule: the use of constraints allows to avoid overlapping inside substitutions originating from previous inferences. This strategy is called *basic*. It is again complete, although it influences in various ways the whole inference process.

So far we have seen that the use of constraints on automatic theorem proving raises the need to decide the satisfiability of basic constraints, that is the need for an algorithm to decide the validity of $\exists^*$-formula. If we want to decide the *subsumption relation* between two constrained rules then we have to be able to decide the validity of $\forall^*\exists^*$-formulas. Consider two constrained rewrite rules

$$\begin{array}{ll} (R1) & s \to t \mid c(\bar{x}) \\ (R2) & u \to v \mid d(\bar{y}) \end{array}$$

Then, $(R2)$ *subsumes* $(R1)$ if every rewrite of a ground term by $(R1)$ can be simulated by a rewrite step by $(R2)$, that is if the following formula is true:

$$\forall \bar{x}\Big(c(\bar{x}) \to \exists \bar{y}(s = u \wedge t = v \wedge d(\bar{y}))\Big)$$

We can go even further and try to make the rewrite rules disjoint in the sense that no rewrite of a ground term can be done by two different rewrite rules. In the case of the two rewrite rules above this can be achieved by replacing the constraint $d(\bar{y})$ of the rule $(R2)$ by the stronger constraint

$$d(\bar{y}) \wedge \neg\exists \bar{x}\Big(c(\bar{x}) \wedge s = u \wedge t = v\Big)$$

Note that iterated applications of this step will produce constraints with arbitrarily nested existential and universal quantifiers. As a consequence, we have to cope with a set of constraints which is the complete set of first-order languages.

More about the use of constraints in theorem proving can be found in [KKR90, GN01].

2.2.4 Constraint Satisfaction Problems

So far we have seen that constraints are useful to define an expressive programming paradigm and as a tool to express strategies in automated deduction. Many important problems already employ constraints in the problem formulation, and the task is to find a solution to these constraints. We demonstrate this with a well known game:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

Each letter denotes a different number between 0 and 9 such that, when read as an addition, the equation is satisfied.

This problem can easily be formalized in a logical setting, by choosing variables S, E, N, D, M, O, R, Y with possible values $0, \ldots, 9$, stating that each of the variables takes a different value, and stating the arithmetical relation that corresponds to the above equation. A naïve approach to solving such a constraint consists in enumerating all possible values of the 8 variables and checking the difference and arithmetical constraints for each of these valuations. There are 10^8 assignments of different digits to the letters which makes this brute-force approach impractical.

Here is how this problem is modelized in GNU-Prolog [Dia07]:

```
sendmore(S,E,N,D,M,O,R,Y) :-
    fd_domain([E,N,D,O,R,Y],0,9),
    fd_domain([S,M],1,9),
                1000*S + 100*E + 10*N + D
     +          1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
    fd_all_different(L),
    fd_labeling(L).
```

This formulation is concise and elegant. It uses a so-called *global* constraint based on the varyadic predicate *all_different* which is true when all its arguments take different integer values. Modern constraint programming systems can solve this problem in a fraction of a second using a clever approach based on *constraint propagation*. Constraint propagation implements a kind of meta-reasoning on the constraints themselves, in order to cut down the search. For an example, M must be equal to 1, since it is a carry on. In turn, this implies that S must be equal to 8 or 9. Constraint propagation on intervals (also called *finite domains*) was first implemented in the CHIP system [DSH90].

Although the above discussion is certainly just a toy example, this kind of application is of paramount importance in practice. Problems of this kind occur, for instance, in all kind of scheduling problems (think of making a staff plan under restricted availability of the persons, scheduling task among different machines where the execution of tasks has to obey to a certain order, making a timetable for a school, etc.).

More about constraint satisfaction problems can for instance be found in [Hen89, MS98].

2.2.5 Constraint Based Grammar Formalisms

So called *unification grammars* have a long history in computational linguistics where they are employed for the analysis and generation of natural language. Originally, these grammars were defined without using constraints or even predicate logic. The name "unification grammar" is a bit unfortunate since "unification" is the operation of resolving equations, calling them *constraint grammars* is much more appropriate since this name emphasizes the declarative aspect of the formalism. However, these constraint grammars are usually not considered as a generic formalism that can be used with any constraint system (as it is the case with constraint logic programming) but come with so-called feature constraints that we will see in an example below, and that we will investigate in greater depth in Section 5.2.

We will demonstrate the principle with a very simple example inspired by [Smo92]. Consider a context-free grammar for English which contains the following rules:

$$\begin{array}{rcl} S & \rightarrow & NP\ VP \\ NP & \rightarrow & D\ N \\ VP & \rightarrow & V\ NP \end{array}$$

where the non-terminals stand for *Subject*, *Noun Phrase*, *Verb Phrase*, *Determiner*, and *Noun* respectively. Additionally, there are so-called lexical rules that represent a dictionary:

$$\begin{array}{rcl} N & \rightarrow & \texttt{girl} \\ N & \rightarrow & \texttt{movie} \\ N & \rightarrow & \texttt{movies} \\ D & \rightarrow & \texttt{a} \\ D & \rightarrow & \texttt{the} \\ V & \rightarrow & \texttt{watches} \end{array}$$

This grammars licenses correct English sentences like

`the girl watches a movie`

but also incorrect sentences like

`the girl watches a movies`

The problem is that context-free grammar rules can only define the possible skeletons of a phrase, however, there are additional conditions to be met for a sentence to be grammatically correct, as for instance correspondence of the numeros of the subject and the verb.

Hence, we augment our context-free grammar with constraints. While a context-free grammar describes a set of parse trees, a constraint grammar describes a set of attributed parse trees. In such an attributed parse tree, nodes can be decorated with functional attributes called *features*, and the constraints in the grammar impose relations between the attributes of the nodes for an attributed parse tree to be admissible, that is to describe a correct English sentence. For example, we might express the correspondence of the numeros of subject and verb by adding some constraints to the structural and to the lexical grammar:

$$
\begin{array}{rcl}
S & \to & NP\ VP \mid NP.numeros = VP.numeros \\
NP & \to & D\ N \mid NP.numeros = D.numeros \wedge D.numeros = N.numeros \\
VP & \to & V\ NP \mid VP.numeros = V.numeros \\
N & \to & \texttt{girl} \mid N.numeros = singular \\
N & \to & \texttt{movie} \mid N.numeros = singular \\
N & \to & \texttt{movies} \mid N.numeros = plural \\
D & \to & \texttt{a} \mid D.numeros = singular \\
D & \to & \texttt{the} \\
V & \to & \texttt{watches} \mid V.numeros = singular
\end{array}
$$

Now, the constraints only allow sentences where the numeros of the noun phrase coincides with the numeros of the verb phrase and where in a noun phrase the numeros of the determiner coincides with the numeros of the noun. An interesting point to note about this grammar is that constraints are used to accomplish two different tasks:

- to express restrictions on the attributes of components, for instance that in a noun phrase the numeros of determiner and noun must coincide; and
- to define the attributes of nodes, for instance the numeros of a noun phrase is defined to be the numeros of its determiner.

Also, note that we did not specify the numeros of the determiner `the` which might be singular or plural. Hence, our grammar licenses both noun phrases `the movie` and `the movies`.

More about the use of feature descriptions in computational linguistics can be found at [Kel93].

2.2.6 Constraint-Based Program Analysis

Constraint based program analysis stands a bit apart from the constraint-based formalisms mentioned before since here constraints are not used as a description language to augment the descriptive power of another formalism (programming language, deduction calculus, grammar) but are rather used as a means to *approximate* the meaning of a program.

Constraint-based program analysis uses *set constraints* as a language to approximate the meaning of a program. Different systems of set constraints have been investigated during the last decade, here we use the constraint system of [HJ90].

The logical language of our set constraint system contains a signature Σ, a binary union operator, and a constant $\top$. In addition, the language contains for every n-ary function symbol $f \in \Sigma$ so called *projections* $f_1^{-1}, \ldots, f_n^{-1}$ which are all unary function symbols. The only predicate symbol is the binary symbol $=$. Constraints are possibly existentially quantified conjunctions of atomic constraints.

Basic data items in the interpretation of set constraints are *sets of ground terms*. Function symbols operate now on sets, that is the meaning of $f(M, N)$ is $\{f(m, n) \mid m \in M, n \in N\}$. The symbol $\top$ denotes the set of all ground terms, $\cup$ and $=$ have their usual meaning, and the meaning of $f_i^{-1}(M)$ is

$$\{m_i \mid \exists m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n\, f(m_1, \ldots, m_i, \ldots, m_n) \in M\}$$

Satisfiability of these constraints is decidable but non-trivial, see the literature for pointers.

As a simple example we demonstrate the approximation of the meaning of a logic program; the method works as well for functional or imperative programs. Consider the well-known logical program for the concatenation of lists:

```
conc(nil,V,V).
conc(H:X,Y,H:Z) ← conc(X,Y,Z).
```

No variable is shared among clauses, this is necessary for the abstraction to work (and can easily be achieved, the case given, by renaming of variables). The abstraction uses one set variable Γ, corresponding to the predicate *conc*, and variables V, H, X, Y, Z. First, we state that a successful computation of *conc* can be obtained by starting from the first clause or by starting from the second clause:

$$\Gamma \quad = \quad conc(nil, V, V) \cup conc(H : X, Y, H : Z)$$

To get constraints for the variables V, X, Y, Z, H we have to look at the clauses where they occur (note that every of these variables has to occur in exactly one clause). The variables V and H do not occur in the bodies of their clause, hence there is no restriction on them and they can denote any term:

$$\begin{aligned} V &= \top \\ H &= \top \end{aligned}$$

The variables X, Y and Z can assume any value that stems from *some* successful computation of *conc*. More precisely, X can take any value such that for *some value* of Y and Z the computation of $conc(X, Y, Z)$ is successful. This is were the approximation happens: the point-wise dependencies between variables are ignored. We get the constraints

$$\begin{aligned} X &= conc_1^{-1}(\Gamma) \\ Y &= conc_2^{-1}(\Gamma) \\ Z &= conc_3^{-1}(\Gamma) \end{aligned}$$

Hence, the complete set constraint to resolve is

$$\Gamma = conc(nil, V, V) \cup conc(H : X, Y, H : Z) \wedge V = \top \wedge H = \top \wedge$$
$$X = conc_1^{-1}(\Gamma) \wedge Y = conc_2^{-1}(\Gamma) \wedge Z = conc_3^{-1}(\Gamma)$$

This constraint is satisfiable, its minimal solution assigns to V, H, Y and Z the set of all ground terms, and to X the set of all lists.

2.2.7 Constraint Solving for Typing

It is well-known that the type-inference problem for terms in a polymorphic type systems can be reduced to unification, that is the resolution of equational constraints. This approach has been pioneered by Damas and Milner [DM82]. For instance, given the term

$$f(f((1, x, z), 1.0, y)$$

We obtain the following equational constraint system:

$$\begin{array}{rcl} t_f & = & arrow(f_1, f_2, f_3, f_r) \\ f_1 & = & int \\ f_2 & = & real \\ f_r & = & f_1 \end{array}$$

The first equation expresses the fact that f is a function with 3 arguments which is obtained from a syntactical analysis of the term; the second one that the first argument of f is of type *int* which is obtained from the inner application of f with first argument 1; the third one that the second argument of f is of type *real*, obtained from the outer application of f with second argument 1.0, and the last one that the result type of f is the same as the first argument type of f, obtained from the nested application of f. The most general solution to this constraint system amounts to giving f the polymorphic type $int, real, \alpha \to int$, where α is a type variable. Note that the above example is simplified for the purpose of illustration. In the context of ML typing, in fact, fresh type variables are only introduced for formal arguments in function definitions.

The problem becomes more difficult in case of more expressive type systems [PWO97, OW97]. One might for instance combine polymorphism with subtyping, which leads to the constraint systems of so-called structural and non-structural subtyping introduced in Sections 4.3.2 and 4.3.3. Type inference amounts to constraint solving in a such a system. A more difficult problem is the one of type *matching*, that is the question whether every instance of a type is also instance of some other type. This problem occurs for instance when one wants to check whether a module implementation fits an interface for that module. On the constraint side this matching problems corresponds to an entailment (implication) problem between constraints.

2.2.8 Constraint-Based Model Checking

Reactive systems are traditionally modeled by finite labelled graphs, the vertices and edges of which denoting respectively the set of states and the set of transitions between states under the action of the events labelling the edges. Because of the finiteness assumption, these graphs can be seen as automata whose alphabet is the set of events. Safety analysis is concerned with the problem whether a given set of (forbidden) states is accessible from the initial one. This question is of course decidable by computing the set of all reachable states.

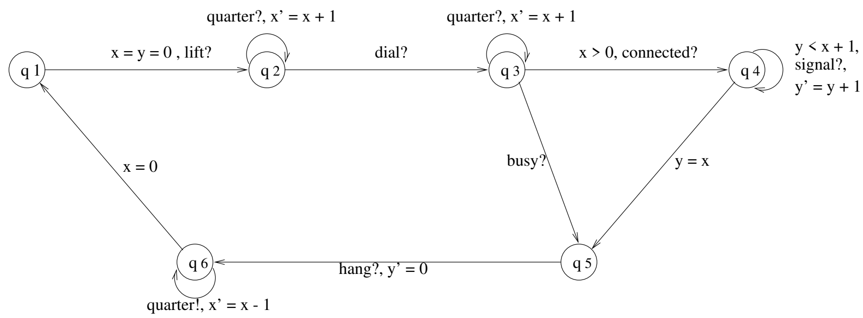


Figure 2.2: A pay phone controller

The actual description of a reactive system is usually given by a set of concurrent programs updating various variables. A state of the system can then be thought of as a mapping from the set of variables to the set of values they may take, while the events are interpreted as state updates. In this view, the above finiteness assumption becomes overly restrictive for modeling practical applications, since it rules out variables taking their value in an infinite set. A more general view based on the idea of Myhill and Nerode's theorem allows for variables taking (possibly infinite) sets of values in a given state, by merging those states having the same behavior with respect to the set of events. Constraints may then serve to describe the set of values taken by the set of variables in a given state via their (possibly infinitely many) solutions. In practice, the constraints are actually used to label the transitions, and are interpreted both as guards and updates. Depending on the constraint system used in this more general setting, the computation of the set of accessible states may still be in PSPACE, or of a higher complexity, or even become undecidable.

The example of a pay phone controller given in figure 2.2 is taken from [CJ99]. There are two variables ranging over natural numbers usually called counters, x for the available credit (number of quarters fed in the machine), and y for the number of time units already used by the customer. The automaton may receive signals such as "quarter?", meaning that a coin has been fed in the machine, "connected?", meaning that the connection has been set up, as well as "busy?", "dial?", "lift?", "hang?" and "signal?" whose meaning is self explanatory. The automaton may itself emit the signal "quarter!" meaning that a quarter will be given back to the user. Transitions occur under the reception or emission of a given signal, and may require the satisfaction of some formula relating the variables x and y, as well as their updated values x' and y'. A formula like $x = y$ should then be understood as a guard, while a formula like $y' = 0$ should be understood as an update of the variable y. The following is a valid transition sequence of the automaton, where the name of a state comes with with the values of x and y successively:

$$
\begin{array}{llllll}
(q_1, 0, 0) & \xrightarrow{lift?} (q_2, 0, 0) & \xrightarrow{quarter?} (q_2, 1, 0) & \xrightarrow{quarter?} (q_2, 2, 0) \\
& \xrightarrow{dial?} (q_3, 2, 0) & \xrightarrow{connected?} (q4, 2, 0) & \xrightarrow{signal?} (q_4, 2, 1)
\end{array}
$$

For instance, the last transition step is acceptable because $(2, 0), (2, 1) \models y < x + 1 \wedge y' = y + 1$. Note that two successive states are needed to verify if a guard is satisfied when it involves primed variables.

Here, safety (from the point of view of the telephone company) will require that, e.g., a connection cannot be established without paying beforehand. In the case above, safety properties are decidable. This is shown by reducing the problem to the emptiness problem for the kind of automaton used in this example, and showing that the latter property is decidable.

The constraints used in this example belong to Presburger arithmetic, and indeed, such constraints occur naturally in reactive systems when modeling the various resources used by the system (in our example, the quarters). Discrete time is a resource that can be naturally modeled by a natural number, giving rise to such constraints. But discrete time implies the existence of a global clock monitoring all processes in the system. When there are several independent clocks in a reactive system, discrete time cannot be used anymore, and must be replaced by real time (or an approximation of it). Then, constraints belong to Tarski's theory of real arithmetic. Guards expressing geometric conditions also use real arithmetic.

Unfortunately, decidability of a constraint system is not enough to ensure the decidability of the emptiness problem for a class of automata whose guards are expressed in this system. For example, the emptiness problem for automata with multiple counters using full Presburger arithmetic for expressing the guards is undecidable (since Minsky machines are an instance of this model). Various restrictions (of the constraint language, or of the automata-based underlying model) have been explored for which decidability is recovered.

2.2.9 Constraints for the Verification of Cryptographic Protocols

In its most basic form, a cryptographic protocol consists of several programs called *roles*, where each program is a sequence of receive-send pairs of the form

$$\texttt{receive}(r_i) \rightarrow \texttt{send}(s_i)$$

In the symbolic modelisation of cryptographic protocols proposed by Dolev and Yao [DY83], the expressions r_i and s_i in such a program are symbolic terms possibly containing variables, with the restriction that every variable appearing in a send-term s_i occurs in one of the receive-patterns r_j preceding it, that is with $j \leq i$. The semantics of such a program is that, if the program is at the receive-send pair i and if it receives from the communication network a (ground) term t then it tries to match the receive pattern r_i with t, if this is possible it applies the thus obtained matching substitution to the rest of the program and emits on the communication network the send-term t_i which is now, due to the restriction of variable occurrences, instantiated to a ground term. The challenge is to verify security properties of such a protocol in presence of an *active* attacker who has complete control over the communication network. The attacker can to a certain extend analyze terms he has observed on the network and exchange them against new terms he has synthesized. The exact analysis and synthesis capabilities of an intruder are described by an inference system named after Dolev and Yao. The most basic security property of a protocol is the so-called *secrecy property*: can the attacker modify the messages exchanged on the network in such a way that he can at the end of the execution of the protocol deduce some knowledge that is supposed to be secret?

This secrecy problem can be modeled as a constraint problem [MS01] where the constraint system contains varyadic constraints of the form $t_1, \ldots, t_n \vdash t$, with the meaning that the intruder can deduce the term t from the terms $t_1, \ldots, t_n$. If we fix the number of parallel sessions of the protocols (this is a necessary prerequisite for decidability of the problem), and if we choose one of the finitely many possible interlacings of the execution of the protocol this problem can be stated as a constraint system

$$\begin{array}{rcl} s_0 & \vdash & r_1 \\ s_0, s_1 & \vdash & r_2 \\ \vdots & & \vdots \\ s_0, s_1, \ldots, s_{n-1} & \vdash & r_n \end{array}$$

$$s_0, s_1, \ldots, s_{n-1}, s_n \quad \vdash \quad secret$$

The term s_0 can be thought of as a token that triggers the start of the protocol. A line in this constraint system, apart from the last one, states that the intruder can emit on the network a message that is accepted by some legitimate agent as instance of the receive-pattern r_i, and that the intruder can construct this message from the initial token s_0 and from the messages $s_1, , \ldots s_{i-1}$ which have previously been sent by the legitimate agents. The last line of the constraint system states that the intruder can in the end deduce the secret term.

2.3 Constraint Systems

We can define a *constraint system* C as consisting of

- a language L_C of first-order logic, that is a collection of function symbols and of predicate symbols together with their respective arities. The atomic formulae built over this language are called the *atomic constraints* of C;
- a subset of the set of first-order formulae of L_C, called the set of *constraints* of C, comprising at least the set of basic constraints of L_C. The set of *basic constraints* of L_C is the smallest set which contains all atomic constraints of L_C and which is closed under conjunction, existential quantification and renaming of bound variables.
- a first-order structure A_C of the language L_C.

For the execution of a constraint-based mechanism, or for the analysis of its properties, one needs effective means to calculate with constraints. Usually these calculations on constraints can be phrased as the problem of deciding whether certain first-order formulas are valid in the structure of the constraint system:

- The most basic question arising in practically all constraint-based computation mechanisms is to decide whether a given conjunction of atomic constraints is satisfiable. This amounts to deciding whether a formula of the form $\exists x_1, \ldots, x_n \phi$, where ϕ is a constraint of C, is valid in A_C.
- For some applications (constraint programming, simplification in constraint-based rewriting) we need an algorithm for deciding whether a constraint ϕ *entails* a constraint ψ, that is whether all solutions of ϕ are solutions of ψ. This amounts to deciding whether a formula of the form $\forall x_1, \ldots, x_n(\phi \to \psi)$, where ϕ, ψ are constraints of C, is valid in A_C.
- Certain applications in automated theorem proving require indeed that the set of constraints is the set of all first-order formulas over L_C. In this case the problem of deciding satisfiability or entailment of constraints amounts simply to deciding the complete first-order theory of L_C.

It should be noted that for many practical purposes complete algorithms for calculating with constraints either do not exist or have an infeasible complexity (see, for instance, Section 6.2). In this case a constraint-based computation mechanism may be designed in such a way that he can cooperate with incomplete constraint solvers. The question of how incomplete constraint solvers can be integrated in a meaningful way into constraint based formalisms is beyond the scope of these lectures (see for instance [MS98] for the use of incomplete constraint solvers in the context of constraint logic programming). See [SC06] for theoretical considerations and implementation techniques of constraint *propagators* which are used for an incomplete realisation of computationally infeasable constraints.

Chapter 3

Preliminaries

3.1 Power Sets

If X is a set then we write 2^X for the set of subsets of X and $\mathit{FinSets}(X)$ for the set of finite subsets of X. A function $f\colon D \to D$ extends to 2^X by $f(X) := \{f(x) \mid x \in X\}$.

3.2 Orderings

An ordering $\leq$ on a set X is a binary relation on X that is

1. *reflexive*: $x \leq x$ for all $x \in X$;
2. *symmetric*: $x \leq y$ and $y \leq x$ imply $x = y$ for all $x, y \in X$;
3. *transitive*: $x \leq y$ and $y \leq z$ imply $x \leq z$ for all $x, y, z \in X$.

A binary relation $<$ on a set X is a *strict ordering* if it is

1. *irreflexive*: $x < x$ holds for no $x \in X$;
2. *anti-symmetric*: if $x < y$ then $y < x$ does not hold;
3. transitive.

If $<$ is a strict ordering then $< \cup =$ is an ordering, and if $\leq$ is an ordering then $\leq \setminus =$ is a strict ordering, also called the *strict part* of $\leq$. One sometimes writes $y \geq x$ for $x \leq y$ and $y > x$ for $x < y$.

An ordering $\leq$ on X is called *total* if for all $x, y \in X$ at least one of $x \leq y$ or $y \leq x$ holds.

An element $x \in X$ is a *minimum* (resp. *maximum*) of X if $x \leq y$ (resp. $x \geq y$) holds for every $y \in X$. An element x is *minimal* (resp. *maximal*) in X if there is no $y \in X$ with $y < x$ (resp. $x < y$). A maximum is of course a maximal element and a minimum is a minimal element, but the inverse does not necessarily hold (unless the ordering is total, in which case the two notions are in fact equivalent).

Lemma 1 *A set X equipped with a total order $\leq$ is finite iff every non-empty subset of X has a minimum and a maximum.*

Proof: Every non-empty subset of a finite set X obviously has a minimum and a maximum. Let X be infinite and consider the following pseudo-program:

```
L := X
K := ∅
while L has a maximum do
    {Invariant: ∀k ∈ K, l ∈ L  k > l}
    let m := maximum(L) in
        L := L − {m}
        K := K ∪ {m}
end
```

If this program terminates then the final value of L has no maximum, and is non-empty since X is an infinite set. If the program does not terminate then the limit of the successive values of K has no minimum and is of course also non-empty. □

A strict order $<$ on X is called *well-founded* if there is no infinite sequence $x_0, x_1, x_2, \dots$ of elements of X with $x_{i+1} < x_i$ for all i.

3.3 Finite Terms

A *term signature* Σ is a family $\Sigma_0, \dots, \Sigma_n$ of sets of symbols. If $f \in \Sigma_n$ then we say that f has *arity* n. The set of *finite ground terms* $T(\Sigma)$ over Σ is the least set satisfying

- if $c \in \Sigma_0$ then $c \in T(\Sigma)$,
- if $f \in \Sigma_n$ and $t_1, \dots, t_n \in T(\Sigma)$, $n \geq 1$, then $f(t_1, \dots, t_n) \in T(\Sigma)$.

If X is a distinguished set of *variables* disjoint from Σ_0 then the set of *terms* $T(\Sigma, X)$ over Σ and X is defined as $T(\Sigma, X) = T(\Sigma^X)$ where $\Sigma_0^X = \Sigma_0 \cup X$, and $\Sigma_n^X = \Sigma_n$ when $n \geq 1$.

The set of *occurrences* $O(t)$ of a tree $t = f(t_1, \dots, t_n)$ is $\{\epsilon\} \cup \bigcup_{i=1}^n \{iw | w \in O(t_i)\}$. If $\pi \in O(t)$ then the *subterm* $t[\pi]$ of t at π is defined by $t[\epsilon] = t$ and $f(t_1, \dots, t_n)[i\pi] = t_i[\pi]$. The set of *subterms* of a term $t = f(t_1, \dots, t_n)$ is $\texttt{subterms}(t) = \{t\} \cup \bigcup_{i=1}^n \texttt{subterms}(t_i)$. We have that $\texttt{subterms}(t) = \{t[\pi] \mid \pi \in O(t)\}$. A term s is a *proper subterm* of t iff $s \neq t$ and $s \in \texttt{subterms}(t)$. If $t \in T(\Sigma, X)$ then the set of *free variables of* t is $\mathcal{FV}(t) = \texttt{subterms}(t) \cap X$.

The *root symbol* of a term $t = f(t_1, \dots, t_n)$ is $\texttt{root}(t) = f$. The *domain* D_t of a finite term $t \in T(\Sigma)$ is inductively defined by

- $D_c := \{\epsilon\}$ if $c \in \Sigma_0$
- $D_{f(t_1,\dots,t_n)} := \{\epsilon\} \cup \bigcup_{i=1,\dots,n} iD_{t_i}$

A strict ordering relation $\prec$ on $T(\Sigma)$

1. is *monotonous* if $s_i \prec t_i$ implies
   $f(s_1, \dots, s_{i-1}, s_i, s_{i+1}, \dots, s_n) \prec f(s_1, \dots, s_{i-1}, t_i, s_{i+1}, \dots, s_n)$
2. has the *subterm property* if $s \prec t$ for every proper subterm s of t.

A strict ordering on terms that is monotonous and has the subterm property is called a *simplification ordering.*

Theorem 2 ([Der82]) *Any simplification ordering is well-founded, provided the signature is finite.*

A *substitution* is a mapping $\sigma: X \to T(\Sigma, X)$ such that $\sigma(x) = x$ for almost all $x \in X$. Substitutions extend homomorphically to terms, we write $t\sigma$ for the application of the substitution σ to a term t. In particular if x is a variable and t a term then $s\{x \mapsto t\}$ denotes the term obtained from s by replacing all its occurrences of x by t. A term $t \in T(\Sigma, X)$ is *reducible* by a term $l \in T(\Sigma, X)$ if there is a substitution σ such that $l\sigma \in \mathtt{subterms}(t)$. Some authors use the alternative name "encompassement" for this relation.

3.4 Finite and Infinite Terms

Given a term signature Σ a (general) *term* t is given by

- a *domain* $D_t \subseteq 2^{\{1,\ldots,n\}^*}$ such that
  - D_t is non-empty;
  - D_t is prefix-closed, that is if $\pi i \in D_t$ then $\pi \in D_t$;
  - D_t is closed under left brother, that is if $\pi(i+1) \in D_t$ then $\pi i \in D_t$.
- a function $L_t: D_t \to \Sigma$ such that if $n = max\{i \mid \pi i \in D_t\}$ then $L_t(\pi) \in \Sigma_n$

An element $\pi \in D_t$ is called a *leaf* if $\pi\pi' \in D_t$ implies that $\pi' = \epsilon$ (the empty word). Note that an infinite tree may or may not have leaves.

Hence, finite terms are exactly the terms with a finite domain.

3.5 Tree Automata

We recall the definition of tree automata.

Definition 1 *Given a finite signature Σ, a (bottom-up)* tree automaton $\mathcal{A}$ *is given by* (Q, F, Δ) *where*

- Q *is a finite set of* states.
- $F \subseteq Q$ *is called the set of* accepting states.
- Δ *is a set of rewrite rules of the form* $f(q_1, \ldots, q_n) \to q$ *with* $f \in \Sigma_n$, $q_1, \ldots, q_n, q \in Q$.

The automaton A accepts *a tree t iff $t \to^* q \in F$ by the transition rules Δ. The* language $\mathcal{L}(A)$ *is the set of all trees accepted by A.*

Example

The following tree automaton accepts the set of trees representing lists of integers, where integers are build with 0 and s, and lists with *nil* and *cons*:

- $\Sigma_0 = \{0, nil\}$, $\Sigma_1 = \{s\}$, $\Sigma_2 = \{cons\}$
- $Q = \{q_n, q_l\}$
- $F = \{q_l\}$
- $0 \to q_n$, $s(q_n) \to q_n$, $nil \to q_l$, $cons(q_n, q_l) \to q_l$

When writing down tree automata rules we will in the following allow for transition rules that have on the left-hand side any ground term build over the signature $\Sigma \cup Q$. This does not extend the expressivity since we can always translate such a rewrite rule into a set of rules according to Definition 1 by virtue of intermediate state symbols. However, it has the advantage of allowing for a more concise notation of tree automata.

Example

The rule $f(g(a, q_1), h(q_2, b)) \to q_3$ is equivalent to the set of rules $g(a, q_1) \to q'_1$, $h(q_2, b) \to q'_2$, $f(q'_1, q'_2) \to q_3$ where q'_1, q'_2, q'_3 are fresh state symbols.

Tree automata enjoy (almost) all the nice properties of word automata: Closure under Boolean operation, decidability of the emptiness problem, determinisation, minimization, etc. An overview of tree automata can be found in [CDG$^+$97].

3.6 Logic

3.6.1 Syntax

A *first-order language* L consists of

- a set F_L of *function symbols*, and for any function symbol $f \in F_L$ an *arity* which is a non-negative integer. A function symbol with arity 0 is called a *constant symbol*, or simply a *constant*.
- a set P_L of *relation symbols*, and for any relation symbol an *arity* which is a non-negative integer.

Both the sets of function symbols and relation symbols may be empty, finite, or infinite. A language with an empty set of function symbols is called *relational*.

We assume an infinite supply $\mathcal{X}$ of variables that is disjoint to the set of function symbols of any first-order language. Given a language $L = (F_L, P_L)$, the set of (first-order) formulas over L is the smallest set such that

- `True` and `False` are formulas.
- If $P \in P_L$ of arity n and $t_1, \ldots, t_n \in T(F_L, \mathcal{X})$ (see Section 3.3), then $P(t_1, \ldots, t_n)$ is a formula.
- If p_1 and p_2 are formulas then so are $(p_1 \wedge p_2)$ and $p_1 \vee p_2$.
- If p is a formula then so is $\neg p$.
- If p is a formula and $x \in \mathcal{X}$ then $\forall x\, p$ and $\exists x\, p$ are formulas.

Note that equalities $t_1 = t_2$ are formulas only when the language explicitly contains a binary equality symbol $=$.

We will make frequently use of common abbreviations when writing formula:

- we use the fact that conjunction $\wedge$ and disjunction $\vee$ are associative and commutative operators (as justified by the semantics of first-order logic, see below) and drop unnecessary parentheses, writing for instance $p_1 \wedge p_2 \wedge p_3$ instead of $(p_1 \wedge (p_2 \wedge p_3))$.
- we allow to group successive quantifications of the same kind, writing for instance $\exists x_1, \ldots, x_n p$ instead of $\exists x_1 \ldots \exists x_n\, p$.

We will also use the usual definitions for derived propositional operators like $\rightarrow$, $\leftrightarrow$, etc.

We define the *syntactic replacement* of a variable x by a term t in a formula p as follows:

- $P(t_1, \ldots, t_n)\{x \mapsto t\} = P(t_1\{x \mapsto t\}, \ldots, t_n\{x \mapsto t\})$
- $(p_1 \wedge p_2)\{x \mapsto t\} = p_1\{x \mapsto t\} \wedge p_2\{x \mapsto t\}$
- $(p_1 \vee p_2)\{x \mapsto t\} = p_1\{x \mapsto t\} \vee p_2\{x \mapsto t\}$
- $(\neg p)\{x \mapsto t\} = \neg(p\{x \mapsto t\})$
- $(\forall y\, p)\{x \mapsto t\} = \forall y\, (p\{x \mapsto t\{x \mapsto x'\}\})$ where x' is some variable different from y
- $(\exists y\, p)\{x \mapsto t\} = \exists y\, (p\{x \mapsto t\{x \mapsto x'\}\})$ where x' is some variable different from y

$\mathcal{FV}(p)$ denotes the set of *free variables* of a formula p, it is formally defined as

- $\mathcal{FV}(\texttt{True}) = \mathcal{FV}(\texttt{False}) = \emptyset$
- $\mathcal{FV}(P(t_1, \ldots, t_n)) = \mathcal{FV}(t_1) \cup \ldots \cup \mathcal{FV}(t_n)$ (see Section 3.3)
- $\mathcal{FV}(p_1 \wedge p_2) = \mathcal{FV}(p_1 \vee p_2) = \mathcal{FV}(p_1) \cup \mathcal{FV}(p_2)$
- $\mathcal{FV}(\neg p) = \mathcal{FV}(p)$
- $\mathcal{FV}(\forall x\, p) = \mathcal{FV}(\exists x\, p) = \mathcal{FV}(p) \setminus \{x\}$

A formula without free variables is called *closed*, or a *sentence*.

A quantifier-free formula is called *positive* when it does not contain any negation (in its form with all derived propositional operators expanded, like $\rightarrow$). A formula is in *prenex normal form* if it is of the form

$$Q_1 x_1, \ldots, Q_n x_n.\phi$$

where ϕ is a quantifier-free formula, and every of Q_i is a quantifier ($\forall$ or $\exists$). This formula consists in a *quantifier prefix* and the *matrix* which is the quantifier-free formula ϕ. A formula in prenex normal form is called positive when its matrix is positive. The set of formulas Σ_n and Π_n for arbitrary $n \geq 0$ (and a fixed logical language) are defined as follows :

- $\Sigma_0 = \Pi_0$ is the set of quantifier-free formulas (over the given logical language)
- $\Sigma_{n+1} = \{\exists x_1, \ldots, x_k.\psi \mid \psi \in \Pi_n, k \geq 1\}$
- $\Pi_{n+1} = \{\forall x_1, \ldots, x_k.\psi \mid \psi \in \Sigma_n, k \geq 1\}$

3.6.2 Semantics

Given a first-order language L, an *L-structure* $\mathcal{A}$ consists of

- a nonempty set called the *universe* of $\mathcal{A}$, and denoted by the same symbol $\mathcal{A}$,
- for any n-ary function symbol $f \in F_L$ an n-ary function

$$f^{\mathcal{A}}: \underbrace{\mathcal{A} \times \ldots \times \mathcal{A}}_{n} \to \mathcal{A}$$

  this means that $f^{\mathcal{A}} \in \mathcal{A}$ when f is a constant symbol

- for any n-ary predicate symbol an n-ary relation

$$P^{\mathcal{A}} \subseteq \underbrace{\mathcal{A} \times \ldots \times \mathcal{A}}_{n}$$

An $\mathcal{A}$-*assignment* is any total function from $\mathcal{X}$ to an L-structure $\mathcal{A}$. The *interpretation* of a term $t \in T(F_L, \mathcal{X})$ relative to an L-structure $\mathcal{A}$ and an $\mathcal{A}$-assignment α, written $[[t]]_{\mathcal{A},\alpha}$, is defined inductively by

- $[[x]]_{\mathcal{A},\alpha} = \alpha(x)$ when $x \in \mathcal{X}$
- $[[f(t_1, \ldots, t_n)]]_{\mathcal{A},\alpha} = f^{\mathcal{A}}([[t_1]]_{\mathcal{A},\alpha}, \ldots, [[t_1]]_{\mathcal{A},\alpha})$

We define by induction when it is the case that an L-formula p holds in an L-structure $\mathcal{A}$ and relative to an $\mathcal{A}$-assignment α, written $\mathcal{A}, \alpha \models p$:

- $\mathcal{A}, \alpha \models \texttt{True}$
- $\mathcal{A}, \alpha \models P(t_1, \ldots, t_n)$ when $([[t_1]]_{\mathcal{A},\alpha}, \ldots, [[t_1]]_{\mathcal{A},\alpha}) \in P^{\mathcal{A}}$
- $\mathcal{A}, \alpha \models p_1 \wedge p_2$ when $\mathcal{A}, \alpha \models p_1$ and $\mathcal{A}, \alpha \models p_2$
- $\mathcal{A}, \alpha \models p_1 \vee p_2$ when $\mathcal{A}, \alpha \models p_1$ or $\mathcal{A}, \alpha \models p_2$
- $\mathcal{A}, \alpha \models \neg p$ when $\mathcal{A}, \alpha \models p$ does not hold
- $\mathcal{A}, \alpha \models \exists x\, p$ when $\mathcal{A}, \alpha[x/a] \models p$ for some $a \in \mathcal{A}$
- $\mathcal{A}, \alpha \models \forall x\, p$ when $\mathcal{A}, \alpha[x/a] \models p$ for all $a \in \mathcal{A}$

where $\alpha[x/a]$ is the assignment that associates a to x, and $\alpha(y)$ to any variable y different from x.

If p is a sentence then we either have that $\mathcal{A}, \alpha \models p$ for any $\mathcal{A}$-assignment α, or we have that $\mathcal{A}, \alpha \models p$ for no $\mathcal{A}$-assignment α. In the first case we write $A \models p$ and say that p is *valid* in the structure A. The *(first-order) theory* of a structure A is the set of all sentences that are valid in A: $Th(A) = \{p \mid A \models p\}$.

If $x_1, \ldots, x_n$ is a fixed enumeration of $\mathcal{FV}(\phi)$ and A a structure over the language of ϕ then

$$\phi^A = \{(a_1, \ldots, a_n) \in A^n \mid A, \alpha[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] \models \phi \text{ for any } \alpha\}$$

In particular, when ϕ is closed, then we have that

$$\phi^A = \begin{cases} \{()\} & \text{if } A \models \phi \\ \emptyset & \text{if } A \not\models \phi \end{cases}$$

We also write $\mathcal{A} \models \phi[a_1, \ldots, a_n]$ for $(a_1, \ldots, a_n) \in \phi^{\mathcal{A}}$.

Any formula can be rewritten into a logically equivalent formula in prenex normal form. The Σ_n-fragment (resp. Π_n-fragment) of the first-order theory of a structure $\mathcal{A}$ is $Th(A) \cap \Sigma_n$ (resp. $Th(A) \cap \Pi_n$). The problem of deciding the Σ_n-fragment of the theory of A reduces trivially (by negation of the formula) to the problem of deciding the Π_n-fragment, and vice versa. Deciding the theory of A amounts to deciding the union of its Σ_n-fragments (or Π_n-fragments) for all $n \geq 0$ as any formula can be rewritten into prenex normal form.

Let $\phi(x)$ be a formula with free variable x. The *relativisation of quantifiers by $\phi(x)$* is the operation that replaces in a formula every subformula of the form $\exists x \psi$ by $\exists x(\phi(x) \wedge \psi)$, and every subformula $\forall x \psi$ by $\forall x(\phi(x) \rightarrow \psi)$.

3.6.3 Interpreting one structure in another

Intuitively, an interpretation of a structure $\mathcal{B}$ in a structure $\mathcal{A}$ consists in an onto function that associates an element of $\mathcal{B}$ to any n-tuple of elements of $\mathcal{A}$ taken from a designated subset of $\mathcal{A}^n$, such that for every atomic $\mathcal{B}$-formulas there is a corresponding $\mathcal{A}$-formula with respect to the interpretation ([Hod93b], Section 5.3). The designated subset of $\mathcal{A}^n$ is defined as the set of solutions of some formula with n free variables interpreted in the structure $\mathcal{A}$. In the formal definition given below we use the following abbreviation : If $\phi(x_1, \ldots, x_{n*m})$ is a formula and $\mathbf{a}_1, \ldots, \mathbf{a}_m \in \mathcal{A}^n$ then $\mathcal{A} \models \phi[\mathbf{a_1}, \ldots, \mathbf{a_m}]$ stands for $\mathcal{A} \models \phi[\mathbf{a}_1^1, \ldots, \mathbf{a}_1^n, \mathbf{a}_2^1, \ldots, \mathbf{a}_m^n]$.

Definition 2 *Let $\mathcal{A}$ be a K-structure, $\mathcal{B}$ an L-structure, and $n \geq 1$. An (n-dimensional) interpretation of $\mathcal{B}$ in $\mathcal{A}$ is given by*

1. *a K-formula $\delta(x_1, \ldots, x_n)$,*
2. *an onto function $\phi: \delta^{\mathcal{A}} \rightarrow \mathcal{B}$,*
3. *for every m-ary predicate symbol $P \in L$ a K-formula $\Phi_P(x_1, \ldots, x_{n*m})$,*
4. *for every m-ary function symbol $f \in L$ a K-formula $\Phi_f(x_1, \ldots, x_{n*(m+1)})$*

such that

1. *For every m-ary predicate $P \in L$ and $a_1, \ldots a_m \in \delta^{\mathcal{A}}$:*
   $\mathcal{A} \models \Phi_P(a_1, \ldots, a_m)$ iff $\mathcal{B} \models P[\phi(a_1), \ldots, \phi(a_m)]$.
2. *For every m-ary function symbol $f \in L$, $a_1, \ldots a_m, a \in \delta^{\mathcal{A}}$:*
   $A \models \Phi_f[a_1, \ldots, a_m, a]$ iff $\mathcal{B} \models f(\phi(a_1), \ldots, \phi(a_m)) = \phi(a)$.

In this case we say that $\mathcal{B}$ can be interpreted in $\mathcal{A}$.

Example

Let $L = K$ be the language consisting in constants 0 and 1, binary functions $+$ and $*$, and the binary predicate $=$. Let $\mathcal{A}$ be the structure of natural numbers, and $\mathcal{B}$ the structure of rational numbers (with the canonical interpretation of the predicate and function symbols). There is a 2-dimensional interpretation of $\mathcal{B}$ in $\mathcal{A}$ as follows:

1. $\delta(x^1, x^2) = (x_2 \neq 0)$,
2. $\phi(x^1, x^2) = \frac{x_1}{x_2}$,
3. $P_=(x_1, x_2, y_1, y_2) = x_1 * y_2 = y_1 * x_2$,
4. $P_0(x_1, x_2) = (x_1 = 0)$,
5. $P_1(x_1, x_2) = (x_1 = x_2)$,
6. $P_+(x_1, x_2, y_1, y_2, z_1, z_2) = x_1 * y_2 * z_2 + y_1 * x_2 * z_2 = z_1 * x_2 * y_2$
7. $P_*(x_1, x_2, y_1, y_2, z_1, z_2) = x_1 * y_1 * z_2 = z_1 * x_2 * y_2$

Proposition 3 ([Hod93b]) *If $\mathcal{B}$ and be interpreted in $\mathcal{A}$ and $Th(\mathcal{A})$ is decidable then $Th(\mathcal{B})$ is decidable.*

Proof: One associates to any L-formula p a K-formula $\Phi(p)$ as follows, where we restrict ourselves without loss of generality to flat formulas:

$$\begin{array}{rcl}
\Phi(P(x_1, \ldots, x_m)) & = & \Phi_P(x_1^1, \ldots, x_m^n) \\
\Phi(f(x_1, \ldots, x_m) = x_{m+1}) & = & \Phi_f(x_1^1, \ldots, x_{m+1}^n) \\
\Phi(p_1 \wedge p_2) & = & \Phi(p_1) \wedge \Phi(p_2) \\
\Phi(p_1 \vee p_2) & = & \Phi(p_1) \vee \Phi(p_2) \\
\Phi(\neg p) & = & \neg\Phi(p) \\
\Phi(\forall x\, p) & = & \forall x_1, \ldots, x_n(\delta(x^1, \ldots, x^n) \rightarrow \Phi(p)) \\
\Phi(\exists x\, p) & = & \exists x_1, \ldots, x_n\, (\delta(x^1, \ldots, x^n) \wedge \Phi(p))
\end{array}$$

where we assume that there is a function that associates to any variable x n distinguished variables $x^1, \ldots, x^n$. One can easily show that for any L-sentence p one has that

$$\mathcal{B} \models p \qquad \text{iff} \qquad \mathcal{A} \models \Phi(p)$$ □

Exercice 1
Complete the proof of Proposition 3.

We will use this proposition to transfer decidability results of theories (from $\mathcal{A}$ to $\mathcal{B}$), to transfer undecidability results of theories (from $\mathcal{B}$ to $\mathcal{A}$), and to demonstrate the impossibility of interpreting $\mathcal{B}$ in $\mathcal{A}$ (when $Th(\mathcal{A})$ is decidable and $Th(\mathcal{B})$ is not).

3.7 Decidability

There are some undecidable problems that are often more convenient to use in undecidability proofs than the halting problem of Turing machines, namely the Post Correspondence Problem and the halting problem for Minsky Machines.

The Post Correspondence Problem is a quite simple puzzle on words:

Definition 3 *An instance P of the* Post Correspondence Problem (PCP) *is a finite set of pairs of non-empty words over a binary alphabet $\{a,b\}$:*

$$P = \{(p_1, q_1), \ldots, (p_n, q_n)\} \qquad \textit{where } p_i, q_i \in \{a,b\}^+$$

*A pair of words $(v,w) \in \{a,b\}^* \times \{a,b\}^*$ is P-*constructible *if there is a sequence $i_1, \ldots, i_m$, $1 \leq i_j \leq n$, such that*

$$\begin{aligned} v &= p_{i_1} \cdots p_{i_m} \\ w &= q_{i_1} \cdots q_{i_m} \end{aligned}$$

An instance P of the Post Correspondence Problem is solvable *if there is a non-empty word v such that (v,v) is P-constructible. In this case we call the sequence $i_1, \ldots, i_m$ that we used to construct (v,v) a* solution *of P.*

Theorem 4 *([Pos46]) The problem whether a given instance of the Post Correspondence Problem is solvable is undecidable.*

A *Minsky Machine* [Min67] is a simple deterministic two-counter machine whose only instructions are the increment of one of its counters, and a conditional jump instruction. More precisely:

Definition 4 *A Minsky Machine is a numbered sequence $(w_i)_{i=1,\ldots,n}$, for some $n \geq 1$, of instructions where each instruction is one of:*

$$\mathit{incr}(0), \mathit{incr}(1), \mathit{doj}(0,j), \mathit{doj}(1,j)$$

where $1 \leq j \leq n+1$. A configuration of the machine is a triple (pc, c_0, c_1) where $pc \in \{1, \ldots, n+1\}$ and $c_0, c_1 \in \mathbb{N}_0$. The transition rules on configuration are:

$$\begin{array}{lcll} (pc, c_0, c_1) & \to & (pc+1, c_0+1, c_1) & \textit{if } w_{pc} = \mathit{incr}(0) \\ (pc, c_0, c_1) & \to & (pc+1, c_0, c_1+1) & \textit{if } w_{pc} = \mathit{incr}(1) \\ (pc, c_0, c_1) & \to & (pc+1, c_0-1, c_1) & \textit{if } w_{pc} = \mathit{doj}(0,j), c_0 > 0 \\ (pc, c_0, c_1) & \to & (j, c_0, c_1) & \textit{if } w_{pc} = \mathit{doj}(0,j), c_0 = 0 \\ (pc, c_0, c_1) & \to & (pc+1, c_0, c_1-1) & \textit{if } w_{pc} = \mathit{doj}(1,j), c_1 > 1 \\ (pc, c_0, c_1) & \to & (j, c_0, c_1) & \textit{if } w_{pc} = \mathit{doj}(1,j), c_1 = 1 \end{array}$$

A halting configuration *is any configuration $(n+1, c_0, c_1)$.*

Intuitively, a configuration (pc, c_0, c_1) consists of a program counter pc (indicating the instruction to be executed next) and the two counters. An instruction incr(i) increments counter i and moves to the next instruction, and doj(i,j) decrements counter i when it is non-null and then moves to the next instruction, or jumps to instruction j when the counter i is null ("doj" for Decrement Or Jump). The machine starts with instruction 1, and halts when the program counter reaches the value $n+1$.

We obtain as immediate consequence of the theorem stating the simulation of Turing Machines by Minsky Machines (Theorem 14.1-1 in [Min67]):

Theorem 5 *([Min67]) It is undecidable for a given Minsky Machine whether there exists a halting configuration that is reachable from $(1,0,0)$.*

3.8 Arithmetic

The following classical theorem and its proof can be found in any textbook on algebra:

Theorem 6 *(Chinese Remainder Theorem) Let $a_0, \ldots, a_{n-1}$ be pairwise relatively prime integers. For all integers $x_0, \ldots, x_{n-1}$ there exists an integer x such that $x \equiv x_i \pmod{a_i}$ for $0 \leq i < n$.*

Proof: For $0 \leq i < n$ let

$$b_i = \prod_{\substack{0 \leq j < n \\ j \neq i}} a_j$$

Then for every i the numbers a_i and b_i are relatively prime, hence for all i there is a m_i such that $m_i b_i \equiv 1 \pmod{a_i}$. We define $z_i = m_i b_1$ for all i, hence for all i we have that $z_i \equiv 1 \pmod{a_i}$ and $z_i \equiv 0 \pmod{a_j}$ if $j \neq i$. It is now easy to verify that

$$x = x_1 z_1 + \ldots + x_n z_n$$

does what is required. □

3.9 Complexity of the Validity Problem

The following is a classical result:

Theorem 7 *([Sto74]) The validity of quantified Boolean formulas is PSPACE-complete.*

For all the structures that we are interested in, the validity problem of quantified Boolean formulas reduces in polynomial time to the validity problem of formulas in that structure (in fact this reduction fails only for the most trivial structures). Hence when trying to design efficient algorithms for deciding the first-order theory of some structure then PSPACE is the best that we can hope for.

Chapter 4

A Repository of Structures Considered in these Lectures

In this section we define some structures that we will investigate in the following chapters.

4.1 Natural Numbers

We are mainly interested in two first-order languages for natural numbers: *Presburger Arithmetic*, which has addition as the only binary operator, and *Peano Arithmetic* which contains both addition and multiplication. *Successor Arithmetic* is rather a toy example which will serve us to introduce the method of quantfier elimination.

Structure Successor Arithmetic $\mathcal{A}_S$

The language of Successor Arithmetic *consists of*

- *a constant* 0*,*
- *a unary function symbol* s*,*
- *a binary relation symbol* $=$*.*

The structure $\mathcal{A}_S$ *has domain* $\mathbf{N}$*, the set of natural numbers including zero. It assigns the constant symbol* 0 *the number zero,* s *the successor function* $\lambda n.n+1$*, and* $=$ *the equality relation.*

Structure Presburger Arithmetic $\mathcal{A}_{Pr}$

The language of Presburger Arithmetic *consists of*

- *a constant* 0*,*
- *a binary function symbol* $+$*,*
- *a binary relation symbol* $<$*.*

The structure $\mathcal{A}_{Pr}$ *has domain* $\mathbb{N}$*, the set of natural numbers including zero. It assigns the constant symbol* 0 *the number zero,* $+$ *the addition, and* $<$ *the relation "strictly less then".*

As with all the other structures there is some freedom of choice concerning the language of the structure. For instance, we could have included constants $1, 2, \ldots$. These are of course definable by terms composed of the constant 0 and successive applications of s. Equality is definable by

$$x = y \quad :\Leftrightarrow \quad \neg x < y \wedge \neg y < x$$

On the other hand we could of course have defined Presburger Arithmetic with the equality predicate instead of the $<$ predicate, and then define

$$\begin{aligned} x = 1 \quad &:\Leftrightarrow \quad x \neq 0 \wedge \forall y, z\, (x = y + z \rightarrow (y = x \vee z = x)) \\ x < y \quad &:\Leftrightarrow \quad \exists z\, y = x + z + 1 \end{aligned}$$

A minimal language for Presburger Arithmetic consists of just $+$ and $<$. In this minimal system we could then define

$$\begin{aligned} x = 0 \quad &:\Leftrightarrow \quad \neg \exists y\, y < x \\ x = 1 \quad &:\Leftrightarrow \quad 0 < x \wedge \forall y, z\, (0 < y \wedge y < z \rightarrow x < z) \end{aligned}$$

The actual choice of a language may be motivated by what is needed in the application of interest, or by what is convenient in calculating with formulas. The right choice of language has a great influence of the design of algorithms calculating with formulas, and often one has to change language in order to build a good algorithm (see the discussion in Section 7.1).

Structure Skolem Arithmetic $\mathcal{A}_{Sk}$

The language of Skolem Arithmetic *consists of*

- *a constant* n *for every non-zero natural number*
- *a binary function symbol* $*$
- *a binary relation symbol* $=$

The structure $\mathcal{A}_{Sk}$ has domain $\mathbf{N}^+$, the set of non-zero natural numbers. A constant symbol n is interpreted as the natural number n, the function $*$ as multiplication, and $=$ as equality.

Structure Presburger Arithmetic with Divisibility $\mathcal{A}_{Pr|}$

The language of Presburger Arithmetic with Divisibility *consists of*

- *a constant* 0*,*
- *a unary function symbol* s*, and a binary function symbol* $+$*,*
- *a binary relation symbol* $<$*, and a binary relation symbol* $|$

The structure $\mathcal{A}_{Pr}$ *has domain* $\mathbf{N}$*, the set of natural numbers including zero. It assigns the constant symbol* 0 *the number zero,* s *the successor function* $\lambda n.n+1$*,* $+$ *the addition,* $<$ *the relation "strictly less then", and* $x \mid y$ *the relation "*x *divides* y*".*

Structure Peano Arithmetic $\mathcal{A}_{Pe}$

The language of Peano Arithmetic *consists of*

- *a constant* 0
- *two binary function symbols* $+$ *and* $*$
- *binary relation symbols* $<$, $\leq$ *and* $=$

The structure $\mathcal{A}_{Pe}$ *has domain* $\mathbf{N}$*, the set of natural numbers including zero. The constant* 0 *and the function and relation symbols take the usual meaning.*

Note that the term *Peano Arithmetic* is usually used to denote a particular set of first-order axioms of the structure $\mathcal{A}_{Pe}$. Their set of consequences does, by Theorem 89, not cover the whole theory of our structure $\mathcal{A}_{Pe}$.

Structure Real Arithmetic $\mathcal{A}_{\mathbb{R}}$

The language of Real Arithmetic *consists of*

- *the constants* 0, 1, *and* -1
- *two binary function symbols* $+$ *and* $*$
- *binary relation symbols* $<$, $\leq$ *and* $=$

The structure $\mathcal{A}_{\mathbb{R}}$ *has domain* $\mathbf{R}$*, the set of real numbers. The constants, function and relation symbols take the usual meaning.*

Structure Rational Arithmetic $\mathcal{A}_{\mathbb{Q}}$

The language of Rational Arithmetic *consists of*

- *the constants* 0, 1, *and* -1
- *two binary function symbols* $+$ *and* $*$
- *binary relation symbols* $<$, $\leq$ *and* $=$

The structure $\mathcal{A}_{\mathbb{Q}}$ *has domain* $\mathbf{Q}$*, the set of rational numbers. The constants, function and relation symbols take the usual meaning.*

4.2 Words

Structure Words $\mathcal{A}_W$

Given an alphabet Σ *the language of the structure of words consists of*

- *a constant* c *for every letter* $c \in \Sigma$*, and a constant* ϵ
- *a binary operator* $\cdot$*. Usually we abbreviate* $x \cdot y$ *as* xy*.*
- *the equality predicate.*

The structure $\mathcal{A}_W$ has domain Σ^*, the set of finite words over Σ. The constant ϵ is interpreted as the empty word, a constant c as the word consisting of the single letter c, the function symbol $\cdot$ as concatenation, and $=$ as equality.

If the alphabet is not specified then the alphabet $\{a, b\}$ is assumed.

Note that we could have defined the constant ϵ in the rest of the structure by

$$x = \epsilon \qquad := \qquad xx = x$$

Note that inequations can be expressed equivalently by positive formulas

Proposition 8 *In case of a finite alphabet Σ, any inequation in $\mathcal{A}_W$ is equivalent to an existential positive formula.*

Proof: Two words differ iff one of them is lexicographically smaller than the other. Hence, given a Σ_1-sentence we can eliminate all inequations $x \neq y$ by

$$\begin{array}{ll} \exists x', y', z' \quad (& x = yay' \vee x = yby' \vee (x = z'ax' \wedge y = z'by') \vee \\ & y = xax' \vee y = xbx' \vee (y = z'ay' \wedge x = z'bx') \quad) \end{array}$$

In this formula, the first three disjuncts stand for "x is lexicographically smaller then y", the last three for "y is lexicographically small than x". □

The structures $\mathcal{A}_{WS2S}$ and $\mathcal{A}_{S2S}$ are usually presented in the context of second-order logic. We deviate from this tradition in defining them here as two-sorted first-order structures. In fact, the distinction between first-order and higher-order is only important when speaking about *classes* of structures; as long as one is interested in one single structure it is just a question of notation.

Structure Weak Monadic Structure of Two Successor Functions $\mathcal{A}_{WS2S}$

This structure has two sorts s and w. The language contains

- *a constant $\epsilon \colon w$*
- *unary functions $0 \colon w \to w$ and $1 \colon w \to w$.*
- *binary predicate symbols $\in \colon w \times s$, $=_w \colon w \times w$ and $=_s \colon s \times s$.*

The domain of $\mathcal{A}_{WS2S}$ of sort w is $\{0,1\}^*$, and its domain of sort s is FinSets$(\{0,1\}^*)$. The function symbols are interpreted as $0(v) = v0$ and $1(v) = v1$, $\in$ as set membership, and the two equality symbols as equality.

We usually write $x0$ (resp. $x1$) instead of $0(x)$ (resp. $1(x)$). Furthermore we do not write the indices of the equality symbols. We avoid declaring the sort of a variable by writing variables ranging over sort w in lower case, and variables ranging over sort s in upper case.

Structure Monadic Structure of Two Successor Functions $\mathcal{A}_{S2S}$

The language of $\mathcal{A}_{S2S}$ is the same as of $\mathcal{A}_{WS2S}$, and the interpretation is the same except that the domain of sort s is now $2^{\{0,1\}^*}$.

We can express in $\mathcal{A}_{S2S}$ that x is a prefix of y by

$$x \leq y \quad \Leftrightarrow \quad \forall X \, (x \in X \wedge (\forall z \, (z \in X \to (z0 \in X \wedge z1 \in X))) \to y \in X)$$

Using the prefix ordering we can define the lexicographic ordering on words by

$$x \preceq y \quad \Leftrightarrow \quad x \leq y \vee \exists z\,(z0 \leq x \wedge z1 \leq y)$$

Since $\preceq$ is a total order on words we can express the fact that a set of words is finite by

$$\begin{aligned} Fin(X) \quad \Leftrightarrow \quad & \forall Y\,(Y \subseteq X \wedge Y \neq \emptyset \rightarrow \\ & \exists x\,(x \in Y \wedge \forall z\,(z \in Y \rightarrow z \preceq x)) \wedge \\ & \exists x\,(x \in Y \wedge \forall z\,(z \in Y \rightarrow z \succeq x))) \end{aligned}$$

This definition is justified by Lemma 1. This allows to reduce the decision problem of the first-order theory of $\mathcal{A}_{WS2S}$ to the the decision problem of the first-order theory of $\mathcal{A}_{S2S}$ by relativizing all quantifiers with *Fin*.

In fact one can, for any regular language R, express in $\mathcal{A}_{S2S}$ the predicate "$x \in R$" (see, for instance, [Tho90]). The same holds for the structure $\mathcal{A}_{WS2S}$.

In order to express the prefix relation in $\mathcal{A}_{WS2S}$ we can not use the same definition as we did above for $\mathcal{A}_{S2S}$ since it relies on the existence of the infinite sets of words. A definition which works for $\mathcal{A}_{WS2S}$ as well as for $\mathcal{A}_{S2S}$ is

$$x \leq y \quad \Leftrightarrow \quad \forall X\,(y \in X \wedge (\forall z\,((z0 \in X \vee z1 \in X) \rightarrow z \in X)) \rightarrow x \in X)$$

4.3 Ground Terms

4.3.1 Equality and Equality Modulo

Ground terms are a classical data structure for instance in Logic Programming and in Automated Deduction.

Structure $\mathcal{A}_H$

Given a term signature Σ, the language of the structure of Herbrand *consists of*

- *the constants from Σ_0*
- *the function symbols from Σ_i, $i \geq 1$, with their respective arity*
- *the equality predicate $=$.*

The domain of the structure $\mathcal{A}_H$ is the set of finite Σ-terms $T(\Sigma)$. A function symbol f is interpreted as the function that given terms $t_1, \ldots, t_n$ yields the term $f(t_1, \ldots, t_n)$. The predicate symbol $=$ is interpreted as (syntactic) equality.

In many applications it is interesting to study terms modulo some equational theory. The most important equational theory is associativity-commutativity (short: AC):

Structure $\mathcal{A}_{T/AC}$

Given a tree signature Σ where in addition some symbols of Σ_2 may be tagged with status AC, the language of the structure of ground terms modulo AC *consists of*

- *the constants from Σ_0*

- *the function symbols from Σ_i, $i \geq 1$, with their respective arity*
- *the equality predicate $=$.*

The domain of the structure $\mathcal{A}_{T/AC}$ is the set of finite Σ-terms $T(\Sigma)$. A function symbol f is interpreted as the function that given terms $t_1, \ldots, t_n$ yields the term $f(t_1, \ldots, t_n)$. The predicate symbol $=$ is interpreted as equality module associativity and commutativity of the binary function symbols that are tagged as AC.

For instance if a, b, c are constants, f a unary function symbol and $+$ a binary AC function symbol then $\mathcal{A}_{T/AC} \models (f(a) + b) + c = (c + b) + f(a)$.

4.3.2 Structural Subtype Ordering

Let Σ be a signature and $<$ an ordering on Σ_0, the set of constants of Σ. The *structural subtype ordering* is the least binary relation $\prec_{\text{struct}}$ on $T(\Sigma)$ such that

1. $c \prec_{\text{struct}} d$ if $c, d \in \Sigma_0$ and $c < d$;
2. $f(s_1, \ldots, f_n) \prec_{\text{struct}} f(t_1, \ldots, t_n)$ if $s_i \prec_{\text{struct}} t_i$ for all i.

Note that if $s \prec_{\text{struct}} t$ then $D_s = D_t$. The following alternative definition of $\prec_{\text{struct}}$ also works for infinite trees, and is equivalent to the above definition in case of finite trees: $s \prec_{\text{struct}} t$ if

1. $D_s = D_t$;
2. for every $\pi \in D_s$ that is a leaf: $L_s(\pi) < L_t(\pi)$;
3. for every $\pi \in D_s$ that is a not a leaf: $L_s(\pi) = L_t(\pi)$.

Structure $\mathcal{A}_{\text{SSub}}$ and $\mathcal{A}_{\text{FSSub}}$

Given a finite term signature Σ and an ordering $<$ on Σ_0, the language of the structure of lpo *consists of*

- *the constants from Σ_0*
- *the function symbols from Σ_i, $i \geq 1$, with their respective arity*
- *the equality predicate $=$*
- *the binary symbol $\prec$*

The domain of the structure $\mathcal{A}_{\text{SSub}}$ is the set of finite or infinite Σ-terms $T(\Sigma)$. A function symbol f is interpreted as the function that given terms $t_1, \ldots, t_n$ yields the term $f(t_1, \ldots, t_n)$. The predicate symbol $=$ is interpreted as (syntactic) equality. The symbol $\prec$ is interpreted as the structural subtyping relation $\prec_{\text{struct}}$ induced by Σ and $<$.

The structure $\mathcal{A}_{\text{FSSub}}$ is obtained from $\mathcal{A}_{\text{SSub}}$ by restricting the domain to finite terms only.

4.3.3 Non-Structural Subtype Ordering

Let Σ be a signature equiped with an ordering. The *non-structural subtype ordering* is the least binary relation $\prec_{\text{non-struct}}$ on $T(\Sigma)$ such that

1. $f(\ldots) \prec_{\text{non-struct}} g(\ldots)$ if $f < g$;
2. $f(s_1, \ldots, f_n) \prec_{\text{non-struct}} f(t_1, \ldots, t_n)$ if $s_i \prec_{\text{non-struct}} t_i$ for all i.

Note that, in contrast to the structural subtype ordering, we may have that $s \prec_{\text{non-struct}} t$ then $D_s \neq D_t$ (hence the name of this ordering — the terms are not required to have the same structure). Usually one considers orderings on Σ in which any two different non-constants are incomparable. The following alternative definition of $\prec_{\text{struct}}$ also works for infinite trees, and is equivalent to the above definition in case of finite trees: $s \prec_{\text{struct}} t$ if for all $\pi \in D_s \cap D_t$:

- If $L_s(\pi) \neq L_t(\pi)$
- And $L_s(\pi') = L_s(\pi')$ for all proper subwords $\pi' < \pi$
- Then $L_s(\pi) < L_t(\pi)$.

In other words, we have $s \prec_{\text{non-struct}} t$ iff at all topmost disagreement positions the relation $<$ holds.

Structure $\mathcal{A}_{\text{NSub}}$ and $\mathcal{A}_{\text{FNSub}}$

The structures $\mathcal{A}_{\text{NSub}}$ and $\mathcal{A}_{\text{FNSub}}$ are defined as $\mathcal{A}_{\text{SSub}}$ and $\mathcal{A}_{\text{FSSub}}$, respectively, except that the non-structural subtyping relation is used instead of the structural one.

4.3.4 Lexicographic Path Ordering

Let Σ be a signature and $<$ an ordering (called *precedence*) on the symbols of Σ. The *lexicographic path ordering*, or short *lpo*, $\succ_{\text{lpo}}$ induced by $\Sigma, <$ is a relation on $T(\Sigma)$ inductively defined as follows: $s = f(s_1, \ldots, s_m) \succ_{\text{lpo}} g(t_1, \ldots, t_n) = t$ if one of the following holds:

1. $s_i \succ_{\text{lpo}} t$ or $s_i = t$ for some i with $1 \leq i \leq m$ or
2. $f > g$ and $s \succ_{\text{lpo}} t_j$ for all j with $1 \leq j \leq n$ or
3. $f = g$ and there is an i with $1 \leq i \leq n$ and
   (a) $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and
   (b) $s_i \succ_{\text{lpo}} t_i$ and
   (c) $s \succ_{\text{lpo}} t_{i+1}, \ldots, s \succ_{\text{lpo}} t_n$

Theorem 9 ([Der87]) *If $<$ is a total order on a finite signature Σ then the induced lpo $\prec_{\text{lpo}}$ has the following properties:*

- *it is a strict total order on $T(\Sigma)$*
- *it is well-founded: there is no infinite sequence $t_1 \succ_{\text{lpo}} t_2 \succ_{\text{lpo}} t_3 \succ_{\text{lpo}} \cdots$*

- *it is monotonous,*
- *it is compatible with the subterm ordering.*

Structure $\mathcal{A}_{\text{lpo}}$

Given a term signature Σ and a precedence $<$, the language of the structure of lpo *consists of*

- *the constants from Σ_0*
- *the function symbols from Σ_i, $i \geq 1$, with their respective arity*
- *the equality predicate $=$*
- *the binary symbol $\prec$*

The domain of the structure $\mathcal{A}_{\text{lpo}}$ is the set of finite Σ-terms $T(\Sigma)$. A function symbol f is interpreted as the function that given terms $t_1, \ldots, t_n$ yields the term $f(t_1, \ldots, t_n)$. The predicate symbol $=$ is interpreted as (syntactic) equality. The symbol $\prec$ is interpreted as the lpo $\prec_{\text{lpo}}$ induced by Σ and $<$.

4.3.5 Knuth-Bendix Ordering

Let Σ be a signature and $<$ an ordering (called *precedence*) on the symbols of Σ, and let $w\colon \Sigma \to \mathbb{N}$ be a so-called *weight function*. The weight function extends to ground terms by $w(f(t_1, \ldots, t_n)) = w(f) + \Sigma_{i=1}^{n} w(t_i)$. The *Knuth-Bendix ordering*, or short *kbo*, $\prec_{\text{kbo}}$ induced by $\Sigma, <, w$ is a relation on $T(\Sigma)$ inductively defined as follows: $s = f(s_1, \ldots, s_m) \prec_{\text{kbo}} g(t_1, \ldots, t_n) = t$ if one of the following holds:

1. $w(s) < w(t)$;
2. $w(s) = w(t)$ and $f < g$;
3. $w(s) = w(t)$ and $f < g$ and there is an i with $1 \leq i \leq n$ and
   (a) $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and
   (b) $s_i \prec_{\text{kbo}} t_i$

Theorem 10 *[KB70, BN98] The order $\prec_{\text{kbo}}$ is a well-founded total order on $T(\Sigma)$ that is monotonous and is compatible with the subterm ordering, provided that*

1. *$<$ is a total order and Σ finite,*
2. *$w(c) > 0$ for any constant $x \in \Sigma_0$,*
3. *if there is a unary function $f \in \Sigma_1$ with $w(f) = 0$ then f is maximal in Σ w.r.t. $<$.*

Since we require that the precedence is total there can be, as consequence of the last condition, at most one unary function symbol with weight 0.

Structure $\mathcal{A}_{\text{kbo}}$

Given a term signature Σ, a precedence $<$, and a weight function w, the language of the structure of lpo *consists of*

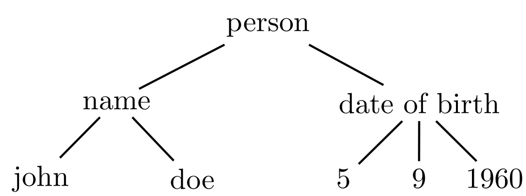


Figure 4.1: Classical terms to represent data.

- *the constants from* Σ_0
- *the function symbols from* Σ_i, $i \geq 1$, *with their respective arity*
- *the equality predicate* $=$
- *the binary symbol* $\prec$

The domain of the structure $\mathcal{A}_{\text{kbo}}$ *is the set of finite* Σ*-terms* $T(\Sigma)$*. A function symbol* f *is interpreted as the function that given terms* $t_1, \ldots, t_n$ *yields the term* $f(t_1, \ldots, t_n)$*. The predicate symbol* $=$ *is interpreted as (syntactic) equality. The symbol* $\prec$ *is interpreted as the kbo* $\prec_{\text{kbo}}$ *induced by* Σ*,* $<$*, and* w*.*

4.4 Feature Trees

The constraint system Herbrand, that is first-order terms, is often used to model symbolic data. For many applications, however, this constraint system turns out to be too inflexible. This inflexibility has two sources, one is the use of classical first-order trees as underlying data model, and the other is the particular set of atomic constraints provided by the Herbrand constraint system.

The problem with using classical trees as a data model is that the subtrees of a tree described in the Herbrand constraint system are numbered, that is we always have to remember the meaning that we have in mind with the first, second, third, ... subtree of a tree. In the example of Figure 4.1 we have to remember that in the subtree labeled *name* the first component is the first name and the second component is the family name, and that the date of birth is stored in the format day-month-year and not, for instance, month-day-year. In many applications the use of symbolic names to denote subtrees is much more natural. This is illustrated in Figure 4.2 where we used a tree with labeled edges, a so-called *feature tree*, to represent some data about a movie (an exact definition of feature trees will follow soon). Using arbitrary names to denote components recalls of course the notion of *record* which is well-known from many programming languages.

The second problem with the Herbrand constraint system comes from its choice of atomic constraints. If we write a constraint $x = f(y, z)$ then we are expressing several things at the same time:

1. that the root of the tree x is labeled with the symbol f,
2. that there are exactly two subtrees at the root of the tree f,
3. that the first subtree of x is denoted by y, and that the second subtree of x is denoted by z.

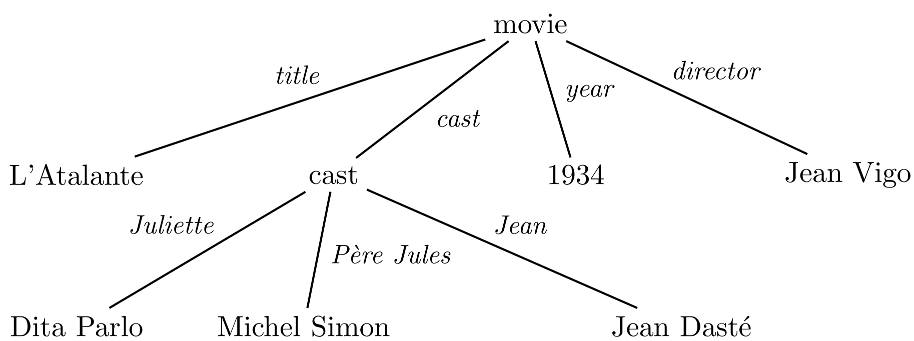


Figure 4.2: Feature trees to represent data.

That the root of x is labeled by f is easily expressed in Herbrand by using an existential quantifier:

$$\exists y, z\; x = f(y, z)$$

assuming that f is known to be binary. The other properties, however, are much more cumbersome to express in Herbrand since they require the use of *disjunctions*:

$$\bigvee_{f \in \Sigma_2} \exists y, z\; x = f(y, z) \qquad x \text{ has two subtrees}$$

$$\bigvee_{f \in \Sigma_2} x = f(y, z) \qquad \text{the first (second) subtree of } x \text{ is } y\ (z)$$

where Σ_2 is the set of binary function symbols. This is impractical, if only for the reason that in applications the signature might not be known in advance since we want to have the freedom to use any function symbol in the course of computation (imagine a system where the user interactively enters data items, using any function symbols he likes). Not knowing the signature in advance is usually modeled by assuming the signature to be infinite, for instance to be the set of all words formed over some alphabet. In this case the above disjunctions are not even formulas of first-order predicate logic.

This is where *feature constraints* come in: Atomic feature constraints are more fine-grained than Herbrand-equations. In the constraint system CFT, for example, there are exactly the three kinds of atomic constraints that correspond to the three properties listed above: *label constraints* Ax (sometimes called *sort constraints*) to specify the symbol at the root node of a tree, *arity constraints* $x\{f_1, \ldots, f_n\}$ to specify the labels of the edges departing from the root, and *subtree* constraints $x[f]y$ to specify the subtree of a tree at a specific edge.

We assume an infinite set $\mathcal{F}$ of *features* ranged over by f, g and an infinite set $\mathcal{L}$ of *labels* ranged over by A, B.

A *path* π is a word of features. The *empty path* is denoted by ε and the free-monoid concatenation of paths π and π' as $\pi\pi'$. We have $\varepsilon\pi = \pi\varepsilon = \pi$. A path π' is called a *prefix of* π if $\pi = \pi'\pi''$ for some path π''. A *tree domain* is a non-empty prefix closed set of paths.

A *feature tree* τ is a pair $\langle D, L\rangle$ consisting of a tree domain D and a (total) function $L : D \to \mathcal{L}$ that we call *labeling function* of τ. Given a feature tree τ, we write D_τ for its tree domain and L_τ

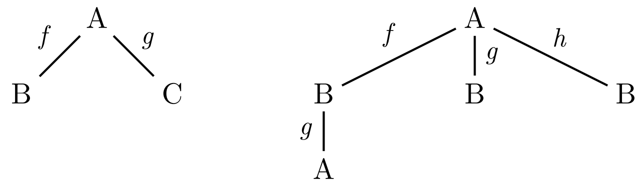


Figure 4.3: Some simple feature trees.

for its labeling function. A feature tree is *finite* if its tree domain is finite, and *infinite* otherwise. A *node of* τ is an element of D_τ. A node π of τ is *labeled with* a if $\langle \pi, a\rangle \in L_\tau$. A node of τ is unlabeled if it is not labeled with any a. The *root* of τ is the node ε. The *root label* of τ is $L_\tau(\varepsilon)$, and $f \in \mathcal{F}$ is a *root feature* of τ if $f \in D_\tau$. For instance, $\tau_0 = \langle\{\epsilon, f, g\}, \{(\epsilon, A), (f, B), (g, C)\}\rangle$ is a feature tree with domain $D_{\tau_0} = \{\epsilon, f, g\}$ and $L_{\tau_0} = \{\langle \epsilon, A\rangle, \langle f, B\rangle, \langle g, C\rangle\}$, depicted to the left of Figure 4.3.

Given a tree τ with $\pi \in D_\tau$, we write as $\tau[\pi]$ the subtree of τ at path π; formally $D_{\tau[\pi]} = \{\pi' \mid \pi\pi' \in D_\tau\}$ and $L_{\tau[\pi]} = \{\langle \pi', a\rangle \mid \langle \pi\pi', a\rangle \in L_\tau\}$.

Structure Feature Trees $\mathcal{A}_{FT}$

The language of FT consists of

- *an binary equality predicate, noted* $x = y$,
- *unary* label predicates, *noted* Ax, *where* A *is a label,*
- *binary* subterm predicates, *noted* $x[f]y$, *where* f *is a a feature,*
- *and unary* undefinedness predicates, *noted* $xf\uparrow$, *where* f *is a feature.*

The universe of the structure $\mathcal{A}_{FT}$ consists of all feature trees. The constraints are interpreted as follows:

- $=$ *is the equality predicate;*
- $a\tau$ *holds iff* $L_\tau(\epsilon) = a$;
- $\tau_1[f]\tau_2$ *holds iff* $\tau_2 = \tau_1[f].$;
- $\tau f\uparrow$ *holds iff* $f \notin D_\tau$.

The predicate $\tau f\uparrow$ could have been defined in the structure itself, by way of the equivalence

$$xf\uparrow \quad\Leftrightarrow\quad \neg\exists y\, x[f]y$$

Structure Feature Trees $\mathcal{A}_{CFT}$

The language of CFT consists of

- *an equality predicate, noted* $x = y$,
- label predicates, *noted* Ax, *where* A *is a label,*

- subterm predicates, *noted* $x[f]y$, *where* f *is a a feature,*

- arity predicates, *noted* xF, *where* F *is a finite set of features.*

The universe of the structure $\mathcal{A}_{CFT}$ *consists of all feature trees. The constraints are interpreted as follows:*

- $a\tau$ *holds iff* $L_\tau(\epsilon) = a$.
- $\tau_1[f]\tau_2$ *holds iff* $\tau_2 = \tau_1[f]$.
- τF *holds iff* $D_\tau \cap \mathcal{F} = F$.

4.4.1 Feature trees with subsumption

Subsumption is a partial order on feature trees:

Structure Feature Trees $\mathcal{A}_{FT\leq}$ with Subsumption

The language of $\mathcal{A}_{FT\leq}$ *consists of*

- *an ordering predicate, noted* $x \leq y$
- label predicates, *noted* Ax, *where* A *is a label*
- subterm predicates, *noted* $x[f]y$, *where* f *is a a feature*

The universe of the structure $\mathcal{A}_{FT\leq}$ *consists of all feature trees. The constraints are interpreted as follows:*

- $a\tau$ *holds iff* $L_\tau(\epsilon) = a$.
- $\tau_1[f]\tau_2$ *holds iff* $\tau_2 = \tau_1[f]$.
- $\sigma \leq \tau$ *holds iff* $D_\sigma \subseteq D_\tau$ *and* $L_\sigma \subseteq L_\tau$.

The ordering relation allows us to recover the equality relation between feature trees, since $\tau = \sigma$ iff $\tau \leq \sigma \wedge \sigma \leq \tau$.

We can construct, for any formula ϕ, formulas $\min x\phi$ and $\max x\phi$ expressing that x *is minimal (maximal) with the property* ϕ:

$$\begin{aligned} \min x\phi &:= \phi \wedge \neg\exists y\,(\phi\{x \mapsto y\} \wedge y \leq x) \\ \max x\phi &:= \phi \wedge \neg\exists y\,(\phi\{x \mapsto y\} \wedge y \geq x) \end{aligned}$$

Here, y is a fresh variable not occurring in ϕ, and $\phi\{x \mapsto y\}$ denotes the formula where every free occurrence of x is replaced by y. Typically, x occurs free in ϕ but this is not required. Note that, in contrast to $\forall x$ and $\exists x$, $\min x$ and $\max x$ are no variable binders that restrict the scope of the variable x; hence x is free in $\min x\phi$ and in $\max x\phi$ if it is free in ϕ. The formulas $\min x\phi$ and $\max x\phi$ do *not* state that x denotes the smallest (resp. greatest) tree satisfying ϕ, in fact such a tree may not exist. This difference is important for the formula $\mathsf{atom}(x)$ defined below.

Example

The sentence $\exists x\,(\min x\top)$ is valid in $\mathcal{A}_{FT\leq}$. In fact there exist infinitely many values that satisfy it: the formula is satisfied exactly be the feature trees that consist of the root node only, that is exactly by the feature trees of the form $(\{\epsilon\}, \{(\epsilon, A\})$ with $A \in \mathcal{L}$. There is no *smallest* element satisfying it. In contrast, $\exists x\ (\min x A(x))$ has only one solution (which hence is the smallest solution).

We hence define:

$$\mathsf{atom}(x) := \min x\top$$

Example

The formula $\min x(x[0]x \wedge x[1]x)$ is satisfied in $\mathcal{A}_{FT\leq}$ by any full binary carrying everywhere the same label.

The formula $x \sim y$ reads *x and y are consistent*, that is that they coincide on the intersection of their tree domains:

$$x \sim y := \exists z\ (x \leq z \wedge y \leq z)$$

We can also express that x and y have the same root label, by

$$\mathsf{same\text{-}root\text{-}label}(x, y) := \exists z\ (z \leq x \wedge z \leq y)$$

The next formula will be used in the undecidability proof of Section 12.1.6:

A tree τ satisfies this formula iff $\{\epsilon, \mathtt{c}\} \subseteq D_\tau \subseteq \{\mathtt{c}\}^*$ and all its nodes are labeled bit $\mathtt{t}$):

$$\mathsf{string}(x) := \exists y\ (\min y(\mathtt{t}(y) \wedge y[\mathtt{c}]y) \wedge x \leq y)$$

In general, we have that

Lemma 11 *The formula $\exists y\ (x[f]y \wedge y \leq x)$ is satisfied by τ iff $f \in D_\tau$ and whenever $n \geq m, f^n, f^m \in D_\tau$, then*

$$\tau[f^n] \leq \tau[f^m]$$

The feature trees depicted on Figure 4.3, for instance, are incomparable in the subsumption order. However, if we change in the left tree the label of the right son from C to B then the left tree becomes smaller then the right tree.

Note that $\mathcal{A}_{FT\leq}$ is an extension of $\mathcal{A}_{FT}$ since we can express an equality $x = y$ in $\mathcal{A}_{FT\leq}$ simply by $x \leq y \wedge y \leq x$.

4.4.2 Feature trees with first-class features

The feature constraint system CFT presented above has one serious limitation: feature symbols are fixed symbols. Variables denote feature trees but not feature symbols! It would be nice if variables could also denote feature symbols, that is if feature symbols would be given *first-class status*.

We thus define the feature constraint system F: The logical language now contains two sorts: features and trees. Feature symbols are now constants of sort feature, and subtree selection $x[v]y$ is now a ternary predicate symbol of profile (trees, features, trees). Label and Arity constraints are as in CFT, and there is an equality predicate on both sorts. Here is an example of an F-constraint:

$$x\{f, g\} \wedge x[v]y \wedge v = f \wedge Ay$$

Structure Feature Trees $\mathcal{A}_F$ with First-Class Features

The language of $\mathcal{A}_F$ consists of two sorts f and t. The language contains

- *unary* label predicates *of sort t, noted Ax, where A is a label;*
- *a ternary* subterm predicate*, noted $x[z]y$, of sort $t \times f \times t$,*
- *unary* arity predicates*, noted xF, where F is a finite set of features,*
- *binary predicate symbols $=_f\colon f \times f$ and $=_t\colon t \times t$.*

The domain of $\mathcal{A}_F$ of sort f is the set of feature symbols FS. The domains of $\mathcal{A}_F$ of sort t is the set of feature trees. $=_f$ and $=_t$ are the equality predicates, label and arity predicates are interpreted as in $\mathcal{A}_{CFT}$, and $\tau_1[\pi]\tau_2$ holds iff $\tau_2 = \tau_1[\pi]$.

Note that the arity predicate could have been defined in $\mathcal{A}_F$ by

$$x\{f_1, \ldots, f\} \quad \Leftrightarrow \quad \forall z \left((\exists y\, x[z]y) \leftrightarrow \bigvee_{i=1,\ldots,n} z = f_i \right)$$

Chapter 5

Syntactic Methods: Deterministic Resolution of Constraints

5.1 Syntactic Methods

Syntactic methods work by rewriting parts of formulas. The machinery employed here is the classical term rewriting mechanism applied to formulas. We usually consider formulas as terms modulo associativity and commutativity of conjunction and disjunction, and modulo swapping of quantifiers of the same kind (that is, $\exists x\, \exists y\, \phi$ is considered equivalent to $\exists y\, \exists x\, \phi$). Given such a rewrite system, we usually have to prove three properties:

1. The system must be terminating.
2. The transformations must be equivalence transformations in the structure under consideration.
3. Depending on the actual problem we might have to prove that the normal forms of the rewrite system fall into some class of formulas.

The proofs of the first two properties (in particular the termination proof) can be simplified by imposing an appropriate strategy on the rewriting. As long as one is only interested in obtaining one algorithm to solve the problem at hand (this will be the case for these lecture notes) one can freely impose strategies as convenient for the proofs. On the other hand one might also be interested in describing a class of algorithms as large as possible, thus imposing strategic constraints only if absolutely unavoidable (this is, for instance the approach of [CL89]).

It is important to remark that confluence of the rewrite system has no importance for us since the rewrite steps are always equivalence transformations in the structure that we are interested in, and do not correspond to a don't-know nondeterminism as in the classical term rewriting setup. In fact it is rather the tool-box for proving termination of rewrite theory that we will profit from.

In this chapter we will look in particular at the use of syntactic methods to solve constraints over feature trees.

5.2 CFT Constraints

5.2.1 Satisfiability of CFT Constraints

The set of CFT constraints is the smallest set of $\mathcal{A}_{CFT}$ formulas that contains all the atomic formulas and that is closed under conjunction and existential quantification. Since existential quantifiers are not relevant when testing satisfiability of constraints we can simply ignore them for the moment, hence for the purpose of checking satisfiability of constraints we assume that a constraint is just a conjunction of atomic constraints.

Definition 5 *A CFT constraint c is said to be a* solved form *if it contains no equation and*

1. *if $x[f]y \in c$ and $x[f]z \in c$ then $y = z$*
2. *if $Ax \in c$ and $Bx \in c$ then $A = B$*
3. *if $xF \in c$ and $x[f]y \in c$ then $f \in F$*
4. *if $xF \in c$ and $xG \in c$ then $F = G$*

A variable x is called *constrained* by a solved form c if c contains an atomic constraint of one of the forms Ax, xF or $x[f]y$. The set of constrained variables of c is written $\mathcal{C}(c)$. For instance, the following constraint is indeed a solved form, and its constrained variables are x and y.

$$Ax \wedge x[f]y \wedge x[g]z \wedge y\{h, k\} \wedge y[h]x$$

Solved forms are satisfiable. However, we can say more: Whatever values we choose for unconstrained variables there are always values for the constrained variables such that the constraint is satisfied. This is expressed by the following proposition:

Proposition 12 *For every solved form c we have that*

$$\mathcal{A}_{CFT} \models \forall \bar{x}\, \exists \bar{y}\, c$$

where $\bar{x} = \mathcal{FV}(c) - \mathcal{C}(c)$ and $\bar{y} = \mathcal{C}(c)$

Hence, in particular, every solved form is satisfiable. In continuation of the above example we get that the following formula is true in $\mathcal{A}_{CFT}$:

$$\forall z\, \exists x, y \Big(Ax \wedge x[f]y \wedge x[g]z \wedge y\{h, k\} \wedge y[h]x \Big)$$

For technical reasons it will be simpler to show Proposition 12 for a slightly stronger version of c. The justification why we may replace c by a stronger constraint d is that $\mathcal{A} \models \forall \bar{x}\, \exists \bar{y}\, d$ and $\mathcal{A} \models d \rightarrow c$ together imply that $\mathcal{A} \models \forall \bar{x}\, \exists \bar{y}\, c$.

We replace c by the following stronger constraint: If c does not contain a label constraint for some constrained variable x then we add to c a label constraint Ax for some arbitrary label A. Also, if c contains an arity constraint xF but no feature constraint $x[f]y$ for some $f \in F$ then we add a constraint $x[f]x_f$ where x_f is a new variable.

Beware that a solved form may contain cycles between constrained variables, as between x and y in the above example. Before giving the proof of Proposition 12 for the general case taking into account the possible existence of such cycles let us first look at the special case where there is no cycle. More formally, we write $x \sqsubset_c y$ when $y[f]x \in c$ for some feature f, and we assume for the moment that $\sqsubset_c^+$ is a strict ordering. For this special case we can show Proposition 12 by induction on the size of $\bar{y}$. If the size is 0 then $\mathcal{C}(c) = \emptyset$, and c is the empty conjunction which is valid. Otherwise, if $\bar{y}$ contains at least one variable, let z be a maximal variable in $\bar{y}$ w.r.t. $\sqsubset_c$ (which must exist since $\sqsubset_c$ is a strict order and $\bar{y}$ nonempty), and $\bar{z} = \bar{y} - \{z\}$. Since c is a solved form and z maximal we can write $\exists\bar{y}\, c$ equivalently as

$$\exists z \Big(z[f_1]v_1 \wedge \ldots \wedge z[f_n]v_n \wedge A_z z \wedge zF_z \wedge \exists\bar{z}\, c' \Big)$$

where c' is c without the atomic constraints containing the variable z. The variables v_i may be in $\bar{z}$ or in $\bar{x}$, and the constraint zF_z may be missing. Let $\bar{x} = (x_1, \ldots, x_k)$ and $\bar{z} = (x_{k+1}, \ldots, x_l)$, and let feature trees $\tau_1, \ldots, \tau_k$ be given. By induction hypothesis there exist feature trees $\tau_{k+1}, \ldots, \tau_l$ such that

$$\mathcal{A}_{CFT}, [x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k, x_{k+1} \mapsto \tau_{k+1}, \ldots, x_l \mapsto \tau_l] \models c'$$

We define the feature tree τ as

$$\begin{aligned} D_\tau &= \{\epsilon\} \cup \bigcup_{i=1}^{i=n} \{f_i \cdot \pi_i \mid \pi_i \in D_{\tau_i}\} \\ L_\tau(\epsilon) &= A_z \\ L_\tau(f_i \cdot \pi_i) &= L_{\tau_i}\pi_i \end{aligned}$$

where A is some arbitrary label, and if there is no label constraint for z then A_z is some arbitrary label. We now show that for

$$\alpha = [x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k, x_{k+1} \mapsto \tau_{k+1}, \ldots, x_l \mapsto \tau_l, z \mapsto \tau]$$

we have that $\mathcal{A}_{CFT}, \alpha \models c$, that is that $\mathcal{A}_{CFT}, \alpha \models b$ for every atomic constraint in c. If $z \notin \mathcal{FV}(b)$ then this follows from the induction hypothesis. If $b = Az$ then it follows from the construction and clause (2) of Definition 5, if $b = z[f]v$ then it follows from construction and clause (1) of Definition 5, and if $b = zF$ then it follows from construction and the last two clauses of Definition 5.

Proof: Given feature trees as values of the non-constrained variables, we can see a solved form as a function mapping a tuple of feature trees (indexed by the constrained variables) into a tuple of feature trees of the same kind. We get a solution of the solved form as the minimal fixed point of this function (we can assume w.l.o.g. that there is a label constraint for every constrained variable). This fixed point is obtained by starting with a tuple of trees containing just a root node plus the subtrees corresponding to the values of the unconstrained variables, and taking the least upper bound of the iterative application of the function. □

In the following we describe a *rule-based* algorithm for deciding satisfiability of CFT constraints.

An equation $x = y$ is *eliminated* in a constraint ϕ if $x = y$ is an atomic constraint of ϕ, y is a variable different from x, and x occurs only once in ϕ (hence this occurrence is the one in the equation $x = y$). In this case we also say that x is *eliminated* in ϕ.

The rules are given in Figure 5.1. We write $c \rightarrow_{CFT} d$ if the constraint c reduces in one step by the system of Figure 5.1 to the constraint d.

Here is an example of a reduction to solved form. The redeces of the rule applications are underlined.

$$\underline{x = y} \wedge x[f]x' \wedge y[f]y' \wedge Ax' \wedge By'$$
$$x = y \wedge \underline{y[f]x'} \wedge \underline{y[f]y'} \wedge Ax' \wedge By'$$

(Eq)	$\dfrac{x = y \wedge \phi}{x = y \wedge \phi\{x \mapsto y\}}$	x and y occur in ϕ, $x \neq y$
(Triv)	$\dfrac{x = x \wedge \phi}{\phi}$	
(SClash)	$\dfrac{Ax \wedge Bx \wedge \phi}{\perp}$	$A \neq B$
(Fun)	$\dfrac{x[f]y \wedge x[f]z \wedge \phi}{x[f]z \wedge y = z \wedge \phi}$	
(AClash)	$\dfrac{xF \wedge xG \wedge \phi}{\perp}$	$F \neq G$
(FClash)	$\dfrac{xF \wedge x[f]y \wedge \phi}{\perp}$	$f \notin F$

Figure 5.1: Simplification rules for CFT.

$$\begin{array}{l} x = y \wedge \underline{x' = y'} \wedge y[f]y' \wedge Ax' \wedge By' \\ x = y \wedge x' = y' \wedge y[f]y' \wedge \underline{Ay'} \wedge \underline{By'} \\ \perp \end{array}$$

Proposition 13 *If $c \rightarrow^*_{CFT} d$ then $CFT \models c \leftrightarrow d$.*

Proof: This is immediate by definition of the semantics of CFT. □

Proposition 14 *The system of Figure 5.1 is terminating.*

Proof: We show this using a lexicographic combination of measuring function ψ_1, ψ_2, $\ldots$, each mapping a constraint to some natural number. To show termination of the system it is sufficient to show that if c reduces to d then there is some i such that ψ_1 to ψ_{i-1} are unchanged and ψ_i strictly decreases.

More precisely, we define three measuring functions:

- $\psi_1(\phi)$ is the number of subtree constraints of ϕ,
- $\psi_2(\phi)$ is the number of atomic constraints of ϕ,
- $\psi_3(\phi)$ is the number of variables of ϕ that are *not* eliminated in ϕ.

The termination argument is now given by the following table:

Rule	ψ_1	ψ_2	ψ_3
Eq	$=$	$=$	$<$
Triv	$=$	$<$	
Fun	$<$		
{S,A,F}Clash	$\leq$	$<$	

□

Proposition 15 *Any constraint which is not reducible by $\rightarrow_{CFT}$ is either $\perp$ or of the form $e \wedge c$ where c is a solved form and e is a system of equations that are eliminated in $e \wedge c$.*

Theorem 16 ([ST94]) *Satisfiability of CFT-constraints is decidable in quasi-linear time.*

Proof: Given a constraint c, we apply the transformation rules of Figure 5.1 in any order until we obtain a constraint d that is no longer reducible. This process terminates by Proposition 14. By Proposition 15, d is either $\perp$ and hence not satisfiable, or of the form $e \wedge c$, where c is a solved form and e is eliminated, and hence satisfiable. By Proposition 13, c is satisfiable iff d is.

It is quite easy to see that the number of possible reductions is linear in the number of atomic constraints of c, hence a normal form of c can be found in polynomial time. An algorithm that exhibits quasi-linear complexity can be found in [ST94]. □

Note that this complexity result is about the so-called off-line complexity, that is it applies to the situation where the input is the complete constraint. In case of the Constraint Logic Programming, for instance, the constraint has to be checked for satisfiability at each stage of the computation. Hence, when we build a constraint consisting of n atomic constraint by successively adding a constraint and checking the conjunction for satisfiability then the total cost sums up to $O(n^2)$. Programming calculi are often designed in such a way that the constraint is stored in a logically equivalent *simplified* form. Then it makes sense to ask for the on-line complexity of the problem: Given a sequence of atomic constraints, $a_1, \ldots, a_n$, what is the complexity of determining the smallest i such that $a_1, \ldots, a_i$ is unsatisfiable? An algorithm solving this problem can store the constraints seen so far in a simplified form. The abstract machine presented in [ST94], based on this principle, has a quasi-linear complexity even in this on-line scenario.

Exercice 2
Consider the extension of the structure $\mathcal{A}_{CFT}$ by a new predicate symbol $\gg$. The interpretation of $\gg$ is that $x \gg y$ means that y is a son of x, that is $\tau_1 \gg \tau_2$ holds iff $\tau_2 = \tau_1[f]$ for some $f \in \mathcal{F}$.

1. Extend the constraint simplification system given for CFT to accommodate for the $\gg$ predicate. Don't forget that $\mathcal{F}$ is supposed to be *infinite*.

2. Show that the normal forms obtained by our obtained system are satisfiable if different from $\perp$.

3. Show that your new system is terminating

Exercice 3
Modify the constraint simplification system for CFT such that it checks for satisfiability in the restricted structure containing only *finite* feature trees.

Exercice 4
Consider the extension of the structure $\mathcal{A}_{CFT}$ by a new predicate symbol $\geq$. The interpretation of $\geq$ is that $x \geq y$ means that y is a descendent of x, that is $\tau_1 \geq \tau_2$ holds iff $\tau_2 = \tau_1[\pi]$ for some $\pi \in \mathcal{F}^*$. The domain of the structure is restricted to finite feature trees only.

1. Extend the constraint simplification system given for CFT to accomodate for the $\geq$ predicate. Don't forget that $\mathcal{F}$ is supposed to be *infinite*.

2. Show that the normal forms obtained by yur obtained system are satisfiable if different from $\perp$.

3. Show that your new system is terminating.
4. What is the difficulty when trying to accomodate for the $\geq$ predicate in a structure containing infinite feature trees?

5.2.2 Negated CFT Constraints and Independence

We might now try to add negated constraints to the constraint system CFT, thus considering constraints like

$$Ax \wedge x[f]y \wedge \neg\exists z\Big(x[g]z \wedge Bz\Big) \wedge \neg\exists v\Big(x[g]v \wedge v\{f,g,h\}\Big)$$

Negative constraints provide additional expressivity. There is another important motivation, stemming from the entailment problem, for instance in Constraint Programming (see Section 2.2.2.

The treatment of conjunctions of positive and negative constraints is simplified by an important property of CFT: The independence of negative constraints (or short: independence).

A constraint system is called *independent* if for all constraints $\gamma, \gamma_1, \ldots, \gamma_n$ (remember that we require the set of constraints to be closed under existential quantification) the following holds:

$$\gamma \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_n \text{ is satisfiable} \qquad \Leftrightarrow \qquad \text{for all } i\text{: } \gamma \wedge \neg\gamma_i \text{ is satisfiable}$$

This is equivalent to say that

$$\gamma \models \gamma_1 \vee \ldots \vee \gamma_n \qquad \Leftrightarrow \qquad \text{there is an } i \text{ such that } \gamma \models \gamma_i$$

Hence, independence allows us to reduce the problem of checking that a conjunction of positive and negative constraints $\gamma \wedge \neg\gamma_1 \ldots \wedge \neg\gamma_n$ is satisfiable, to the problem of checking that for all indices i the entailment judgment $\gamma \models \gamma_i$ does *not* hold.

An example of a constraint system that is not independent is Presburger arithmetic: $x * y = 0 \models x = 0 \vee y = 0$, but $x * y = 0$ does not imply any of $x = 0$ or $y = 0$.

CFT enjoys the independence property. This is the case only because we required the set of label symbols and the set of feature symbols to be infinite. If the set of labels is finite, say $\{A_1, \ldots, A_n\}$, then independence does not hold since *true* $\models (A_1x \vee \ldots \vee A_nx)$ but *true* implies no particular A_ix. A similar counterexample can be found if the set of features is finite.

5.2.3 Entailment of CFT Constraints

Hence, in order to cope with positive and negative constraints we have to solve the so-called *entailment* problem: Given constraints γ and γ', does $\gamma \models \gamma'$ hold? In other words, is the set of data items described by constraint γ' contained in the set of data items described by constraint γ? Besides its use for testing satisfiability of positive and negative constraints, entailment is of paramount importance in the paradigm of constraint programming.

The first observation is that, in an entailment problem $\gamma \models \gamma'$, we can assume γ to contain no existential quantifier. This is justified by the following equivalences, where we can assume without loss of generality that the set of local variables of γ, $\bar{x}$, is disjoint with $\mathcal{FV}(\gamma')$ (otherwise we can rename the local variables of γ since CFT, as all constraint systems, is closed under renaming of bound variables):

$$\begin{array}{ll} & (\exists\bar{x}(a_1 \wedge \ldots \wedge a_n)) \models \gamma' \\ \Leftrightarrow & \forall\bar{x}((a_1 \wedge \ldots \wedge a_n) \Rightarrow \gamma') \text{ is valid} \\ \Leftrightarrow & (a_1 \wedge \ldots \wedge a_n) \models \gamma' \end{array}$$

The existential quantifier on the right-hand side of the entailment problem, however, is essential and can not be dropped: the entailment judgment $x\{f\} \models \exists y\, x[f]y$ holds, however $x\{f\} \models x[f]y$ does not hold.

We will first demonstrate the test of entailment of CFT constraints with some examples before giving the exact criterion (Theorem 17).

$$x[f]y \wedge Ay \wedge x[f]z \wedge Bz \models \exists v\, x[g]v$$

holds since the left-hand side is not satisfiable, the rules of Figure 5.1 reduce the left-hand side to $\perp$. Hence the entailment judgment is vacuously true. The judgment

$$x[f]y \wedge Ay \models \exists z(x[f]z \wedge Bz)$$

does *not* hold since the left-hand side is satisfiable, but both sides together are not. Hence, there is a valuation that makes the left-hand side true and the right-hand side false, that is, implication does not hold. The judgment

$$x[f]y \wedge Ay \wedge x[g]z \wedge Bz \models \exists v(x[f]v \wedge x[g]z)$$

holds since the information on the right-hand side is directly contained in the left-hand side after proper instantiation of the local variable v. The judgment

$$x\{f, g\} \wedge Ax \wedge x[f]z \wedge y\{f, g\} \wedge Ay \wedge y[f]z \models x = y$$

does *not* hold since the values of x and y might differ at the feature g. The judgment

$$x\{f, g\} \wedge Ax \wedge x[f]x \wedge x[g]y \wedge y\{f, g\} \wedge Ay \wedge y[f]x \wedge y[g]y \models x = y$$

does hold. This case needs a more involved reasoning than the preceding cases:

We say that a variable x is *determined* by a constraint if the constraint contains a label constraint Ax, an arity constraint $x\{f_1, \ldots, f_n\}$ and selection constraints $x[f_i]y_i$ for each $i = 1, \ldots, n$. In the above example, x and y are both determined. The left-hand side does not contain any other variable, the consequence is that the left-hand-side has *exactly one solution.*

Furthermore, if we identify in the left-hand side x with y then we don't get a contradiction with the rules of Figure 5.1. Hence, there is a common solution of both sides. Since the left-hand side allows only one solution, we conclude that *all* solutions of the left-hand side also satisfy the right-hand side, that is that entailment holds.

Furthermore, the judgment

$$x\{f, g\} \wedge Ax \wedge x[f]x \wedge x[g]z \wedge y\{f, g\} \wedge Ay \wedge y[f]x \wedge y[g]z \models x = y$$

also holds. This is a generalization of the preceding case since we now have a non-determined variable (that is, z) on the left-hand side. The argument is similar to the preceding case, except that now the left-hand side has for every value of z exactly one solution in the determined variables x and y, and that the left-hand side under the identification $x = y$ is satisfiable for any value of z.

The exact criterion for entailment of CFT constraints is as follows. A CFT constraint γ is called a *saturated graph* if

- ϕ is different from $\perp$,

- ϕ is in normal form with respect to the simplification rules for CFT,
- if $x\{f_1, \ldots, f_n\} \in \phi$ then there are variables $y_1, \ldots, y_n$ such that $x[f_i]y_i \in \phi$ for all $i \in \{1, \ldots, n\}$,
- and ϕ does not contain an equation.

We may always assume that our entailment judgement is of the form $\gamma \models \exists X\, \phi$ where γ is a saturated graph, ϕ is a conjunction of atomic contraints, and X is disjoint with $\mathcal{FV}(\gamma)$. We call a substitution θ an *X-oriented normalizer* of $\gamma \wedge \phi$ if θ is the equational part of some normal form of $\gamma \wedge \phi$ with respect to the simplification rules of CFT, and if $\theta(\mathcal{FV}(\phi) - X) \cap X = \emptyset$ (that is, θ does never map a global variable of ϕ to a local variable). We can easily adapt the simplification rules for CFT such that the equational part of the computed normal form is always an X-oriented normalizer, by adding the rule

$$\text{(Orient)} \quad \frac{x = y \wedge \phi}{y = x \wedge \phi} \quad x \notin X,\, y \in X$$

For a conjunction of atomic constraints ϕ we write $\bar{\phi}$ for the conjunction of the non-equational constraints in ϕ.

Theorem 17 ([ST94]) *Let γ be a saturated graph, ϕ a conjunction of atomic constraints, X a finite set of variables not occurring in γ. Then $\gamma \models \exists X\, \phi$ holds iff normalization of $\gamma \wedge \phi$ by the rules of CFT plus (Orient) does not produce $\perp$, and if the resulting X-oriented normalizer satisfies*

1. *$\mathcal{C}(\theta\bar{\phi} - \theta\gamma) \subseteq X$, and*
2. *if $x = y \in \theta$ and $x \notin X$ then x and y are determined in γ.*

Theorem 18 ([ST94]) *Entailment of CFT constraints can be checked in quasi-linear time.*

A much more detailed exposition with proofs and the description of an abstract machine for testing satisfiability and entailment of CFT constraints can be found in [ST94]. In particular, it is shown there how the calculus of Theorem 17 can be refined to a *relative simplication calculus*, where only the constraint ϕ is actually rewritten and γ is merely used as a context.

Chapter 6

Syntactic Methods: Coping with Non-Determinism

6.1 Non-deterministic Syntactic Methods

In the previous chapter we have seen first examples of a method for solving constraints by syntactic simplification. The examples presented in the previous chapter use a simplification procedure which was *don't-care* non-deterministic: at each simplification step, there was possibly a choice of a redex and possibly a choice of the rule to apply. This choice, however, did not matter, and it was sufficient to pursuit only one simplification sequence until reaching a normal form. This could be turned into a deterministic algorithm by imposing an appropriate strategy.

In the current chapter we will extend this method by using a *don't-know* choice of the rules to apply. This applies in particular to constraint structures where constraint solving necessitates a complete exploitation of disjunctions, and typically leads to an algorithm which is at least in the complexity class NP.

A non-deterministic algorithm constructs some simplification sequence starting from the initial constraint until it reaches a normal form that is satisfiable. If such a sequence can be constructed then the constraint is satisfiable, otherwise it is unsatisfiable.

6.2 First-Class Features

Theorem 19 ([Tre93]) *Satisfiability of F-constraints is NP-complete.*

Proof: It is obvious that satisfiability of constraints can be checked in NP: If there is an atomic constraint $x[v]y$ and an arity constraint xF then we guess the value of v among the features in F; if there is no arity constraint for x then it is safe to take for v any new feature symbol.

To show that the problem is indeed NP-hard we encode 3-SAT in the satisfiability problem of F-constraints. Let $F = L_1 \wedge \ldots \wedge L_k$ be a propositional formula in conjunctive normal form, that is every L_i is a disjunction of three literals. We assume that the propositional variables are taken from the set $\{x_1, \ldots, x_n\}$. We construct an F-constraint ϕ_F such that F is satisfiable if and only if the constraints ϕ_F is satisfiable. The construction of F uses the variables $x_1^+, \ldots, x_n^+, x_1^-, \ldots, and, x_n^-, y_1, \ldots, y_k$. The formula ϕ_F is the conjunction of three subformulas $\phi_F^1 \wedge \phi_F^2 \wedge \phi_F^3$ that we will now describe:

1. For every i, either x_i^+ has root label $True$ and x_i^- has root label $False$, or the other way round.
$$\phi_F^1 = \bigwedge_{i=1}^{n} \exists z\Big(\begin{array}{l} z\{p,n\} \wedge z[p]x_i^+ \wedge z[n]x_i^- \wedge \\ \exists i,v(z[i]v \wedge Truev) \wedge \exists i,v(z[i]v \wedge Falsev)\Big) \end{array}$$

2. Encoding of the clause L_i:
$$\phi_F^2 = \bigwedge_{i=1}^{k} \Big(y_i\{1,2,3\} \wedge y_i[1]q_1 \wedge y_i[2]q_2 \wedge y_i[3]q_3\Big)$$
where L_i is $p_1 \vee p_2 \vee p_3$, and
$$q_j = \begin{cases} x_i^+ & \text{if } p_j = x_i \\ x_i^- & \text{if } p_j = \neg x_i \end{cases}$$
To give an example of this construction: If
$$F = \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3)}_{L_1} \wedge \underbrace{(\neg x_2 \vee x_4 \vee \neg x_5)}_{L_2}$$
then we get
$$\begin{array}{rcl} \phi_F^2 & = & y_1\{1,2,3\} \wedge y_1[1]x_1^+ \wedge y_1[2]x_2^- \wedge y_1[3]y_3^- \wedge \\ & & y_2\{1,2,3\} \wedge y_2[1]x_2^- \wedge y_2[2]x_4^+ \wedge y_2[3]x_5^- \end{array}$$

3. All clauses L_i evaluate to true:
$$\phi_F^3 = \bigwedge_{i=1}^{k} \exists j,v(y_i[j]v \wedge Truev)$$
□

6.3 Orderings on Terms

Theorem 20 ([NR99]) *Satisfiability of constraints in the structure $\mathcal{A}_{\mathrm{lpo}}$ is decidable in NPTIME for every signature Σ with a total precedence relation $<$.*

Proof: We present the proof for the special case of a signature Σ such that $0 \in \Sigma_0$, $f \in \Sigma_1$, $f > 0$, and $g > f$ for all $g \in \Sigma \setminus \{0, f\}$, and for constraints not using the symbols $0, f$. The proof for the general case can be found in [NR99].

Consider the non-deterministic simplification procedure given by the rules in Figure 6.1. Note that for the simplification of a constraint $s < t$ where both s and t are not variables the rule (A1) covers the case $top(s) > top(t)$, the rules (B1), (B2), and (B3) cover the case $top(s) = top(t)$, and the rules (C1), (C2), and (C3) cover the case $top(s) < top(t)$. Finally, the rules (E1) and (E2) simplify equations.

First note that the rules of Figure 6.1 terminate. This can be shown using as order the lexicographic combination of the following two orders:

1. the number of variables in the constraint (with the natural order $<$),
2. the multiset of the sizes of atomic constraints, ordered by the multiset extension of the natural order $<$.

$$\text{(A1)} \quad \frac{s > g(t_1, \ldots, t_n)}{s > t_1 \wedge \ldots s > t_n} \quad top(s) > g$$

$$\text{(B1)} \quad \frac{g(s_1, \ldots, s_n) > g(t_1, \ldots, t_n)}{s_1 = t_1 \wedge \ldots \wedge s_{i-1} = t_{i-1} \wedge s_i > t_i \wedge g(s_1, \ldots, s_n) > t_{i+1} \wedge \ldots \wedge g(s_1, \ldots, s_n) > t_n}$$

$$\text{(B2)} \quad \frac{g(s_1, \ldots, s_n) > t}{s_i > t} \quad top(t) = f \qquad \text{(B3)} \quad \frac{g(s_1, \ldots, s_n) > t}{s_i = t} \quad top(t) = g$$

$$\text{(C1)} \quad \frac{a > t}{\bot} \quad a < top(t)$$

$$\text{(C2)} \quad \frac{g(s_1, \ldots, s_n) > t}{s_i > t} \quad g < top(t) \qquad \text{(C3)} \quad \frac{g(s_1, \ldots, s_n) > t}{s_i = t} \quad g < top(t)$$

$$\text{(E1)} \quad \frac{\phi \wedge s = t}{\bot} \quad mgu(s,t) = \bot \qquad \text{(E2)} \quad \frac{\phi \wedge s = t}{\phi\sigma} \quad mgu(s,t) = \sigma$$

Figure 6.1: Non-deterministic simplification of lpo constraints.

Note that the rule (E2) either decreases strictly the number of variables (in case the equation $s = t$ contains at least one variable) or leaves the number of variables constant and decreases the multiset of sizes of the atomic constraints (in case $s = t$ is ground, since then $\sigma = id$). All other rules do not increase the number of variables and strictly decrease the multiset of sizes of atomic constraints.

Hence, if we build the graph of all constraints reachable from the initial constraint, where edges indicate one-step reachability, we obtain a non-cyclic finite graph. It is easy to see that every node is equivalent in $\mathcal{A}_{\text{lpo}}$ to the disjunction of its successors. Hence, the initial constraint is equivalent to the disjunction of all terminal nodes, that is the initial constraint is satisfiable in $\mathcal{A}_{\text{lpo}}$ iff one of the normal forms is. Note that every normal form different from $\bot$ is a conjunction of inequalities $s < t$ where at least one of s, t is a variable.

We call a *solved form* a constraint ϕ which

- is in normal form
- whenever $s > x \wedge x > t \subseteq \phi$ where s is not a variable then ϕ contains some normal form of $s > t$

The set of solved forms of a constraint ϕ is finite since we can repeatedly apply the rule

$$\text{(Trans)} \quad \frac{\phi \wedge s > x \wedge x > t}{\phi \wedge s > x \wedge x > t \wedge s > t} \quad s \text{ not a variable}$$

as long as the constraint is not in solved form. This process terminates since normalization of $s > t$ adds only inequations between subterms of the initial constraint. As a consequence, a constraint ϕ is satisfiable in $\mathcal{A}_{\text{lpo}}$ if one of its solved forms is.

One property of solved forms which we will use in the following is:

If ψ is a solved form and $x \in \mathcal{FV}(\psi)$ then $\psi' = \{s > t \in \psi \mid x \notin \mathcal{FV}(s,t)\}$ is again a solved form

First, ψ' is obviously in normal form. Now let $s > y, y > t \in \psi'$. Then $x \notin \mathcal{FV}(s,t)$, hence any normal form of $s > t$ does not contain x, and the normal form which is contained in ψ is also contained in ψ'. Hence, ψ' is a solved form.

Given a solved form ψ we define a pre-order $>_v$ on the variables occurring in ψ as the smallest transitive relation such that $x >_v y$ iff there exists a term t such that $x > t \in \psi$ and $y \in \mathcal{FV}(t)$. A solved form ψ is called *cycle-free* iff $x \not>_v x$ for all variables $x \in \mathcal{FV}(\psi)$. Obviously any normal form which is not cycle-free is unsatisfiable. We will now show that a cycle-free solved form is satisfiable.

Let ψ be a cycle-free solved form and $>_v$ be the associated ordering on $\mathcal{FV}(\psi)$. We define a substitution σ by induction on $>_v$ and then show that it is indeed a solution to ψ. Assume that σ is already defined for all variables v with $x >_v y$.

- $x\sigma = 0$ if there is no inequation $x > t$ in ψ
- $x\sigma = f(t)$ if $t = \max\{s\sigma \mid x > s \in \psi\}$ (note that, by the definition of $<_v$ and cycle-freeness of ψ all these $s\sigma$ are ground)

We will now show by induction on the number of variables that σ is a solution to ψ. If $\mathcal{FV}(\psi) = \emptyset$ then ψ is the trivially true constraint. Otherwise let $x \in \mathcal{FV}(\psi)$ be a variable which is maximal w.r.t. $>_v$, and let $\psi' = \{s > t \mid s > t \in \psi, x \notin \mathcal{FV}(s,t)\}$ which is again a cycle-free solved form as we have seen above. By induction hypothesis, σ is a solution to ψ'. It remains to show that $\sigma \models s > t$ for all $s > t \in \psi$ with $x \in \mathcal{FV}(s,t)$. There are three main cases:

1. $x \in \mathcal{FV}(s)$ and $x \in \mathcal{FV}(t)$. Then there are two cases for the form of the constraint:
   (a) $x > t[x]$: impossible since ψ cycle-free
   (b) $t[x]_{\neq\epsilon} > x$: by monotonicity of $\prec_{\text{lpo}}$ we have immediately that $t\sigma \succ_{\text{lpo}} x\sigma$
2. $x \notin \mathcal{FV}(s)$ and $x \in \mathcal{FV}(t)$. Then there are two cases for the form of the constraint:
   (a) $y > t[x]$: impossible since x is maximal w.r.t. $>_v$
   (b) $s > x$: Then $x \notin \mathcal{FV}(s)$, and by maximality of x the term s is not a variable.
   If there is no inequation $x > t \in \psi$ then $x\sigma = 0$ and $s\sigma > x\sigma$ obviously holds.
   Otherwise, $x\sigma = f(u)$ where $u = \max\{t\sigma \mid x > t \in \psi\}$. Let $x > t \in \psi$. Since ψ is solved it contains a normal form of $s > t$. This normal form does not contain x and hence does hold in σ. Hence, $s\sigma \succ_{\text{lpo}} u$, and since the top symbol of s is a function symbol greater than f, $s\sigma \succ_{\text{lpo}} f(u) = x\sigma$.
3. $x \in \mathcal{FV}(s)$ and $x \notin \mathcal{FV}(t)$. Then there are two cases for the form of the constraint:
   (a) $x > t$: Since $x \notin \mathcal{FV}(t)$ we obtain by definition of σ that $x\sigma \geq f(t\sigma) \succ_{\text{lpo}} t\sigma$.
   (b) $s[x]_{\neq\epsilon} > y$: we have that $x \neq y$. Assume that there is such an inequation in ψ which does not hold in σ, that is $s\sigma \not\succ_{\text{lpo}} y\sigma$. Let $s > y$ be such an inequation with a minimal y. Since the top symbol of s is a function symbol different from 0 this implies that $y\sigma \neq 0$, hence $y\sigma = f(u)$ where $u = \max\{t\sigma \mid y > t \in \psi\}$. Let t_u be the term such that $u = t_u\sigma$. We will show that $s\sigma \succ_{\text{lpo}} u$, which implies that $s\sigma \succ_{\text{lpo}} f(u) = y\sigma$ since the head symbol of s is greater than f, which yields a contradiction.
   Since ψ is a solved form it contains a normal form of $s > t_u$. For each of the inequalities in this normal form there are three possibilities:

i. it does not contain x: then it holds in σ, and hence $s\sigma \succ_{\text{lpo}} t\sigma$.

ii. it falls under one of the five preceding cases, hence it holds and as a consequence $s\sigma \succ_{\text{lpo}} t\sigma$

iii. it is of the form $s'[x] > z$, that is $y >_v z$. In this case it holds by maximality of y

Hence, in order to check satisfiability of a constraint we just have to check whether one of its solved forms is cycle-free. □

Decidability of *lpo* constraints in case of a total precedence was first shown by [Com90a, Com90b]. The first algorithm in NP was given by [Nie93].

Corollary 21 ([NR99]) *For every signature Σ and total precedence $<$ it is decidable in NPTIME whether a given constraint is satisfiable in $\mathcal{A}_{\text{lpo}}$ for some extension of $(\Sigma, <)$.*

Proof: If the procedure presented in the proof of Theorem 20 detects unsatisfiability then the constraint is obviously unsatisfiable in any extension of $(\Sigma, <)$. Otherwise the constraint is satisfiable for the extension of Σ by $f, 0$ with $g > f > 0$ for all $g \in \Sigma$. □

Corollary 22 *The Σ_1-fragment of $\mathcal{A}_{\text{lpo}}$ is decidable for any signature Σ and total precedence $<$.*

Proof: We can transform a given constraint in negation normal form and then eliminate any negative literal by the following replacements:

$$(\neq) \quad \frac{s \neq t}{s \prec_{\text{lpo}} t \vee t \prec_{\text{lpo}} s} \qquad\qquad (\not\prec_{\text{lpo}}) \quad \frac{s \neg \prec_{\text{lpo}} t}{s = t \vee t \prec_{\text{lpo}} s}$$

We transform the resulting formula into disjunctive normal form and apply Theorem 20 on each of its clauses. □

Theorem 23 ([CT94]) *Satisfiability of a single atomic lpo constraint in $\mathcal{A}_{\text{lpo}}$ is NP-hard.*

Proof: We encode 3SAT. $\mathcal{F} = \{f, g, h, 0\}$ with the precedence $g > h > f > 0$ and we assume g unary, h, f binary and 0 constant. We will use also the abbreviations: $1 = f(0,0)$ and $2 = f(0, f(0,0))$. Then, we use the following translations:

- each positive literal P is translated into $h(2, x_P) > f(h(x_P, x_P), h(2,0))$ which holds iff x_P is assigned to 1.
- each negative literal $\neg P$ is translated into $1 > x_P$ which holds iff x_P is assigned to 0.
- each clause $s_1 > t_1 \ \vee \ s_2 > t_2 \ \vee \ s_3 > t_3$ is equivalent (w.r.t. the $\geq_{lpo}$ interpretation) to

  $$f(g(C_1(C(0))), f(g(C_2(C(0))), g(C_3(C(0))))) > h(0, g(C(C(0))))$$

  where

  $$\begin{aligned} C(x) &= f(t_1, f(t_2, f(t_3, x))) \\ C_1(x) &= f(s_1, f(t_2, f(t_3, x))) \\ C_2(x) &= f(t_1, f(s_2, f(t_3, x))) \\ C_3(x) &= f(t_1, f(t_2, f(s_3, x))) \end{aligned}$$

- the conjunction $s_1 > t_1 \wedge \ldots \wedge s_n > t_n$ is equivalent to the single inequation

$$\begin{aligned} &C_h(s_1, \ldots, s_n, t_1, \ldots, t_n) \\ &\quad > C_f(C_h(t_1, s_2, \ldots, s_n, t_1, \ldots, t_n), \ldots, C_h(s_1, \ldots, s_{n-1}, t_n, t_1, \ldots, t_n)) \end{aligned}$$

  where C_h and C_f are the right "combs" recursively defined by: $C_\alpha(t, L) = \alpha(t, C_\alpha(L))$ and $C_\alpha(\emptyset) = 0$.

The coding is in $O(n^2)$. It is a routine verification that the resulting inequation is satisfiable iff the set of clauses is satisfiable. □

The decidability result of [NR99] applies in fact to the *recursive path ordering with status (rpos)*, of which the lexicographic path ordering is only a special case. Decidability of *lpo* constraints has first been shown by [Com90a, Com90b], and decidability of *rpo* constraints (another special case of rpos) has first been shown by [JO91]. Decidability of so-called *embedding* constraints, which are lpo constraints over an *empty* precedence relation (or equivalently, rpo constraints over an *empty* precedence relation) has been shown by [BC93]. The general case of decidability of these ordering constraints over any *partial* precedence is still open.

A similar result holds for KBO constraints:

Theorem 24 ([KV01]) *Satisfiability of constraints in the structure $\mathcal{A}_{\mathrm{kbo}}$ is NP-complete.*

6.4 Concluding Remarks on Non-Deterministic Methods

Does NP-completeness of the satisfiability problem imply that the constraint system is useless? There are two answers to this question:

1. It depends on the application. In applications like automatic theorem proving even NP might be an acceptable complexity since the search space is so huge that any means to restrict search by constraints is welcome. In programming, however, NP-complete constraint solving is much harder to accept. One can argue that the complexity of a program should stem from the algorithm, as expressed in the program, and not from the basic mechanisms that handles the data structures. Otherwise, it will be very difficult to obtain a useful complexity model. In other words, one should try to keep built-in constraint solving in programming simple (say linear, or at least polynomial of small degree), and have means to have more complex constraint solving programmed by the user.

2. The constraints might be hard to solve in the general case, however, they are easy to resolve once we have sufficient information about the values of the variables. In the case of F: Solving a constraint where all the feature variables are instantiated is nothing but CFT constraint solving, and this we know to do very efficiently. So one could attempt to modify our constraint-based formalism and allow constraints not to be solved immediately, that is to allow incomplete constraint solving methods.

Chapter 7

Syntactic Methods: Quantifier Elimination

7.1 Quantifier Elimination: Basic Concepts

Quantifier elimination is a classical concept in Model Theory, the subfield of Logic concerned about the expressivity of logical theories (see, for instance, [Hod93a]). The definition is often given for logical theories, here we adopt the classical definition to the case of a single structure.

Definition 6 *A structure $\mathcal{A}$ is said to have* quantifier elimination *if for any formula ϕ of the form $\exists\bar{x}\,(a_1 \wedge \ldots \wedge a_n)$, where the a_i are literals, there is a quantifier-free formula ψ such that*

1. $\mathcal{A} \models \phi \leftrightarrow \psi$
2. $\mathcal{FV}(\psi) \subseteq \mathcal{FV}(\phi)$

Furthermore, the structure has effective quantifier elimination *iff there is computable function yielding ψ from ϕ.*

One can say that a structure has quantifier elimination if quantifiers contribute nothing to the first-order definable relations in the structure. This intuition is justified by the following easy lemma:

Lemma 25 *Let $\mathcal{A}$ be a structure that has quantifier elimination. Then for every formula ϕ there exists a quantifier-free formula ψ with $\mathcal{FV}(\psi) \subseteq \mathcal{FV}(\phi)$ such that $\mathcal{A} \models \phi \leftrightarrow \psi$.*

Proof: The proof is straightforward by the number of quantifiers occurring in ϕ.

If the number of quantifiers in ϕ is not zero then ϕ contains a sub-formula $\exists x\,\phi'$ or $\forall x\,\phi'$ where ϕ' is quantifier-free, that is ϕ can be written as $\phi[\exists x\,\phi']$ or as $\phi[\forall x\,\phi']$.

In the first case we transform ϕ' into disjunctive normal form $c_1 \vee \ldots \vee c_n$. Since the structure has quantifier elimination every formula $\exists x\, c_i$ is equivalent to some quantifier-free formula ψ_i. Hence

$$
\begin{aligned}
\phi[\exists x\,\phi'] &\Leftrightarrow \phi[\exists x\,(c_1 \vee \ldots \vee c_n)] \\
&\Leftrightarrow \phi[(\exists x\, c_1) \vee \ldots \vee (\exists x\, c_n)] \\
&\Leftrightarrow \phi[\psi_1 \vee \ldots \vee \psi_n]
\end{aligned}
$$

which has one quantifier less.

In the second case we transform $\phi[\forall x\, \phi']$ into $\phi[\neg\exists x\, \neg\phi']$ and proceed as above. $\square$

The importance of quantifier elimination for deciding validity of a sentence in some structure is

Lemma 26 *Let $\mathcal{A}$ be a structure that has effective quantifier elimination and where the validity of any atomic sentence is decidable. Then the validity problem of first-order sentences in $\mathcal{A}$ is decidable.*

A first observation is that in proofs of (effective) quantifier elimination we can always restrict our attention to formulas of the form $\exists x\, (a_1 \wedge \ldots \wedge a_n)$ where $x \in \mathcal{FV}(a_i)$ for each i, since $\exists x\, (a \wedge \phi)$ is equivalent to $a \wedge \exists x\, \phi$ if $x \notin \mathcal{FV}(\phi)$.

7.2 Example: Successor Arithmetic

Lemma 27 *The structure $\mathcal{A}_S$ has effective quantifier elimination.*

Proof: Let $\exists x\, (a_1 \wedge \ldots \wedge a_n)$ a basic formula with $x \in \mathcal{FV}(a_i)$ for each i. We first eliminate all (in)equalities having the variable x on both sides:

$$\text{(S1)} \quad \frac{s^k(x) = s^k(x)}{\top} \qquad\qquad \text{(S2)} \quad \frac{s^k(x) = s^l(x)}{\bot} \quad k \neq l$$

$$\text{(S3)} \quad \frac{s^k(x) \neq s^k(x)}{\bot} \qquad\qquad \text{(S4)} \quad \frac{s^k(x) \neq s^l(x)}{\top} \quad k \neq l$$

If the remaining system contains an equality $s^k(x) = t$, where $x \notin \mathcal{FV}(t)$, then we can eliminate the variable x and just keep the assertion that t is not smaller than k:

$$\text{(SE)} \quad \frac{\exists x\, (s^k(x) = t \wedge s^{l_1}(x) \diamond_1 t_1 \wedge \ldots s^{l_n}(x) \diamond_n t_n)}{\bigwedge_{0 \leq i < k} t \neq s^i(0) \wedge s^{l_1}(t) \diamond_1 s^k(t_1) \wedge \ldots \wedge s^{l_n}(t) \diamond_n s^k(t_n)} \quad \begin{array}{l} \diamond_i \in \{=, \neq\} \\ x \notin \mathcal{FV}(t), \mathcal{FV}(t_i) \end{array}$$

If the system contains only inequalities then it is equivalent to $\top$. The reason is that for every valuation of the free variables of the formula there are at most n values for x that make the formula false.

$$\text{(SN)} \quad \frac{\exists x\, (s^{l_1}(x) \neq t_1 \wedge \ldots \wedge s^{l_n}(x) \neq t_n)}{\top} \quad x \notin \mathcal{FV}(t_i)$$

$\square$

Hence, and since validity of atomic sentences is obviously decidable, we obtain

Theorem 28 *The theory of Successor Arithmetic is decidable.*

7.2.1 Related Results

Theorem 29 ([Tar51]) *The theory of $\mathcal{A}_{\mathbb{R}}$ is decidable.*

This result has been found by Tarski in 1930 but was published only twenty years later [Tar51]. The quantifier elimination technique is much more involved than in the case of successor arithmetic, and now has to deal with roots of polynomials. A modern presentation can be found in [BCR98].

Exercice 5
Show that the theory of $\mathcal{A}_{\mathbb{C}}$, that is the structure of complex numbers (with the same language as $\mathcal{A}_{\mathbb{R}}$, enriched by the constant i) is decidable.

7.3 Quantifier Elimination in an Extended Language

Quantifier elimination is a very strong property of a structure. It is not surprising that many structures do not have this property. There are, however, ways to make quantifier elimination work in some of these cases.

Our first example is Presburger Arithmetic. The structure $\mathcal{A}_{Pr}$ does not have quantifier elimination since there is no quantifier-free formula that is equivalent in $\mathcal{A}_{Pr}$ to the formula $\exists y\, x = y + y$. We can fix this by introducing a new predicate $D_2(x)$ that is true iff the value of x is a multiple of 2. This predicate is of course definable in $\mathcal{A}_{Pr}$ and hence does not change the expressivity of $\mathcal{A}_{Pr}$. This is rather a technical trick to enlarge the set of quantifier-free formulas. Of course, when changing the language we have to check again the property of quantifier elimination for the extended language (i.e., for the new definition of quantifier-free formulas). We could push this to an extreme and extend any structure to a structure that has the same expressive power and that has effective quantifier elimination by introducing a predicate symbol for every first-order formula (which leads to an infinite language). In this extreme case, however, the procedure of quantifier elimination does not have any interest since no formula gets simplified. In this case we gain nothing for the application of Lemma 26 since the set of atomic sentences in the extended language is the same as the complete set of sentences. Hence, showing that validity of atomic sentences in the extended language is decidable is exactly the same problem as the one we started from.

Hence, a crucial step in devising a quantifier elimination procedure is to carefully extend the language such that quantifier elimination becomes possible while validity of atomic sentences is still decidable. In the case of Presburger Arithmetic it turns out that the extension by the family of predicates "x is a multiple of n" for any n is sufficient to obtain effective elimination of quantifiers, as we will show in Section 7.4. In some cases there is, however, another way to regain a weaker form of quantifier elimination, as we will see later in Section 7.6.

7.4 Example: Presburger Arithmetic

In this section we show effective quantifier elimination for an extension of Presburger Arithmetic. The procedure is due to Presburger [Pre29]. Our exposition follows [End72]. Another method for deciding the theory of Presburger Arithmetic which uses automata will be presented in Section 9.1.

Structure Extended Presburger Arithmetic $\mathcal{A}_{PrE}$

The language of Extended Presburger Arithmetic *consists of*

- *constants* $\ldots, -1, 0, 1, \ldots$*;*
- *two binary function symbols* $+$ *and* $-$*;*
- *a binary relation symbol* $<$*;*
- *a unary predicate symbol* D_n *for every natural number* n*.*

The structure $\mathcal{A}_{PrE}$ *has domain* $\mathbb{Z}$*, the set of integer numbers including zero. A constant symbol* c *is interpreted as the value* c*,* $+$ *as addition,* $-$ *as subtraction, and* $<$ *as the relation "strictly less then". The predicate* $D_n(x)$ *is true iff the value of* x *is a multiple of* n*.*

Note that, in contrast to $\mathcal{A}_{Pr}$, the structure $\mathcal{A}_{PrE}$ has $\mathbb{Z}$ as domain. In order to obtain (effective) quantifier elimination for $\mathcal{A}_{Pr}$ from a proof of (effective) quantifier elimination for $\mathcal{A}_{PrE}$ we have to account for two main differences between the two structures:

1. The structure $\mathcal{A}_{PrE}$ has domain $\mathbb{Z}$ while the structure $\mathcal{A}_{Pr}$ has domain $\mathbb{N}$.
2. The structure $\mathcal{A}_{PrE}$ has the symbol $-$ while the structure $\mathcal{A}_{Pr}$ does not have this symbol.

In order to cope with the restricted domain in $\mathcal{A}_{Pr}$ in case of a closed formula ϕ we just perform a *relativisation of quantifiers* by the formula $x > -1$, that is

- we replace every subformula of the form $\exists x\, \psi$ by $\exists x\, (x > -1 \wedge \psi)$
- we replace every subformula of the form $\forall x\, \psi$ by $\forall x\, (x > -1 \rightarrow \psi)$

Then we perform quantifier elimination of $\mathcal{A}_{PrE}$ on the obtained formula. If the formula ϕ is not closed and has the free variables $x_1, \ldots, x_n$ then we first replace ϕ by

$$x_1 > -1 \wedge \ldots \wedge x_n > -1 \wedge \phi$$

and then relativize the quantifiers, and do quantifier elimination as in $\mathcal{A}_{PrE}$.

Finally we can easily eliminate all occurrences of $-$, by replacing for instance $r > s - t$ by $r + t > s$, and $D_n(s-t)$ by $\bigvee_{i=0,\ldots,n-1}(D_n(s+i) \wedge D_n(t+i))$.

Lemma 30 *The structure* $\mathcal{A}_{PrE}$ *has effective quantifier elimination.*

Proof: In the following we write nx, where n is a non-negative integer, as abbreviation for $x + \ldots + x$ (n-times), and nx where n is a negative integer as abbreviation for $0 - x - \ldots - x$ ($|n|$ occurrences of x). We assume that all terms are written in the form $n_x x + \ldots n_y y + n_0$, where $n_0, x_x, \ldots, n_y$ are constants.

Our running example will be

$$\exists x\, (\neg(6x < 2x + w) \wedge -2x < -u \wedge \neg(v < 3x) \wedge D_3(x - t))$$

Elimination of Negative Literals The first step is to eliminate all negated predicates, using the following rules:

$$\text{(NotInf)} \qquad \frac{\neg t_1 < t_2}{t_2 < t_1 + 1}$$

$$\text{(NotDiv)} \qquad \frac{\neg D_i(t)}{D_i(t+1) \vee D_i(t+2) \vee \ldots \vee D_i(t+i-1)}$$

Please note that the rule (NotDiv) introduces a disjunction into the formula. We can easily get rid of the disjunction by transforming the matrix of the formula into disjunctive normal form and then distributing the existential quantifier over the disjunction. Introduction of disjunctions does not cause a problem for proving termination of the rule system since if we have an ordering $>$ such that, if a basic formula ϕ transforms by a rule and the above transformations into a disjunction of basic formulas $\phi_1 \vee \ldots \vee \phi_n$ and $\phi > \phi_i$ for each i, then we obtain for the multiset extension $>_{mul}$ of $>$ that $\{\phi\} >_{mul} \{\phi_1, \ldots, \phi_n\}$.

Example:

$$\exists x\ (2x + w < 6x + 1 \wedge -2x < -u \wedge 3x < v + 1 \wedge D_3(x - t))$$

Isolate x in Comparisons, and Eliminate Negative Multiplicities of x The next step is to ensure that every predicate has only one occurrence of the variable x, and that this occurrence has a non-negative multiplicity. Furthermore, we wish all comparison literals to be of the form $nx < t$ or $t < nx$.

$$\text{(InfL)} \qquad \frac{ix + s < jx + t}{(i-j)x < t - s} \quad i \geq j$$

$$\text{(InfR)} \qquad \frac{ix + s < jx + t}{s - t < (j-i)x} \quad j > i$$

$$\text{(NegDiv)} \qquad \frac{D_i(nx + t)}{D_i((-n)x - t)} \quad n < 0$$

Example:

$$\exists x\ (w < 4x + 1 \wedge u < 2x \wedge 3x < v + 1 \wedge D_3(x - t))$$

Note that these rules may make the variable x disappear completely from the literal, for instance $2x + y + 1 < 2x + 3z$ would be simplified to $y + 1 < 3z$. Such a literal not containing x would be moved outside of the scope of the $\exists x$ quantifier.

Same Multiplicity for all Occurrences of x We can assume without loss of generality that, after the preceding step, the variable x still occurs at least once in the formula (otherwise we could just drop the quantifier $\exists x$). Let p be the least common multiple of all multiplicities of the variable x. Then we replace

$$\text{(MInfR)} \qquad \frac{t < nx}{\left(\frac{p}{n}\right) t < px}$$

$$\text{(MDiv)} \qquad \frac{D_i(nx + t)}{D_{i*p/n}(px + \left(\frac{p}{n}\right) t)}$$

$$\text{(MInfL)} \qquad \frac{nx < t}{px < \left(\frac{p}{n}\right) t}$$

Example:

$$\exists x\,(3w < 12x + 3 \wedge 6u < 12x \wedge 12x < 4v + 4 \wedge D_{36}(12x - 12t))$$

Eliminate the Multiplicity of x We can now replace all occurrences of px, where p is the unique multiplicity of x, just by x under the additional assertion that x is a multiple of p.

$$\text{(Norm)}\quad \frac{\exists x\,\phi}{\exists x\,(\phi\{px \mapsto x\} \wedge D_p(x))}\quad \text{all occurrences of } x \text{ in } \psi \text{ have multiplicity } p.$$

Example:

$$\exists x\,\Big(3w < x \wedge 6u < x + 3 \wedge x < 4v + 4 \wedge D_{36}(x - 12t) \wedge D_{12}(x)\Big)$$

Eliminate the Quantifier The formula obtained so far is of the form

$$\exists x\,\left(\bigwedge_{0\le j<l} r_j < x \wedge \bigwedge_{0\le i<k} x < s_i \wedge \bigwedge_{0\le i<n} D_{m_i}(x + t_i)\right)$$

If $l = 0$ or $k = 0$, that is if there is no lower bound for x or if there is no upper bound for x, then let M be the smallest common multiple of $m_1, \ldots, m_n$. It is sufficient to check whether one value in the interval $[0 \ldots M - 1]$ satisfies all the divisibility predicates since in this case we can find a value satisfying the formula including the upper bounds by subtracting a suitable multiple of M:

$$\bigvee_{0\le q<M}\ \bigwedge_{0\le j<n} D_{m_j}(q + t_j)$$

If $l > 0$, $k > 0$ and $n = 0$, that is if there are no divisibility predicates, then there is a value for x satisfying the formula iff the formula is satisfied for the largest of the r_i plus one. Hence we can replace by the following formula:

$$\bigvee_{0\le i<l}\left(\bigwedge_{0\le j<l} r_j < r_i + 1 \wedge \bigwedge_{0\le j<k} r_i + 1 < s_j\right)$$

If $l > 0$ and $n > 0$ then let M be the smallest common multiple of $m_1, \ldots, m_n$. It is not sufficient to take for x the value of the largest of the r_i plus one since this may not satisfy the divisibility constraints. Rather we have to try all values from the largest r_i plus one to the largest r_i plus $M + 1$. Hence, we replace by the following formula:

$$\bigvee_{0\le i<l}\left(\bigwedge_{0\le j<l} r_j < r_i + 1 \wedge \bigvee_{0\le q<M}\left[\bigwedge_{0\le j<k} r_i + q + 1 < s_j \wedge \bigwedge_{0\le j<n} D_{m_j}(r_i + q + 1 + t_j)\right]\right)$$

Example:

$$\begin{array}{ll} & 6u < 3w + 1 \wedge \\ & \quad \bigvee_{i=0,\ldots,35} (3w + i + 1 < 4v + 4 \wedge D_{36}(i + 3w + 1 - 12t) \wedge D_{12}(i + 3w + 1)) \\ \vee & 3w < 6u + 1 \wedge \\ & \quad \bigvee_{i=0,\ldots,35} (6u + i + 1 < 4v + 4 \wedge D_{36}(i + 6u - 12t + 1) \wedge D_{12}(i + 6u + 1)) \end{array}$$

□

Hence, we obtain

Theorem 31 *The theories of $\mathcal{A}_{PrE}$ and of Presburger Arithmetic are decidable.*

A word of warning: Although this quantifier elimination yields theoretically an algorithm to decide validity of any Presburger formula this algorithm does not have a feasible complexity. There are two major sources of the complexity of the quantifier elimination procedure presented so far:

1. The rule (NotDiv) replaces a negative literal $\neg D_n(t)$ by $O(n)$ many positive literals $D_n(t')$. Note that the *value* of n is exponential in the *length of the binary representation* of n, that is exponential in the size of the formula.

   However, it was not really necessary to eliminate the negative literals $\neg D_n(t)$ since their validity is cyclic in n as it is the case for the positive literals $D_n(t)$. Hence, the reasoning of the succeeding rules would have been the same in presence of negative divisibility literals.

2. Another reason for the inefficiency of this algorithm lies in the fact that we always work with a formula in disjunctive normal form. Transforming a formula into disjunctive normal form causes in the worst case an exponential increase in size.

   Working with a disjunctive normal form seems to be crucial for our quantifier elimination procedure. However, we will see in the next section that we can in fact do without it.

Some related results on Presburger Arithmetic are given at the end of Section 7.5.

7.5 Quantifier Elimination without Transformation into Disjunctive Normal Form

A more efficient quantifier elimination for Presburger arithmetic which does not transform the matrix of a formula into disjunctive normal form before eliminating an innermost existential quantifier has been given by Cooper [Coo72] (see also [Opp73] which analyzes the complexity of the algorithm).

As in the previous section we work in the extended structure $\mathcal{A}_{PrE}$. In order to transform a formula $\exists x\, \psi$, where ψ is any quantifier-free formula of $\mathcal{A}_{PrE}$, into a quantifier-free formula we proceed as follows:

1. Using the same transformations as in the previous section we transform ψ into an equivalent formula ψ' such that the only literals containing the variable x are of one of the forms $x < t$, $t < x$, $D_n(x + t)$, $\neg D_n(x + t)$ with $x \notin \mathcal{FV}(t)$. Note that, in contrast to the previous section, we do not eliminate negations of divisibility predicates. Furthermore we keep literals not containing the variable x inside the scope of the quantifier $\exists x$.

2. Let $\psi'_{-\infty}(x)$ be the formula obtained from ψ' by replacing any literal of the form $x < t$ by *true*, and any literal of the form $t < x$ by *false*. Let M be the least common multiple of all indices of divisibility predicates occurring in ψ' (these may be positive or negative literals). We transform $\exists x\, \psi'$ into

$$\bigvee_{j=1}^{M} \psi'_{-\infty}(j) \vee \bigvee_{j=1}^{M} \bigvee_{t<x\in\psi'} \psi'(j + t)$$

The last transformation remains to justified.

Exercice 6
Let ψ be a quantifier-free formula of $\mathcal{A}_{PrE}$ such that the only literals containing the variable x are of one of the forms $x < t$, $t < x$, $D_n(x+t)$, $\neg D_n(x+t)$ where $x \notin \mathcal{FV}(t)$. Let M be the least common multiple of all indices of divisibility predicates occurring in ψ (positively or negatively). Show that

$$\mathcal{A}_{PrE} \models \exists x\, \psi \equiv \bigvee_{j=1}^{M} \psi_{-\infty}(j) \vee \bigvee_{j=1}^{M} \bigvee_{t<x\in\psi} \psi(j+t)$$

7.5.1 Related Results

Theorem 32 ([FR74]) *The non-deterministic time complexity of deciding validity of formulas in $\mathcal{A}_{Pr}$ has lower bound $2^{2^{cn}}$ for some constant c.*

Theorem 33 ([FR79]) *There is an algorithm for deciding validity of formulas in $\mathcal{A}_{Pr}$ running in space $O(2^{2^{cn}})$, and hence in deterministic time $O(2^{2^{2^{cn}}})$ for some constant c.*

A better complexity bound can be obtained when one limits the number of quantifier alternations of the formulas :

Theorem 34 ([RL78]) *There is a deterministic algorithm for deciding validity of a formula in $\mathcal{A}_{Pr}$ of size n and with m quantifier eliminations running in space $2^{d*n^{m+4}}$ and time $2^{2^{e*n^{m+4}}}$ for appropriate constants $d, e > 0$.*

7.6 Weak Quantifier Elimination

The theory FT does not allow quantifier elimination since there is no quantifier-free formula that is equivalent to $\exists y\,(x[f]y \wedge ay)$. This could be fixed as above by extending the structure with new predicates. The first decidability proof of the first-order theory of $\mathcal{A}_{FT}$ [BS95] employed in fact an extension by a family of so-called *path constraints*. There is, however, a simpler way: feature selection constraints work like partial functions. In the example we could hence trade the existential quantifier for a universal one, plus the assertion that the feature f is defined at x:

$$\neg xf\uparrow \wedge \forall y\,(x[f]y \rightarrow ay)$$

If we apply this transformation to a single existential quantifier then we don't gain anything in trading an $\exists$ for a $\forall$. The trick is now to apply the quantifier "trading" not to a single quantifier but to a block of quantifiers of the same kind. Starting with a formula in prenex normal from we can thus reduce the number of quantifier alternations in the quantifier prefix.

Definition 7 *A structure $\mathcal{A}$ is said to have* weak quantifier elimination *if for any formula ϕ of the form $\exists\bar{x}\,(a_1 \wedge \ldots \wedge a_n)$, where the a_i are literals, there is a formula $\psi = \forall\bar{y}\,\psi'$, where ψ' is quantifier-free, such that*

1. $\mathcal{FV}(\psi) \subseteq \mathcal{FV}(\phi)$
2. *if* $\mathcal{FV}(\phi) = \emptyset$ *then* $\bar{y} = \emptyset$

3. $\mathcal{A} \models \phi \leftrightarrow \psi$.

Furthermore, the structure has effective weak quantifier elimination *iff there is computable function which, given a formula* ϕ*, yields* ψ *as above.*

We will prove in Section 7.7 that the structure $\mathcal{A}_{FT}$ has effective weak quantifier elimination.

Lemma 35 *Let* $\mathcal{A}$ *be a structure that has weak quantifier elimination. Then for every formula* ϕ *there exists a formula* $\psi = \forall \bar{y}\, \psi'$ *with* $\mathcal{FV}(\psi) \subseteq \mathcal{FV}(\phi)$ *and* ψ' *quantifier-free such that* $\mathcal{A} \models \phi \leftrightarrow \psi$*. If* ϕ *is a sentence then* ψ *is a quantifier-free sentence.*

Proof: Let ϕ be a formula in prenex normal form. We proceed by induction on the number n of quantifier blocks in the quantifier prefix.

For $n = 0$ the claim is trivial.

Let $n \geq 1$ and $\phi = Q\exists\bar{x}\, \phi'$, where Q is a (possibly empty) string of quantifiers, not ending with $\exists$, and ϕ' is quantifier-free. We transform ϕ' into disjunctive normal form and obtain an equivalent formula

$$Q\exists\bar{x}\, (\phi_1 \vee \ldots \vee \phi_n)$$

where every ϕ_i is a conjunction of literals. This is equivalent to

$$Q(\exists\bar{x}\, \phi_1 \vee \ldots \vee \exists\bar{x}\, \phi_n)$$

We now apply weak quantifier elimination to every of the $\exists\bar{x}\, \phi_i$ and obtain

$$Q(\forall\bar{y}_1\, \psi_1 \vee \ldots \vee \forall\bar{y}_n\, \psi_n)$$

where every ψ_i is quantifier-free, and where all $\bar{y}_i$ are empty if Q is the empty string and $\mathcal{FV}(\phi) = \emptyset$. (by condition (3) of Definition 7). After possibly renaming bound variables, this can be transformed into the sentence $Q\forall\bar{z}\, \psi'$, where ψ' is quantifier-free. By condition (2) of Definition 7,

$$\mathcal{FV}(Q\forall\bar{z}\, \psi') \subseteq \mathcal{FV}(Q\exists\bar{x}\, \phi')$$

Since the number of quantifier alternations in $Q\forall\bar{x}\, \psi$ is $n - 1$, we can now apply the induction hypothesis.

If the innermost block of quantifiers consists of universal quantifiers, that is if $\phi = Q\exists\bar{x}\, \phi'$, then we consider the equivalent formula

$$\neg\tilde{Q}\forall\bar{x}\, \neg\phi'$$

(where $\tilde{Q}$ denotes the quantifier string obtained from Q by swapping existential and universal quantifiers) and apply the above transformation to it. We hence obtain

$$\neg\tilde{Q}\exists\bar{z}\, \neg\psi'$$

which is equivalent to

$$Q\forall\bar{z}\, \psi' \qquad \square$$

Lemma 36 *Let* $\mathcal{A}$ *be a structure that has effective weak quantifier elimination and where the validity of any atomic sentence is decidable. Then the validity problem of first-order sentences in* $\mathcal{A}$ *is decidable.*

7.7 Example: Feature Trees

The first quantifier elimination procedure for $\mathcal{A}_{FT}$ has been given by [BS95], working on a language extended by so-called *path constraints*. The weak quantifier elimination procedure presented here is derived from the more general result of [Tre97] which follows ideas of [Mal71] and [CL89].

We have eight possible literals:

$$x = y, x \neq y, Ax, \neg Ax, xf\uparrow, \neg xf\uparrow, x[f]y, \neg x[f]y$$

As usual we consider equality as symmetric, that is we identify an equation $x = y$ with $y = x$. Logical conjunction is considered associative and commutative, that is we identify two conjunctions of literals that differ only by the order of their literals.

In contrast to the previous procedures of quantifier elimination we model here explicitly the rule that allows to move some literals out of the scope of an existential quantifier (rule (Lift) below).

We call *basic formula* any formula of the form $b_1 \wedge \ldots \wedge b_m \wedge \exists \bar{x}\, (a_1 \wedge \ldots \wedge a_n)$ where the a_i and the b_i are literals.

Elimination of $\neg xf\uparrow$ and $\neg x[f]y$ We first show how to eliminate completely literals of the form $\neg xf\uparrow$ and $\neg x[f]y$.

(Undef1) $\dfrac{\neg xf\uparrow}{\exists y\, x[f]y}$ (Feat1) $\dfrac{\neg x[f]y}{xf\uparrow \vee \exists z\, (x[f]z \wedge z \neq y)}$

Hence, we can transform any basic formula of the form $\exists \bar{x}\, \phi$ into an equivalent basic formula not containing the literals $\neg xf\uparrow$ and $\neg x[f]y$.

Computing presolved forms We can now suppose that only the following types of literals are left:

$$x = y, x \neq y, Ax, \neg Ax, xf\uparrow, x[f]y$$

We now apply some normalization of label, subterms and undefinedness constraints. Note that the rule (Feat2) introduces an equality. The system (PS) consists of the following rules:

(Eq1) $\dfrac{x = x}{\top}$ (Label1) $\dfrac{Ax \wedge Bx}{\bot}$ $A \neq B$

(Eq2) $\dfrac{\exists \bar{x}, x\; x = y \wedge \phi}{\exists \bar{x}\, \phi\{x \mapsto y\}}$ $x \neq y$ (Label2) $\dfrac{Ax \wedge \neg Ax}{\bot}$

(DisEq) $\dfrac{x \neq x}{\bot}$ (Label3) $\dfrac{Ax \wedge \neg Bx}{Ax}$ $A \neq B$

(Lift) $\dfrac{\exists \bar{x}\, (\phi \wedge \psi)}{\phi \wedge \exists \bar{x}\, \psi}$ $\mathcal{FV}(\phi) \cap \bar{x} = \emptyset$ (Feat1) $\dfrac{x[f]y \wedge xf\uparrow}{\bot}$

(Feat2) $\dfrac{x[f]y \wedge x[f]z}{x[f]y \wedge z = y}$

Proposition 37 *The system (PS) is terminating.*

Proof: We show this using a lexicographic combination of three measuring functions m_1, m_2 and m_3, each mapping a basic formula to some natural number. To show termination of the system it is sufficient to show that if c reduces to d then there is some i such that m_1 to m_{i-1} are unchanged and m_i strictly decreases.

More precisely, we define two measuring functions:

- $m_1(\phi \wedge \exists \bar{x}\, \psi)$ is the number of subtree constraints in ϕ and ψ.
- $m_2(\phi \wedge \exists \bar{x}\, \psi)$ is the number of atomic constraints in ϕ and ψ.
- $m_3(\phi \wedge \exists \bar{x}\, \psi)$ is the number of atomic constraints in ψ.

The rules of (PS) decrease these measures as follows:

	Eq1	Eq2	DisEq	Lift	Label1	Label2	Label3	Feat1	Feat2
m_1	$=$	$=$	$\leq$	$=$	$\leq$	$\leq$	$=$	$<$	$<$
m_2	$<$	$<$	$<$	$=$	$<$	$<$	$<$		
m_3				$<$					

In fact, none of the rules increases a measure function. Hence, it would have been sufficient to just use the sum of the three measures. □

Proposition 38 *The rules of (PS) are equivalence transformations in $\mathcal{A}_{FT}$.*

Definition 8 *We say that a basic formula $\phi \wedge \exists \bar{x}\, \psi$ is FT-presolved if*

1. *it does not contain literals of the form $\neg x[f]y$, $\neg xf\uparrow$, or $x \neq x$*
2. *ψ does not contain any equation*
3. *if $x[f]y \wedge x[f]z \subseteq \psi$ then $y = z$,*
4. *if $x[f]y \in \psi$ then $xf\uparrow \notin \psi$,*
5. *if $Ax \in \psi$, then ψ contains no other positive or negative sort constraint for x*
6. *if $a_i \in \psi$ then $\mathcal{FV}(a_i) \cap \bar{x} \neq \emptyset$*

Lemma 39 *Let ϕ be a basic formula not containing formulas of the form $\neg xf\uparrow$ or $\neg x[f]y$. Any (PS)-normal form of ϕ is either $\perp$ or FT-presolved.*

Proof: This is immediate by the design of (PS). Note that equalities in ψ are eliminated by the rules (Eq1) or (Eq2) in case at least one of the variables is local, and are moved out of ψ by the rule (Lift) in case both variables are not local. □

Eliminating presolved forms

Definition 9 *The set of* constrained variables $\mathcal{C}(\psi)$ *of a conjunction* ψ *of FT-literals is the set of all variables* x *such that there is a literal* $x[f]y$ *in* ψ *for some variable* y *and some feature* f*.*

Proposition 40 *Let* $\psi = \exists\bar{x}\,\psi'$ *be a FT-presolved form with* $\mathcal{C}(\psi') \subseteq \bar{x}$*. Then* $\mathcal{A}_{FT} \models \forall\mathcal{FV}(\psi)\,\psi$*.*

Example

$$\forall y_1, y_2\, \exists x_1, x_2\, (\neg A(x_1) \wedge x_1[f]x_2 \wedge x_2[g]y_1 \wedge x_2[f]x_1 \wedge x_1 \neq x_2)$$

is valid in $\mathcal{A}_{FT}$.

This is proven analogously to Proposition 12. However, an additional argument is necessary since an FT-presolved form may contain an inequation $x \neq y$. Let ψ be a FT-presolved form as required in the proposition, and let f be a new feature symbol not occurring in ψ. Consider any valuation α of the free variables of ψ. Since f does not occur in ψ we can construct the valuation of the $\bar{x}$ such that every local variable has at the feature f a different subtree, and one that is different from all the subtrees of the values of the free variables at f (since ψ has only finitely many free variables, and since there are infinitely many feature trees).

Example

Let

$$\psi' = x[f]z \wedge x[g]x \wedge y[f]z \wedge y[g]y \wedge x \neq y$$

The construction of the proof of Proposition 12 yields for every value of z values for x and y which satisfy the above formula without the inequation $x \neq y$. However, we can strengthen the formula ψ by adding constraints that force x and y to take different values:

$$\psi'' = x[f]z \wedge x[g]x \wedge y[f]z \wedge y[g]y \wedge x[h]x \wedge yh{\uparrow} \wedge x \neq y$$

The construction of the proof of Proposition 12 to ψ'' yields, for every value of z, values for x and y that are different and that are hence a solution to ψ''. Since ψ'' implies ψ' we also have a solution to ψ'.

Hence, inequalities do not matter for the satisfiability of pre-solved forms. The only problem is that we have to ensure that in a pre-solved form all constrained variables are local. This is achieved by the following rule (Con). The rule (CleanFT) allows to get rid of any FT-presolved form in which all constrained variables are local.

(Con) $$\frac{\exists\bar{x}, x\, (y[f]x \wedge \psi)}{\neg yf{\uparrow} \wedge \forall x\, (y[f]x \rightarrow \exists\bar{x}\,\psi)} \quad y \notin \bar{x} \cup \{x\}$$

(CleanFT) $$\frac{\exists\bar{x}\,\psi}{\top} \quad \psi \text{ is FT-presolved}, \mathcal{C}(\psi) \subseteq \bar{x}$$

Proposition 41 *The rules (Con) and (CleanFT) are equivalence transformations in* $\mathcal{A}_{FT}$*. The system consisting the rules (Con), (CleanFT) and (Lift) is strongly terminating, and reduces any formula in FT-presolved form into a* Π_1*-formula.*

We hence arrive at

Theorem 42 *[BS95] The first-order theory of* $\mathcal{A}_{FT}$ *is decidable.*

7.7.1 Related Results

The above quantifier elimination procedure can be extended to CFT (plus the undefinedness predicate which is convenient to retain). The definition of pre-solved forms is extended by the clauses

1. if $xF \wedge xG \subseteq \psi$ then $F = G$
2. if $xF \wedge x[f]g \subseteq \psi$ then $f \in F$
3. if $xF \wedge xf\!\uparrow \subseteq \psi$ then $f \notin F$

The rule set (PS) for calculating pre-solved forms is now extended by

(Ar1) $\dfrac{xF \wedge xG}{\perp} \quad F \neq G$

(ArSel) $\dfrac{xF \wedge x[f]y}{\perp} \quad f \notin F$

(Ar2) $\dfrac{xF \wedge \neg xF}{\perp}$

(ArUndef) $\dfrac{xF \wedge xf\!\uparrow}{\perp} \quad f \in F$

(Ar3) $\dfrac{xF \wedge \neg xG}{xF} \quad F \neq G$

The problem is, however, that Proposition 40 does not easily extend to CFT. The reason is that with CFT we may loose the freedom to adjoin new features to make values different. For instance,

$$Ax \wedge x\{f\} \wedge x[f]z \wedge Ay \wedge y\{f\} \wedge y[f]z \wedge x \neq y$$

is not satisfiable. A solution can be found in [Tre97].

Theorem 43 *[Bac95] The first-order theory of $\mathcal{A}_{CFT}$ is decidable.*

Using the same techniques it is possible to show that the first-order theory of Herbrand is decidable. This particular result is due to Malc'ev [Mal71]. It is been rediscovered and extended to infinite signatures and structures of infinite trees by Maher [Mah88] and Comon and Lescanne [CL89].

Theorem 44 ([Mal71, Mah88, CL89]) *The first-order theory of $\mathcal{A}_H$ is decidable.*

Exercice 7
Show Theorem 44 by quantifier elimination in the spirit of the quantifier elimination procedure for $\mathcal{A}_{FT}$. For the sake of simplicity you may assume that the signature Σ is finite. (Why is this a simplification?)

In the quantifier elimination procedure given above we may, in the worst case, get an exponential blow-up of the size of the formula every time we reduce the number of alternations of quantifiers by 1. This can be seen as follows:

Consider a formula $\exists \bar{x}\, \psi$ where ψ has n variables. Hence there are $O(n)$ many atoms that can appear in ψ, hence the disjunctive normal form of ψ can have up to $O(2^n)$ many clauses. It may happen for each of these clauses that we introduce new universally quantified variables that all have to be renamed apart when we recombine the disjuncts into one formula $\forall \bar{y}\, \psi'$. This means that the length of $\bar{y}$ may be of the order of 2^n, hence ψ' can contain up to 2^n many different variables!

Theorem 45 *[Vor96],[Mie04] The problem of deciding validity of formulas in $\mathcal{A}_H$, $\mathcal{A}_{FT}$ or in $\mathcal{A}_{CFT}$ is non-elementary.*

This came as a surprise to many people working on quantifier elimination for theories of trees. The pitfall when evaluating the complexity of the algorithms is that many operations (like the transformation into disjunctive normal form) are left implicit, and are easily overseen when one looks only at the local transformation rules which do not look so bad after all.

The following two results have been shown using quantifier elimination techniques, the proofs are much more technically involved than the ones presented in this chapter:

Theorem 46 ([ZSM05]) *The first-order theory of $\mathcal{A}_{\mathrm{kbo}}$ is decidable.*

The quantifier elimination for the theory of $\mathcal{A}_{\mathrm{kbo}}$ relies on an extension of the language, the decidability results is hence for a more expressive theory than the theory of $\mathcal{A}_{\mathrm{kbo}}$.

Theorem 47 ([KR03]) *The first-order theory of $\mathcal{A}_{\mathrm{FSSub}}$ is decidable.*

The principal result of [KR03] is much more general, it states that the theory of a the so-called Σ-term-power of a structure can be reduced to the theory of that structure. Decidability of the theory of $\mathcal{A}_{\mathrm{SSub}}$ is still an open question. The complexity is known for basic constraints:

Theorem 48 ([Fre02]) *Satisfiability of basic constraints in both $\mathcal{A}_{\mathrm{SSub}}$ and $\mathcal{A}_{\mathrm{FSSub}}$ is decidable in PSPACE.*

Theorem 49 ([Tiu92]) *There are signatures and precedences for which satisfiability of constraints in $\mathcal{A}_{\mathrm{FSSub}}$ is PSPACE-hard.*

Chapter 8

Syntactic Methods: Coping with Non-Termination

Sometimes it is not possible to device syntactic transformations of formulas that always terminate. In this case we need additional criteria which allow us to cut infinite branches of the search tree.

8.1 Basic Concepts

There are two possible cases concerning the set of reachable formulas:

1. In the simplest of cases, there is only a finite number of possible formulas that can be reached by the syntactic transformation system. Such a transformation system is also called *quasi-terminating*. In this case we have to augment the algorithm by memorizing already generated formulas, which yields trivially a terminating algorithm. We now need a semantic argument to guarantee completeness of the algorithm. The criterion will now not be on normal form of the transformation system but on the complete (finite but cyclic) graph of transformations between formulas.

2. If the set of reachable formulas is infinite then the case is much more difficult. The difficulty is to find a criterion when to cut off branches of the search tree which ensures finiteness of search tree while maintaining completeness of the algorithm.

In this chapter we will first review a classical example of a quasi-terminating system which is restricted class of unification problems of words. The full problem of unification of equations over words was a long-standing open problems before being solved by Makanin [Mak77], improvements of the algorithm are still subject of ongoing research. Then, we present some of our own works falling in this class of problems which stem from the resolution of constraints over infinite trees.

In the previous chapters we have have seen that there are three fundamental properties of a constraint transformation procedure whcih must be fulfilled in order to obtain an algorithm for constraint solving: correctness, termination, and completeness. In the cases considered in the previous chapters only the proof of correctness used properties of the underlying semantic structure, while termination and completeness could be proof by purely symbolic techniques or were trivial. In case of quasi-terminating procedures, also the completeness proof depends on semantical properties of the structure.

$$\text{(CDecomp)}\quad \frac{at = as \wedge \phi}{t = s \wedge \phi} \qquad \text{(CClash)}\quad \frac{at = bs \wedge \phi}{\bot}\quad a \neq b$$

$$\text{(VDecomp)}\quad \frac{xt = xs \wedge \phi}{t = s \wedge \phi} \qquad \text{(Eq)}\quad \frac{x = t \wedge \phi}{x = t \wedge t\{x \mapsto t\}}\quad x \in \mathcal{FV}(\phi), x \neq t$$

$$\text{(VTriv)}\quad \frac{x = x \wedge \phi}{\phi} \qquad \text{(ETriv)}\quad \frac{\epsilon = \epsilon \wedge \phi}{\phi}$$

$$\text{(EClash)}\quad \frac{\epsilon = at \wedge \phi}{\bot} \qquad \text{(EDecomp)}\quad \frac{\epsilon = xt \wedge \phi}{\epsilon = x \wedge \epsilon = t \wedge \phi}$$

$$\text{(Letter)}\quad \frac{xt = a \wedge \phi}{(x = a \wedge t = \epsilon \wedge \phi) \vee (x = \epsilon \wedge t = a \wedge \phi)}$$

Figure 8.1: The set WE of rules to simplify word equations

8.2 Unification of Words

The following famous problem had been open for a long time, before finally being solved by Makanin [Mak77]. This problem is also known as *equation solving in free semi-groups.*

> Given terms $s_1, t_1, \ldots, s_n, t_n$ of the language of $\mathcal{A}_W$, does
>
> $$\mathcal{A}_W \models \tilde{\exists}(s_1 = t_1 \wedge \ldots \wedge s_n = t_n)$$
>
> hold ?

We start with a first system of rules for simplifying word equations which is given in Figure 8.1. As a first simplification we can use the fact that in the structure $\mathcal{A}_W$ under consideration, the multiplication is associative and that ϵ is a neutral element. We may hence write a term in the language of $\mathcal{A}_W$ as a string of variables and letters. In particular, we may always eliminate all occurrences of ϵ in a term, except when the term is ϵ itself. Furthermore we take $=$ to be symmetric, that is we identify an equation $s = t$ with $t = s$.

The system WE is obviously correct.

Exercice 8
Show that WE is terminating.

The normal forms of the system WE consist of a set of eliminated equations, plus a set of equations where both sides are terms with at least two members and starting on different symbols. We now consider another set of rules which are only applied to normal forms with respect to WE:

$$\text{(Var1)}\quad \frac{xv = aw \wedge \phi}{(x = \epsilon \vee \exists x'\ x = ax') \wedge xv = aw \wedge \phi}$$

$$\text{(Var2)}\quad \frac{xv = yw \wedge \phi}{(\exists x'\ x = yx' \vee \exists y'\ y = xy') \wedge xv = yw \wedge \phi}$$

Again these rules are obviously correct. After applying one of these rules, we transform the formula back into existentially quantified disjunctive normal form, and then normalize by the system WE. This leads in particular to an elimination of the existentially quantified variable. Since the solved variables do not matter for the existence of a solution, we can just ignore the solved equations and reuse the variable. This leads to the following alternative rule system which has no longer the property that the premise and the conclusion are semantically equivalent, but only that the premise is satisfiable if and only the conclusion is satisfiable:

$$\text{(Var1)} \quad \frac{xv = aw \wedge \phi}{(xv = aw \wedge \phi)\{x \mapsto \epsilon\} \vee (xv = aw \wedge \phi)\{x \mapsto ax\}}$$

$$\text{(Var2)} \quad \frac{xv = yw \wedge \phi}{(xv = yw \wedge \phi)\{x \mapsto yx\} \vee (xv = yw \wedge \phi)\{y \mapsto xy\}}$$

Note that in all cases the replacement will trigger certain simplifications by the system WE. For instance, when we replace in the equation $xv = aw$ the x by ax then we can cancel the prefix a on both sides. Unfortunately, this system is not terminating. Even worse, application of a transformation rule might make the equations increase in size.

Example

$\exists x\, x = axx$ transforms in one step into $\exists x\, x = axax$, and in n steps into $\exists x\, x = axa^n x$. Note that we cannot simply fix this problem by adding an "occur-check" since $\exists x\, ax = xa$ does have a solution.

However, the system is complete. Let $|w|$ denote the length of a word v. This extends to the size of valuations into $\mathcal{A}_W$ by

$$|\{x_1 \mapsto w_1, \ldots, x_n \mapsto w_n\}| = \Sigma_{i=1}^{n} |w_i|$$

For any *satisfiable* system of equations $s = e_1 \wedge \ldots \wedge e_n$ we denote

$$\mu(s) := \min\{|\alpha| \mid \mathcal{A}_W, \alpha \models s\}$$

Now, if s rewrites by the above system into a disjunction of systems and if s is satisfiable, then there is a system of equations s' in the resulting disjunct with $\mu(s') < \mu(s)$. This shows that if s is satisfiable then there is a derivation starting from s which contains $\top$.

Example

Consider the equation $axb = yybb$. One possible solution (in fact, the shortest solution) is $x = ab$ and $y = a$. We have

$$\begin{array}{ll} axb = yybb & y \mapsto ay \\ xb = yaybb & y \mapsto \epsilon \\ xb = abb & x \mapsto ax \\ xb = bb & x \mapsto bx \\ xb = b & x \mapsto \epsilon \\ \top & \end{array}$$

If we restrict the problem to so called *quadratic systems* where no variable has more than two occurrences in a single equation, then it is easy to show that although the system is still non-terminating the size of the equations does not increase. Hence,we can construct the complete graph of all reachable equations.

Proposition 50 *Satisfiability of quadratic systems of word equations is decidable.*

Example

Consider the equation $ax = xb$. With $x \mapsto \epsilon$ this reduces to $\perp$, and with $x \mapsto ax$ we get the same equation again. Since we constructed the complete graph and not found a solved form we conclude that this equation in not solvable.

In the general case, Makanin has given a criterion which allows to eventually cut infinite branches from the search tree, thus leading to decision algorithm. His criterion uses an advanced argument about the combinatorics of words.

Theorem 51 *[Mak77] Satisfiability of systems of word equations is decidable.*

A simple consequence is

Corollary 52 *([BS86]) The Σ_1-fragment of the theory of $\mathcal{A}_W$ is decidable.*

Proof: This follows from Theorem 51 and Proposition 8. □

We mention to other extensions

Theorem 53 *([Sch92]) Satisfiability of systems of word equations extended by unary predicates "$x \in R$" for all regular expressions R is decidable.*

Theorem 54 *([BS96]) The positive fragment of $\mathcal{A}_W$ is decidable in case of an infinite set of constants.*

Note, however, that the complete first-order theory of $\mathcal{A}_W$ is undecidable in case of alphabet containing at least two symbols (see Theorem 79), and that the positive theory of $\mathcal{A}_W$ is undecidable in case of a *finite* alphabet containing at least two symbols (Corollary 80).

The exact complexity of Word Unification is still an open problem. Makanin's original algorithm had a complexity of $4 - NEXPTIME$. The upper bound of the complexity of word unification has been improved step by step over the last 25 years. The best upper bound known so far has been obtained by a non-deterministic bottom-up procedure:

Theorem 55 *([Pla99]) Satisfiability of Word Equations is in PSPACE.*

The only lower bound known so far is

Theorem 56 *(Folk) Satisfiability of Word Equations is in NP-hard.*

This hold already for an alphabet consisting of one letter only.

Exercice 9
Prove Theorem 56 for an alphabet $\{a\}$ by reducing 3-SAT to the satisfiability of wordequations.

Theorem 57 *If the alphabet contains at least two symbols then for every system of wordequations there exists a single equivalent word equation, of size polynomial in the system.*

Proof: The system

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n$$

is equivalent to the single equation

$$s_1 a \ldots s_n a s_1 b \ldots s_n b = t_1 a \ldots t_n a t_1 b \ldots t_n b$$

$\square$

Exercice 10
Complete the proof by showing this equivalence.

A recent overview of Word Equations can be found in [Die02].

Chapter 9

Semantic Methods: Automatic Structures

We call a method for deciding (a fragment of) the first-order theory of a structure *semantic* if it is based on a reduction to another decidable problem. The most important semantic method consists in a translation to automata.

9.1 Automatic Structures

The use of automata for particular decision problems in mathematical logic goes back to the works of Büchi in the early sixties [Büc60b, Büc60a]. The exposition of the general method of *automatic structures* follows Blumensath and Grädel [BG00] which is the first systematic investigation of this concept.

9.1.1 Basic Concepts

The basic idea is to first provide an encoding of elements of the structure by words, and then to construct for any formula ϕ an automaton A_ϕ such that any tuple of words encoding a tuple of elements of the structure is accepted by the automaton A_ϕ if and only if the tuple of elements satisfies the formula ϕ. For technical reasons, this construction is restricted to purely *relational* signatures, that is signatures that do not contain constant symbols or function symbols. Note that it is always possible to transform a structure into a structure over a relational signature: we just have to replace the constants by unary predicates, and replace all n-ary functions by $n+1$-ary relations.

An important technical problem lies in the fact that the set of solutions to a formula is not a set of values, but rather a set of n-tuples of values where n is the number of free variables of the formula. That is, we have to provide a way to extend our encoding of elements of the structure to encodings of tuples of elements.

Definition 10 *Let Σ be a finite alphabet and $\Box \notin \Sigma$. We define the alphabet $\Sigma_n^\Box$, for $n \geq 0$, as*

$$\Sigma_n^\Box = \{(c_1, \ldots, c_n) \mid c_i \in \Sigma \cup \{\Box\} \text{ and } c_i \neq \Box \text{ for some } i\}$$

The convolution *of words $x_1, \dots, x_n \in \Sigma^*$ is defined as*

$$x_1 \otimes \ldots \otimes x_n := \begin{bmatrix} x_1^1 \\ \vdots \\ x_n^1 \end{bmatrix} \cdots \begin{bmatrix} x_1^l \\ \vdots \\ x_n^l \end{bmatrix}$$

where $l = max\{|x_i| \mid i \leq i \leq n\}$ is the length of the longest word among the x_i, and

$$x_i^j = \begin{cases} \text{the } j\text{-th symbol of } x_i \text{ if } j \leq |x_i| \\ \square \text{ otherwise} \end{cases}$$

As a consequence of Definition 10, $\Sigma_1^\square = \Sigma$, and $\Sigma_n^\square$ is the empty set. There are exactly two languages over the empty alphabet: the empty language, and the language consisting of the empty word only.

Example

Let $\Sigma = \{a, b\}$. We have that

$$a \otimes bba \otimes aa = \begin{bmatrix} a \\ b \\ a \end{bmatrix} \begin{bmatrix} \square \\ b \\ a \end{bmatrix} \begin{bmatrix} \square \\ a \\ \square \end{bmatrix}$$

Definition 11 *Let $\mathcal{A}$ be a structure over a relational signature with relation symbols $R_1, \dots, R_n$. An* automatic representation *of $\mathcal{A}$ is given by*

1. *a finite alphabet Σ,*
2. *a regular language $L_\delta \subseteq \Sigma^*$,*
3. *an onto function $\nu: L_\delta \to \mathcal{A}$,*
4. *a regular language $L_R \subseteq (\Sigma_n^\square)^*$ for each n-ary relation symbol R of the signature of $\mathcal{A}$, such that for all $x_1, \dots, x_n \in L_\delta$:*

$$x_1 \otimes \ldots \otimes x_n \in L_R \text{ iff } (\nu(x_1), \dots, \nu(x_n)) \in R^{\mathcal{A}}$$

The representation is called one-to-one *if ν is a one-to-one function. A structure is* automatic *if it has an automatic representation.*

Note that there may be several possible automata for a given atomic formula since the behavior of the automaton is not specified when the input word is not in $L_\delta \otimes \cdots \otimes L_\delta$.

9.1.2 Example: Presburger Arithmetic

The canonical example of a an automatic structure is Presburger Arithmetic. This was somewhat indirect in Büchi's original work [Büc60b] since he was mostly interested in automata for restricted forms of second order arithmetic, a special case of what is now known as Rabin's theorem (see Section 9.3 below). The automatic presentation of Presburger arithmetic can be derived from this result. Here we follow the direct construction by Boudet and Comon [BC96].

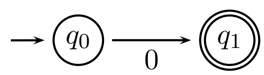
Since our technique applies only to relational structures we have to slightly modify the language. The new relational language contains the following predicates:

- $x = 0$
- $x = y + z$

Exercice 11
Explain how to represent the formula $x = 1$ in this restricted relational language.

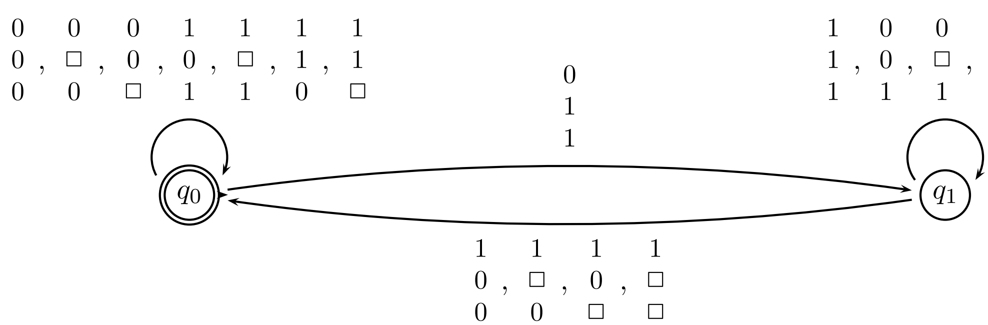
Let $\mathcal{A}$ be the structure of Presburger arithmetic in the restricted relational language. The idea is to represent a natural number by its binary notation with the least significant bit first. With this representation, the automaton for $x = y + z$ just mimics the usual bit-wise addition algorithm. We choose

1. $\Sigma = \{0, 1\}$
2. $L_\delta = 0 + (0 + 1)^*1$
3. $\nu(0) = 0, \nu(1) = 1, \nu(0w) = 2\nu(w), \nu(1w) = 1 + 2\nu(w)$
4. The automaton for $x_1 = 0$ is



   Another possible automaton, consisting of only one state q_0, recognizes the language 0^*. This is legal since we only care for words in L_δ, and since we have $L_\delta \cap 0^* = \{0\}$.

5. The automaton for $x_1 = x_2 + x_3$ is



   In this automaton, the state q_0 represents a carry-over of 0, and the state q_1 represents a carry-over of 1. Note that some letters do not occur in the transitions of this automaton: there is no letter with the first component $\square$ since the representation of the value of x_1 cannot be shorter than the representations of the values of x_2 and x_3. Also note that this automaton accepts words that are not $L_\delta \otimes L_\delta \otimes L_\delta$, as for instance $(0, \square, 0)(1, 1, \square)$.

9.1.3 Decidability Results

Theorem 58 ([BG00]) *The theory of any automatic structure is decidable.*

Proof: We show by induction on the the formula ϕ that we can construct an automaton A_ϕ for any formula ϕ such that for all $x_1, \ldots, x_n \in L_\delta$:

$$x_1 \otimes \ldots \otimes x_n \in L_{A_\phi} \text{ iff } (\nu(x_1), \ldots, \nu(x_n)) \in \phi^A$$

For atomic formulas this is a hypothesis of the theorem.

The automaton $A_{\neg\phi}$ is defined as the complement automaton of A_ϕ, that is the language recognized by $A_{\neg\phi}$ is $L_{A_{\neg\phi}} = (\Sigma_n^\square)^* \setminus L_{A_\phi}$.

For any $x_1, \ldots, x_n \in L_\delta$:

$$\begin{aligned} x_1 \otimes \ldots \otimes x_n \in L_{A_{\neg\phi}} &\Leftrightarrow x_1 \otimes \ldots \otimes x_n \in (\Sigma_n^\square)^* \setminus L_{A_\phi} \\ &\Leftrightarrow x_1 \otimes \ldots \otimes x_n \notin L_{A_\phi} \\ &\Leftrightarrow (\nu(x_1), \ldots, \nu(x_n)) \notin \phi^A \\ &\Leftrightarrow (\nu(x_1), \ldots, \nu(x_n)) \in (\neg\phi)^A \end{aligned}$$

Note that another possibility for $A_{\neg\phi}$ would have been, for instance, the intersection of the complement of A_ϕ with the automaton recognizing $L_\delta \otimes \ldots \otimes L_\delta$.

Given a conjunction $\phi_1 \wedge \phi_2$ we can, by induction hypothesis, build the automata A_{ϕ_1} and A_{ϕ_2}. Note that the two formulas may have different sets of free variables. Hence, we first have to extend the alphabets of A_{ϕ_1} and A_{ϕ_2} by new components which are not taken into consideration in the transitions. *Cylindrification* of an automaton $\mathcal{A}$ working on n-tuples yields an automaton $\mathcal{A}^\uparrow$ working on $(n+1)$-tuples (where we assume for the sake of notation that the newly added component is the $n+1$-th one):

- $\mathcal{A}^\uparrow$ has the same states, initial state and accepting states as $\mathcal{A}$.
- If $\mathcal{A}$ contains a transition

  $$q, (c_1, \ldots, c_n) \;\; \rightarrow \;\; q'$$

  then $\mathcal{A}^\uparrow$ contains for every $c \in \Sigma \cup \{\square\}$ the transition

  $$q, (c_1, \ldots, c_n, c) \;\; \rightarrow \;\; q'$$

- If q is an accepting state of $\mathcal{A}$ then $\mathcal{A}^\uparrow$ contains for every $c \in \Sigma$ the transition

  $$q, (\square, \ldots, \square, c) \;\; \rightarrow \;\; q$$

The central property of cylindrification is: for all $x_1, \ldots, x_n, x_{n+1} \in \Sigma^*$:

$$x_1 \otimes \ldots \otimes x_n \in L_A \Leftrightarrow x_1 \otimes \ldots \otimes x_n \otimes x_{n+1} \in L_{A^\uparrow}$$

Let $\mathcal{FV}(\phi_1 \wedge \phi_2) = \{x_1, \ldots, x_n\}$, and $\mathcal{FV}(\phi_1) = \{x_{i_1}, \ldots, x_{i_k}\}$, $\mathcal{FV}(\phi_2) = \{x_{j_1}, \ldots, x_{j_l}\}$. Let $A^{\uparrow}_{\phi_1}$ (resp. $A^{\uparrow}_{\phi_2}$) be the appropriately cylindrification of A_{ϕ_1} (resp. A_{ϕ_2}). We define $A_{\phi_1 \wedge \phi_2}$ as the product of $A^{\uparrow}_{\phi_1}$ and $A^{\uparrow}_{\phi_2}$, since then for all $x_1, \ldots, x_n \in L_\delta$:

$$\begin{aligned}
x_1 \otimes \ldots \otimes x_n \in L_{A_{\phi \wedge \phi_2}} &\Leftrightarrow x_1 \otimes \ldots \otimes x_n \in L_{A^{\uparrow}_{\phi_1}} \\
&\ \text{and } x_1 \otimes \ldots \otimes x_n \in L_{A^{\uparrow}_{\phi_2}} \\
&\Leftrightarrow x_{i_1} \otimes \ldots \otimes x_{i_k} \in L_{A_{\phi_1}} \\
&\ \text{and } x_{j_1} \otimes \ldots \otimes x_{j_l} \in L_{A_{\phi_2}} \\
&\Leftrightarrow (\nu(x_{i_1}), \ldots, \nu(x_{i_k})) \in \phi_1^A \\
&\ \text{and } (\nu(x_{j_1}), \ldots, \nu(x_{j_l})) \in (\phi_2)^A \\
&\Leftrightarrow (\nu(x_1), \ldots, \nu(x_n)) \in (\phi_1 \wedge \phi_2)^A
\end{aligned}$$

We obtain $A_{\exists x_i\, \phi}$ by *projection* from A_ϕ, that is by simply erasing the i-th component of the symbols. That is, if the original automaton A contains a transition

$$q, (c_1, \ldots, c_{i-1}, c_i, c_{i+1}, \ldots, c_n) \quad \rightarrow \quad q'$$

then the projected automaton $A^{\downarrow}$ contains

$$q, (c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n) \quad \rightarrow \quad q'$$

The central property of projection is: for all $x_1, \ldots, x_n, \in \Sigma^*$:

$$\text{exists } x_{n+1} \in \Sigma^* \text{ with } x_1 \otimes \ldots \otimes x_n \otimes x_{n+1} \in L_A \Leftrightarrow x_1 \otimes \ldots \otimes x_n \in L^{\downarrow}_A$$

Hence, we obtain

$$\begin{aligned}
x_1 \otimes \ldots \otimes x_n \in L_{A_{\exists x\, \phi}} &\Leftrightarrow x_1 \otimes \ldots \otimes x_n \in L_{A^{\downarrow}_\phi} \\
&\Leftrightarrow \text{exists } x_{n+1} \in \Sigma^* \text{ with } x_1 \otimes \ldots \otimes x_n \otimes x_{n+1} \in L_A \\
&\Leftrightarrow \text{exists } x_{n+1} \in \Sigma^* \text{ with } (\nu(x_1), \ldots, \nu(x_n), \nu(x_{n+1})) \in (\phi)^A \\
&\Leftrightarrow (\nu(x_1) \otimes \ldots \otimes \nu(x_n)) \in \exists x\, \phi^A
\end{aligned}$$

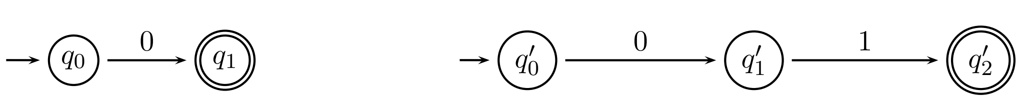
Finally, to decide validity of a closed sentence ϕ we construct the automaton A_ϕ and check whether $\mathcal{L}(A_\phi) \neq \emptyset$. $\Box$

Note that our construction yields for any closed formula an automaton over the alphabet of 0-tuples. This alphabet consists of one symbol, namely the empty tuple ().

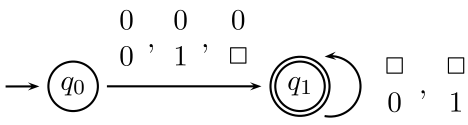
It should be remarked that a decision procedure based on this method has a non-elementary complexity since computing the complement of an automaton requires determinisation.

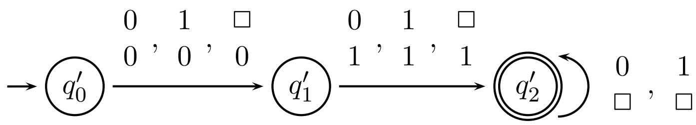
Example

We illustrate the two crucial steps in this construction at hand of the example Presburger Arithmetic. The automaton for the two formulas $x_1 = 0$ and $x_2 = 2$ are as follows:

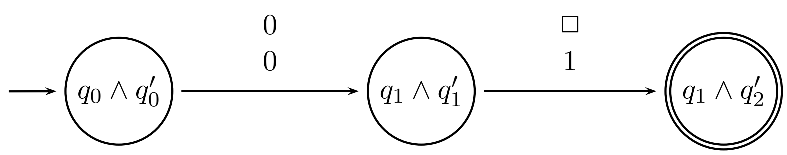


Cylindrification yields the following automata:

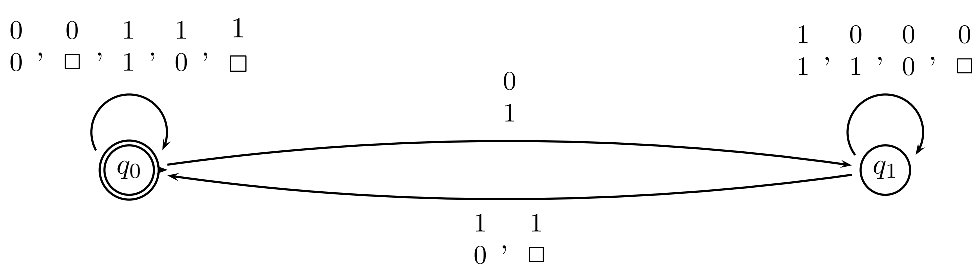




Finally, the product of these two automata yields as expected



We obtain the automaton for $\exists x_2\,(x_1 = x_2 + x_3)$ by projecting away the second component of the automaton for $x_1 = x_2 + x_3$ given above:

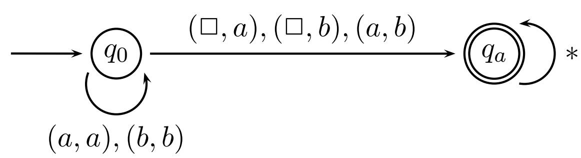


This automatons accepts the pairs of integers where at the highest position at which the binary representations differ the bit for x_1 is 1 and the bit for x_3 is 0. In other words, this automaton corresponds to the formula $x_1 \geq x_3$, which is equivalent to $\exists x_2\,(x_1 = x_2 + x_3)$ as expected.

Lemma 59 *Let $\mathcal{A}$ be an automatic structure containing the equality relation. Then $\mathcal{A}$ has a one-to-one automatic representation.*

Proof: It is sufficient to define a regular subset $L_{\delta'} \subseteq L_\delta$ that contains exactly one element from each equivalence class of L_δ modulo $L_=$.

Let $L_<$ the automaton that defines the binary relation "x_1 is strictly smaller than x_2 in the lexicographic ordering" (in fact the strict part of any total order on Σ^* would do). Here is the automaton for the case of a binary alphabet $\{a, b\}$:



Let A_1 be the intersection of $L_=$ and $L_<$, A_2 the projection of A_1 obtained by hiding the first component x_1. The automaton for $L_{\delta'}$ is the intersection of L_δ with the complement of A_2. $\square$

In fact, the theory of any automatic structure in a slight extension of first-order logic is decidable: Let the quantifier $\exists^\infty x\, \phi$ denote "there are infinitely many values for x such that ϕ holds". Adding this quantifier is a proper extension of first-order logic since in this extended logic the compactness theorem [CK90] no longer holds.

Theorem 60 ([BG00]) *Let $\mathcal{A}$ be a relational structure with a one-to-one automatic representation. Then the theory of $\mathcal{A}$ in the extension of first-order logic by the quantifier $\exists^\infty$ is decidable.*

Proof: We just have to extend the construction of the proof of Theorem 58 by the construction of the automaton for $\exists^\infty x\, \phi$. Let us assume for the sake of simplification of notation that $\mathcal{FV}(\phi) = \{x_1, x_2\}$, and that we wish to construct the automaton for $\exists^\infty x_2\, \phi$. Let A_ϕ be the automaton for the formula ϕ, and n the number of states of A_ϕ. Thanks to the pumping property [HU79] of automata, for any given value of a_1 there exist infinitely many values of a_2 satisfying ϕ if and only if there exists a value of a_2 with

1. $(a_1, a_2) \in \phi^{\mathcal{A}}$
2. there exist $x_1, x_2 \in L_\delta$ with $\nu(x_1) = a_1$, $\nu(x_2) = a_2$, and $|x_2| > |x_1| + n$.

The automaton for $\exists^\infty x_2\, \phi$ is hence simply obtained by

1. the product of the automaton for ϕ, and the automaton that verifies that the length of the second component exceeds the length of the first component by at least $n + 1$.
2. Projecting away the variable x_2. $\square$

Exercice 12
Let $\mathcal{A}_1$ be the structure following structure:

- The language consists of the following binary relation symbols:
  $=$, *equal-length*, $\leq$, *pre-a*, *post-a*, *pre-b*, *post-b*.
- The universe is $\{a, b\}^*$.
- The relation symbols are interpreted as follows:
  - $=$ is equality
  - *equal-length*(v, w) iff v and w have the same length
  - $v \leq w$ iff v is a prefix of w
  - *pre-a*(v, w) iff $v = aw$, *pre-b*(v, w) iff $v = bw$
  - *post-a*(v, w) iff $v = wa$, *post-b*(v, w) iff $v = wb$

Show that the structure $\mathcal{A}_1$ is automatic.

9.2 Automatic Structures and Tree Automata

Definition 11 defined automatic structures using recognizable languages in the classical sense, that is languages that are recognized by finite automata over finite words. This easily generalizes to other classes of automata: we just have to define a notion of convolution on the syntactic objects recognized by the new class of automata, and we have to ensure that the class of automata is effectively closed under Boolean operations, cylindrification, projection, and that emptiness of recognizable languages is decidable. Definition 14 below gives an example of such a generalization to the class of tree-automatic structures. In general we can speak of $\mathcal{C}$-automatic structures when $\mathcal{C}$ is a class of automata.

In order to generalize the technique of the preceding section to structures represented by trees we have to generalize the convolution operation to trees.

Definition 12 *Let Σ be a signature with $\Box \notin \Sigma$. We define the signature $\Sigma_n^\Box$, for $n \geq 1$, as*

$$\Sigma_n^\Box = \{(f_1, \ldots, f_n) \mid f_i \in \Sigma \cup \{\Box\} \text{ and } f_i \neq \Box \text{ for some } i\}$$

The arity of $[f_1 \ldots, f_n]$ in $\Sigma_n^\Box$ is the maximum of the arities of those f_i that are in Σ.

The signature $\Sigma_0^\Box$ consists of the constant $()$ only.

The convolution $t_1 \otimes \ldots \otimes t_n$ is the tree t defined by

- $D_t = D_{t_1} \cup \ldots \cup D_{t_n}$
- $L_t(\pi) = [f_1, \ldots, f_n]$ *where* $f_i = L_{t_i}(\pi)$ *if* $\pi \in D_{t_i}$, *and* $f_i = \Box$ *otherwise.*

In contrast to Definition 10, the definition of the signature $\Sigma_n^\Box$ needs a special case for $n = 0$. The reason for this is that that every tree has a non-empty domain, while words may be empty. As a consequence, we need one constant in the tree signature in case $n = 0$.

Example

Let $\Sigma = \{h, f, a\}$, where a is a constant, f unary, and h binary. Then we have that

$$f(a) \otimes h(a, f(a)) = [f, h]([a, a], [\Box, f]([\Box, a]))$$

Definition 13 (Projection of a Tree) *Let t be a $\Sigma_n^\Box$-tree and $1 \leq i \leq n$. The i-th* component *of t, denoted $t.i$, is the Σ-tree t_i such that there exist Σ-trees $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ with*

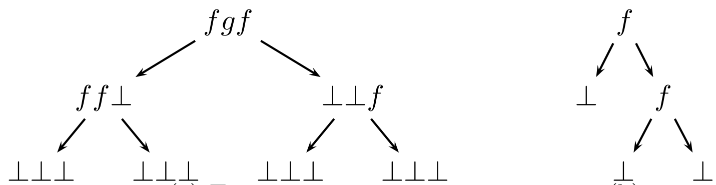
$$t = t_1 \otimes \ldots \otimes t_{i-1} \otimes t_i \otimes t_{i+1} \otimes \ldots \otimes t_n$$

Note that $O(\tau.i) \subseteq O(t)$. This inclusion may be strict, as is illustrated in the example given on Figure 9.1.

Proposition 61 *For every $t_1, \ldots, t_n \in T(\Sigma)$ there is a unique $t \in T(\Sigma_n^\Box)$ with $t.i = t_i$ for $1 \leq i \leq n$.*

Proof: The tree t is defined by $O(t) := \bigcup_{1 \leq i \leq n} O(t_i)$ and $t(\pi)_i = t_i(\pi)$ if $\pi \in O(t_i)$, $t(\pi)_i = \perp$ if $\pi \in O(t) - O(t_i)$. $\Box$

Now, one can define *tree-automatic representations* and *tree-automatic structure* analogously to Definition 11:



(a) Tree τ (b) $\tau.3$

Figure 9.1: Example of a component of a tree.

Definition 14 *Let $\mathcal{A}$ be a structure over a relational signature with relation symbols $R_1, \ldots, R_n$. A* tree-automatic representation *of $\mathcal{A}$ is given by*

1. *a finite signature Σ,*
2. *a regular tree language $L_\delta \subseteq T(\Sigma)$,*
3. *an onto function $\nu\colon L_\delta \to \mathcal{A}$,*
4. *a regular tree language $L_R \subseteq T(\Sigma_n^{\square})$ for every relation symbol R of the signature of $\mathcal{A}$, such that for all $x_1, \ldots, x_n \in L_\delta$:*

$$x_1 \otimes \ldots \otimes x_n \in L_R \text{ iff } (\nu(x_1), \ldots, \nu(x_n)) \in R^{\mathcal{A}}$$

A structure is tree-automatic *if it has a tree-automatic representation.*

We obtain as in the case of Theorem 58:

Theorem 62 *The theory of any tree-automatic structure is decidable.*

Exercice 13
Does Lemma 59 transfer to tree-automatic structures?

9.2.1 Example: WS2S

Theorem 63 *([TW68]) The theory of $\mathcal{A}_{WS2S}$ is decidable.*

In order to apply Theorem 62 we have to move the structure to a purely relational language. Furthermore we avoid the difficulty of encoding two different sorts in the tree language by working with one sort of sets, where words are represented as singleton sets. The modified structure is

Structure $WS2S_0$

The language of $\mathcal{A}_{WS2S0}$ contains

- *binary relations $\subseteq$, S_0 and S_1*

The domain of $\mathcal{A}_{WS2S0}$ is FinSets($\{0,1\}^*$). The predicate $S_0(V, W)$ is true in $\mathcal{A}_{WS2S0}$ iff V is a singleton set $\{v\}$, and $W = \{0(v)\}$. Analogously, $S_1(V, W)$ is true in $\mathcal{A}_{WS2S0}$ iff V is a singleton set $\{v\}$, and $W = \{1(v)\}$. $\subseteq$ is interpreted as set inclusion.

We can transform any $\mathcal{A}_{WS2S}$-sentence σ into a $\mathcal{A}_{WS2S0}$-sentence σ_0 such that $\mathcal{A}_{WS2S0} \models \sigma_0$ iff $\mathcal{A}_{WS2S} \models \sigma$. First note that we can express in $\mathcal{A}_{WS2S0}$ that a set is a singleton set:

$$\texttt{singleton}(X) \quad := \quad \exists Y\, \neg X \subseteq Y \wedge \forall Y, Z\, (Y \subseteq X \wedge Z \subseteq X \rightarrow (Z \subseteq Y \vee Y \subseteq Z))$$

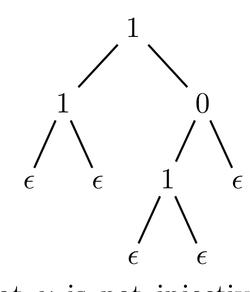
We can now transform a formula σ:

1. Eliminate ϵ: Replace $x = \epsilon$ by $\neg \exists y\, (y0 = x \vee y1 = x)$
2. Eliminate nested terms and equalities between two non-variable terms. For instance, $x0 = y01$ is rewritten as
$$\exists z_1, z_2\, (x0 = z_1 \wedge y0 = z_2 \wedge z_2 1 = z_1)$$
3. Replace all w-variables by s-variables:
$$\begin{array}{rcl} \forall x\, \phi(x) & \mapsto & \forall X\, (\texttt{singleton}(X) \rightarrow \phi[X/x]) \\ \exists x\, \phi(x) & \mapsto & \exists X\, (\texttt{singleton}(X) \wedge \phi[X/x]) \\ x = y & \mapsto & X = Y \\ x0 = y & \mapsto & S_0(X, Y) \\ x1 = y & \mapsto & S_1(X, Y) \\ x \in Y & \mapsto & X \subseteq Y \end{array}$$
4. Replace $X = Y$ by $X \subseteq Y \wedge Y \subseteq X$

We apply Theorem 62 to the structure $\mathcal{A}_{WS2S0}$. We choose the tree signature Σ consisting of a constant ϵ and two binary function symbols 0 and 1. We choose $L_\delta = T(\Sigma)$, and we define the set of words represented by Σ-tree as

$$\begin{array}{rcl} \nu(\epsilon) & = & \emptyset \\ \nu(0(t_1, t_2)) & = & 0\nu(t_1) \cup 1\nu(t_2) \\ \nu(1(t_1, t_2)) & = & \{\epsilon\} \cup 0\nu(t_1) \cup 1\nu(t_2) \end{array}$$

For example, the tree



represents the set $\{\epsilon, 0, 10\}$. Note that ν is not injective, since for instance both ϵ and $0(\epsilon, \epsilon)$ represent the empty set of words.

The automaton for the predicate $X_1 \subseteq X_2$ verifies that the tree does not contain any of $[1, 0]$, $[1, \epsilon]$, or $[1, \Box]$ It has one state q which is accepting:

$$\begin{array}{lll lll lll lll} & & & [\Box, \epsilon] & \rightarrow & q & [\Box, 0](q, q) & \rightarrow & q & [\Box, 1](q, q) & \rightarrow & q \\ [\epsilon, \Box] & \rightarrow & q & [\epsilon, \epsilon] & \rightarrow & q & [\epsilon, 0](q, q) & \rightarrow & q & [\epsilon, 1](q, q) & \rightarrow & q \\ [0, \Box](q, q) & \rightarrow & q & [0, \epsilon](q, q) & \rightarrow & q & [0, 0](q, q) & \rightarrow & q & [0, 1](q, q) & \rightarrow & q \\ & & & & & & & & & [1, 1](q, q) & \rightarrow & q \end{array}$$

The automaton for $S_0(X_1, X_2)$ verifies that he sees never $[1, 1]$, exactly once $[1, 0]$ and exactly once $[0, 1]$, and that the occurrence of $[0, 1]$ is the left son of the occurrence of $[1, 0]$. The automaton has three states q_0, q_1, q_2 where q_2 is accepting.

$$\begin{array}{lcl}
[\epsilon,\Box] & \to & q_0 \\
{[0,\Box](q_0,q_0)} & \to & q_0
\end{array}
\quad
\begin{array}{lcl}
[\Box,\epsilon] & \to & q_0 \\
{[\epsilon,\epsilon]} & \to & q_0 \\
{[0,\epsilon](q_0,q_0)} & \to & q_0
\end{array}
\quad
\begin{array}{lcl}
[\Box,0](q_0,q_0) & \to & q_0 \\
{[\epsilon,0](q_0,q_0)} & \to & q_0 \\
{[0,0](q_0,q_0)} & \to & q_0 \\
{[1,0](q_1,q_0)} & \to & q_2 \\
{[0,0](q_0,q_2)} & \to & q_2
\end{array}
\quad
\begin{array}{lcl}
[\Box,1](q_0,q_0) & \to & q_1 \\
{[\epsilon,1](q_0,q_0)} & \to & q_1 \\
{[0,1](q_0,q_0)} & \to & q_1 \\
 & & \\
{[0,0](q_2,q_0)} & \to & q_2
\end{array}$$

The automaton construction given here looks, in comparison to the one usually given in the literature, slightly bloated. This is due to the fact that we have distinguished the symbols $\Box$,ϵ and 0, while many presentations confuse these three symbols.

The automaton for $S_1(X_1, X_2)$ is defined analogously.

Exercice 14
Show that $\mathcal{A}_{Sk}$, that is the structure of Skolem arithmetic, is tree-automatic (and hence decidable).

Hint: think of the prime factorization of natural numbers.

9.2.2 Example: The First-Order Theory of Terms Modulo Certain Ground Equations

Let Σ be a signature consisting of an infinite number of constants and of a finite number of non-constant function symbols. Let Ax be a set of ground equations of the form

$$Ax = \{c_1 = t_1, \ldots, c_n = t_n\}$$

with the following properties :

- $c_i \neq c_j$ if $i \neq j$
- $c_i \notin \texttt{subterms}(t_j)$ for all $1 \leq i \leq n$ and $1 \leq j \leq n$

and let $G \subseteq T(\Sigma)$ be a finite set of ground terms. We define $C_\Pi = \{c_1, \ldots, c_n\}$.

Structure $\mathcal{A}_{Ax,G}$

the language of $\mathcal{A}_{Ax,G}$ consists of

- *a unary predicate $x = t$ for each $t \in G$;*
- *a unary predicate $t \leq x$ for each $t \in G$;*
- *for any subset $E \subseteq Ax$ a binary relation $x =_E y$*

The universe of the structure of $\mathcal{A}_{Ax,G}$ is $T(\Sigma)$.

- *A unary predicate $x = t$ is true if the value of x is t.*
- *A unary predicate $t \leq x$ is true if t is a subterm of the value of x.*

- *A binary predicate $x =_E y$ is true if the respective values of x and y are congruent modulo the equational theory E.*

Note that this structure, in contrast to the Herbrand structure, does not feature function symbols in the logical language. As a consequence, one can not express relations like $x = f(y, z)$.

The congruence relations modulo sub-systems of E are of a very special nature: Congruence modulo E is in fact equivalent to one-step parallel joinability. That is, for $E = \{c_{i_1} = t_{i_1}, \ldots, c_{i_m} = t_{i_m}\} \subseteq Ax$ let ρ_E be the replacement $[c_{i_1} \mapsto t_{i_1}, \ldots, c_{i_m} \mapsto t_{i_m}]$. Then it is easy to see that for any two ground terms $s, t \in T(\Sigma)$ one has that $t =_E s$ iff $\rho_E(s) = \rho_E(t)$.

Lemma 64 *The structure $\mathcal{A}_{Ax,G}$ is tree-automatic for any Ax, and G with the restrictions given above.*

Exercice 15
Show Lemma 64.

9.2.3 Example: The First-Order Theory of a Monadic RPO

We call a signature *monadic* if it consists only of constants and unary function symbols. The goal of this section is to show that the structure $\mathcal{A}_{\mathrm{lpo}}$ in case of a monadic signature and a total precedence is tree-automatic. In order to to this we have to consider a relational variant of $\mathcal{A}_{\mathrm{lpo}}$, consisting of the relations

- $x_1 \succ_{\mathrm{lpo}} x_2$
- $x_1 = x_2$
- $x_1 = c$ for any constant $c \in \Sigma_0$
- $x_1 = a(x_2)$ for any unary function symbol $a \in \Sigma_1$

It is easy to see that this variant is equivalent in expressivity to the original $\mathcal{A}_{\mathrm{lpo}}$. We could have dropped the equality relation $x_1 = x_2$ since it can be expressed as $x_1 \not\succ_{\mathrm{lpo}} x_2 \wedge x_2 \not\succ_{\mathrm{lpo}} x_1$, or as $\exists z(z = a(x_1) \wedge z = a(x_2))$ in case $a \in \Sigma_1$. The reason for leaving the equality relation in is that the automaton for the equality relation will be necessary anyway as sub-automaton for the other relations.

We will assume for simplicity that there is exactly one constant $c \in \Sigma_0$. in this case, we can identify trees with words, by virtue of the isomorphism $\omega : T(\Sigma) \to \Sigma_1^*$ defined by $\omega(c) = \epsilon$ and $\omega(a(x)) = a \cdot \omega(x)$ for any $a \in \Sigma_1$ and $x \in T(\Sigma)$. For instance, $\omega(a(b(b(c)))) = abb$.

We repeat the definition of the lexicographic path ordering for a given precedence $<$ on Σ_1 in case of words: $s \succ_{\mathrm{lpo}} t$ if one of the following holds:

1. $s \neq \epsilon$ and $t = \epsilon$
2. $s = as'$ and $t = bt'$, and either
   (a) $a > b$ and $s \succ_{\mathrm{lpo}} t'$ or
   (b) $a = b$ and $s' \succ_{\mathrm{lpo}} t'$ or
   (c) $a < b$ and $s' \succ_{\mathrm{lpo}} t$

From this definition it should be clear that $\mathcal{A}_{\mathrm{lpo}}$ is not an automatic structure, at least not if we take the representation of trees as words as given by the function ω (that is, $\nu = \omega^{-1}$). The reason is that in the relation $\succ_{\mathrm{lpo}}$ is defined by a traversal from left to right, but the traversal of the two words advances at a different pace in the two cases 2.(a) and 2.(c).

Proposition 65 *If a is greater then any symbol in t then $s_1as_2 \succ_{\mathrm{lpo}} t$ for any words s_1, s_2.*

Proof: We have $as_2 \succ_{\mathrm{lpo}} \epsilon$ by the definition of $\succ_{\mathrm{lpo}}$, hence $as_2 \succ_{\mathrm{lpo}} t$ by the fact that a is greater than any symbol of t, and hence $s_1as_2 \succeq_{\mathrm{lpo}} as_2 \succ_{\mathrm{lpo}} t$ by the subterm property. □

Proposition 66 *If $s \succ_{\mathrm{lpo}} t$ and a is greater than any symbol in t' then $s'as \succ_{\mathrm{lpo}} t'at$ for any word s'.*

Proof: If $s \succ_{\mathrm{lpo}} t$ then $as \succ_{\mathrm{lpo}} at$, hence $as \succ_{\mathrm{lpo}} t'at$ since a is greater than any symbol in t', and we conclude with the subterm property that $s'as \succeq_{\mathrm{lpo}} as \succ_{\mathrm{lpo}} t'at'$. □

The key to the tree-automatic representation is the observation that we can define the order $\succ_{\mathrm{lpo}}$ by an induction on the precedence relation. For a given word w, let us denote by $max(w)$ the maximal (in the precedence $>$) symbol occurring in w, and denote by $mul(w)$ the number of occurrences of $max(w)$ in w. We define a new binary relation $\gg$ on Σ_1^* : We say that $s \gg t$ if one of the following holds:

1. $max(s) > max(t)$ or
2. $max(s) = max(t)$ and $mul(s) > mul(t)$ or
3. $a = max(s) = max(t)$, $mul(s) = mul(t)$, $s = s_0as_1a\cdots as_n$, $t = t_0at_1a\cdots at_n$, the s_i and t_i contain only letters smaller than a, and there is an i such that $s_i \gg t_i$ and $s_j = t_j$ for all $j > i$.

Proposition 67 *For any $s, t \in \Sigma_1^*$ we have that $s \succ_{\mathrm{lpo}} t$ iff $s \gg t$.*

Proof: First note that we only have to prove that $s \gg t$ implies $s \succ_{\mathrm{lpo}} t$. The inverse direction follows immediately : Suppose that $s \succ_{\mathrm{lpo}} t$. We obviously have by the definition of the relation $\gg$ that one of $s \gg t$, $s = t$, or $t \gg s$ holds. $s = t$ is impossible since $\succ_{\mathrm{lpo}}$ is a strict order, and if we had $t \gg s$ then this would imply $t \succ_{\mathrm{lpo}} s$, which is impossible since $\succ_{\mathrm{lpo}}$ is a strict order.

If the first case of the definition of $\gg$ applies then we have that $s \succ_{\mathrm{lpo}} t$ by Proposition 65, and in the two remaining cases we have that $s \succ_{\mathrm{lpo}} t$ by Proposition 66. □

The representation of a word $w \in \Sigma_1^*$ can now be defined by an induction on the alphabet Σ_1. If the alphabet consists of only one letter a then we represent a word a^n of length n as the tree

$$a(\epsilon, a(\epsilon, a(\ldots, a(\epsilon, \epsilon_a)\ldots)))$$

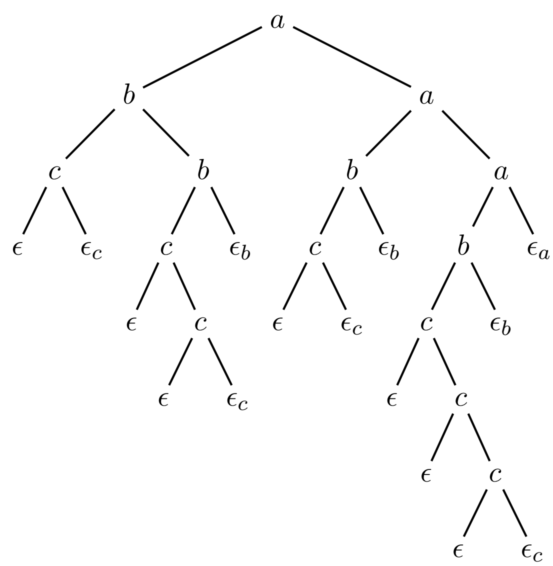
with $n + 1$ occurrences of a. Otherwise, let a be the maximal letter in Σ_1. Then we can splice w into $w_0aw_1a\cdots aw_n$, where a does not occur in the w_i. We represent w by

$$a(t_n, a(t_{n-1}, \ldots a(w_0, \epsilon_a)))$$

where the term t_i represents the word w_i over the alphabet consisting of Σ_1 minus the letter a. Note the inversion of the order.

Example

Let $\Sigma_1 = \{a, b, c\}$ with $a > b > c$. The word $ccaacb$ is represented by



Theorem 68 *[NR00] The relational structure $\mathcal{A}_{\mathrm{lpo}}$ is tree-automatic in case of a monadic signature and a total precedence.*

Proof: The tree signature Γ contains a constant ϵ, and for every letter c of the alphabet a constant ϵ_c and a binary symbol c.

The regular language $L_\delta \subseteq T(\Gamma)$ of representing trees is given by the following tree automaton:

- It has a state p_c for every letter $c \in \Sigma_1$
- If a is the maximal letter of Σ_1 then p_a is the only accepting state
- The transition rules are

$$\begin{array}{lllllll} \epsilon_c & \rightarrow & p_c & c(\epsilon, d_c) & \rightarrow & p_c & \text{if } c \text{ is the minimal letter of } \Sigma_1 \\ \epsilon_c & \rightarrow & p_c & c(p_b, p_c) & \rightarrow & p_c & \text{if } b \text{ is the predecessor of the letter } c \end{array}$$

The interpretation function $\nu \colon L_\delta \rightarrow \Sigma_1^*$ is given by

- $\nu(\epsilon) = \epsilon$
- $\nu(c(t_1, \epsilon_c)) = \nu(t_1)$
- $\nu(c(t_1, t_2)) = \nu(t_2) \cdot c \cdot \nu(t_1)$ if $t_2 \neq \epsilon_c$

The automaton for the equality relation $x_1 = x_2$ is defined as follows:

- It has one state $q_=$, which is also accepting
- The transition rules are

$$\begin{array}{llll} [c, c] & \rightarrow & q_= & \text{if } c \text{ is a constant} \\ [c, c](q_=, q_=) & \rightarrow & p_= & \text{if } c \text{ is binary} \end{array}$$

The automaton for the relation $x_1 \succ_{\text{lpo}} x_2$ is defined as follows:

- It has states $q_=$ and $q_\top$, and $q_>$.
- The only accepting state is $q_>$.
- The transitions are those of the automaton for $x_1 = x_2$, plus the rules stating that *every* tree is accepted in state $q_\top$, plus for every letter c:

$$\begin{array}{lllllllll} [\epsilon_c, \Box] & \to & q_> & [c, \Box] & \to & q_> & & & \\ [c, \epsilon_c](q_\top, q_>) & \to & q_> & [c, c](q_\top, q_>) & \to & q_> & [c, c](q_>, q_=) & \to & q_> \end{array}$$

The automaton for the relation $x_1 = a \cdot x_2$ is defined as follows:

- It has states $q_\perp$ and q_a
- The only accepting state is q_a
- The transitions are as follows, where c is any letter:

$$\begin{array}{lllllllll} [\epsilon, \Box] & \to & q_\perp & [c, \Box](q_\perp, [\epsilon_c, \Box]) & \to & q_\perp & & & \\ [a, \epsilon_a](q_\perp, [\epsilon_a, \Box]) & \to & q_a & [c, c](q_=, q_a) & \to & q_a & [c, c](p_a, [\epsilon_c, \epsilon_c]) & \to & p_a \end{array}$$

$\Box$

9.3 Rabin Tree Automata and Logic

Definition 15 *Let Σ be a signature consisting of binary function symbols only. A* Rabin Tree Automaton *A is given by (Q, q_0, Ω, Δ) where*

- *Q is a finite set of* states*.*
- *$\Omega \subseteq 2^Q$ is a set of* accepting sets of states*.*
- *$q_0 \in Q$ is the* inital state
- *Δ is a set of rewrite rules of the form $q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n))$.*

Usually, Rabin Tree automata are restricted to signatures that contain only binary function symbols. Note that the rewrite rules go in the "opposite" direction of rewrite rules of classical tree automata (Section 3.5), that is they are *top-down* instead of *bottom-up*. This is of course a necessity for automata that are to be run on infinite trees.

Definition 16 *A* run *of a Rabin automaton A on an infinite Σ-tree t is a mapping $r \colon \{0, \ldots, n\}^* \to Q$ with*

- *$r(\epsilon) = q_0$*
- *$r(w)(t(w)(x_1, \ldots, x_n)) \to t(w)(r(w1)(x_1), \ldots, r(wn)(x_n)) \in \Delta$ for all $w \in D_t$.*

A path *in a run r is an infinite sequence $\pi \in (\{1,\ldots,n\})^\omega$. For a path π in a run r we denote $In(r \mid \pi) \subseteq Q$ the set of states that occur infinitely often on the path π in r.*

The Rabin automaton A accepts *a run r iff for all paths π in r we have that $In(r \mid \pi) \in \Omega$. The automaton A accepts a tree t iff A accepts some run of A on t.*

The acceptance condition used here is the acceptance condition given by Muller which is equivalent to the slightly more complicated but equivalent original acceptance condition by Rabin.

As an example consider a signature $\Sigma = \{a, b\}$. The set T_0 of all trees where every path has only finitely many occurrences of a is recognized by an automaton that just memorizes in its state the symbol that he has seen at the father position:

$$\begin{array}{ll} q_a(a(x_1,x_2)) \to a(q_a(x_1), q_a(x_2)) & q_a(b(x_1,x_2)) \to b(q_b(x_1), q_b(x_2)) \\ q_b(a(x_1,x_2)) \to a(q_a(x_1), q_a(x_2)) & q_b(b(x_1,x_2)) \to b(q_b(x_1), q_b(x_2)) \end{array}$$

The initial state is q_a. On every path we must have only a finite number of times state q_a, that is the set of accepting sets is

$$\Omega = \{\{q_b\}\}$$

The complement of T_0, that is the set T_1 of all trees that have at least one path with infinitely many occurrences of a, is again Rabin recognizable. The automaton for T_1 guesses the path where an infinite number of a should occur. The state q_* indicates that the we are outside the path with an infinite number of a's, states q_a and q_b are obtained on the path and indicate the last symbol seen:

$$\begin{array}{ll} q_*(a(x_1,x_2)) \to a(q_*(x_1), q_*(x_2)) & q_*(b(x_1,x_2)) \to b(q_*(x_1), q_*(x_2)) \\ q_a(a(x_1,x_2)) \to a(q_a(x_1), q_*(x_2)) & q_a(b(x_1,x_2)) \to b(q_b(x_1), q_*(x_2)) \\ q_a(a(x_1,x_2)) \to a(q_*(x_1), q_a(x_2)) & q_a(b(x_1,x_2)) \to b(q_*(x_1), q_b(x_2)) \\ q_b(a(x_1,x_2)) \to a(q_a(x_1), q_*(x_2)) & q_b(b(x_1,x_2)) \to b(q_b(x_1), q_*(x_2)) \\ q_b(a(x_1,x_2)) \to a(q_*(x_1), q_a(x_2)) & q_b(b(x_1,x_2)) \to b(q_*(x_1), q_b(x_2)) \end{array}$$

In every run we have exactly one path that is labeled with q_a and q_b. Everywhere else the run is labeled with q_*. Hence, the set of accepting sets is

$$\Omega = \{\{q_*\}, \{q_a\}, \{q_a, q_b\}\}$$

From the latter example one can see that non-determinism is essential for Rabin tree automata. In fact one can easily show that deterministic Rabin tree automata (that is automata with the property that if $q(f(x_1,x_2)) \to f(q_1(x_1), q_2(x_2)) \in \Delta$ and $q(f(x_1,x_2)) \to f(q_1'(x_1), q_2'(x_2)) \in \Delta$ then $q_1 = q_1'$ and $q_2 = q_2'$) are less expressive than general Rabin automata (this is already the case when one compares top-down finite tree automata and bottom-up finite tree automata):

Lemma 69 *The set T_1 is not recognizable by a deterministic Rabin automaton.*

Proof: Suppose that some deterministic Rabin automaton accepts T_1. Let t_a be some tree that has a path with infinitely many occurrences of a, and let t_b be a tree that has b everywhere. Suppose that the transition for the initial state q_0 and symbol a is q_0, a, q_1, q_2. Since $a(t_a, t_b) \in T_1$ the tree t_b must be accepted from q_2, and since $a(t_b, t_a) \in T_1$ the tree t_b must also be accepted from state q_1. Hence $a(t_b, t_b) \notin T_1$ is accepted as well. □

The crucial theorem about Rabin Tree automata is

Theorem 70 *([Rab69])*

- *The class of Rabin-recognizable sets of infinite trees is closed under complementation.*
- *Emptiness of Rabin automata is decidable*

The proof requires quite heavy mathematical machinery. Modern proofs of this theorem (see, for instance, [Tho97] for an excellent presentation of a compact proof) use techniques from Game Theory.

9.3.1 Example: S2S

Since Rabin tree automata have the same nice properties than finite tree automata we obtain

Theorem 71 *([Rab69]) The theory of $\mathcal{A}_{S2S}$ is decidable.*

9.3.2 Related Results

As explained in Section 9.1 an algorithm based on the a reduction to automata yields a priori a non-elementary complexity since the size of the automaton may grow exponentially at every negation. The following result asserts that we can not do better for $S2S$ and $WS2S$:

Theorem 72 *([SM73]) The decision problems of the theories of $\mathcal{A}_{WS2S}$ and of $\mathcal{A}_{S2S}$ have non-elementary complexity.*

Another indication of the strength of Rabin's theorem, besides the difficulty of its proof, is the fact that it is very hard to find extensions of the structure that still have a decidable theory. For instance, the following two variants of $\mathcal{A}_{S2S}$ have undecidable first-order theories:

- the structure where $0(v)$ is interpreted as $0v$ and $1(v)$ is interpreted as $v1$;
- the extension of $\mathcal{A}_{S2S}$ by the predicate "x and y have the same length".

We will come back to both of these two undecidabilty results in Section 12.1.5.

Chapter 10

Semantic Methods: Automata for Monadic Structures

A *monadic* structure is a structure that contains only unary predicate symbols (and hence does not contain the equality relation) and no function sysmbols.

First-order logic without function symbols and with only unary predicate symbols is called *monadic first-order logic*. This logic is quite weak.

Theorem 73 *[Löw15, Ack54] Validity of formulas of monadic first-order logic is decidable.*

The result of Löwenheim and Ackermann is in fact stronger than our Theorem 73, it applies even to second order formula (allowing for quantification over monadic predicate variables) and allows for the equality predicate.

What makes predicate logic interesting is the fact that it allows us to build new complex relations out of basic relations (the interpretation of the perdicate symbols). The capacity of predicate logic to build complex relations out of simple ones is very restricted in the monadic case, as the following lemma shows:

Lemma 74 *Any monadic first-order formula is equivalent to a Boolean combination of formulas of the form*

$$\exists x(P_1(x) \wedge \ldots \wedge P_n(x) \wedge \neg Q_1(x) \ldots \wedge \neg Q_m(x)$$

where the P_i, Q_i are pairwise different predicate symbols.

Exercice 16
Show Lemma 74.

Exercice 17
Show Theorem 73, using Lemma 74.

It follows from Lemma 74 that any monadic structure that is $\mathcal{C}$-automatic for some class $\mathcal{C}$ of automata is decidable provided that $\mathcal{C}$ is closed under Boolean operations, and that emptyness of $\mathcal{C}$-automata is decidable. In this case it is not required that the classs $\mathcal{C}$ be closed under cylindrification or projection.

10.0.3 Example: The Theory of Reducibility

Reduction automata have been defined in [CCD93] as tree automata with equality tests [Mon81] which have a bound on the number of equality tests that can be performed on any path of a run. The emptiness problem of reduction automata is decidable [CCD93, DCC95], see also [CCC$^+$94] for a generalization.

Definition 17 *Given a finite signature* Σ, *a* reduction automaton with equality constraints (short: RAEC) $\mathcal{A}$ *is given by* (Q, F, Δ, ω) *where*

- Q *is a finite set of* states.
- $F \subseteq Q$ *is called the set of* accepting states.
- Δ *is a set of constrained rewrite rules of the form* $f(q_1, \ldots, q_n) \rightarrow^c q$ *with* $f \in \Sigma_n$, $q_1, \ldots, q_n, q \in Q$ *and where* c *is a conjunction of tests of the form* $\pi = \mu$.
- ω *is function from* Q *to some partially ordered set* $(S, <)$ *such that, for any rule* $f(q_1, \ldots, q_n) \rightarrow^c q$ *we have that* $\omega(q) \geq \omega(q_1), \ldots, \omega(q_n)$, *and even* $\omega(q) > \omega(q_1), \ldots, \omega(q_n)$ *in case* $c \neq \texttt{true}$.

The automaton A accepts *a tree* t *iff* $t \rightarrow^* q \in F$ *by the transition rules* Δ. *The* language $\mathcal{L}(A)$ *is the set of all trees accepted by* A.

Definition 18 *A tree* t satisfies *the test* $\pi = \mu$ *iff* $\pi, \mu \in O(t)$ *and* $t[\pi] = t[\mu]$, *this extends in the canonical way to conjunctions of tests.*

The rewriting relation of a CRA $A = (\Gamma, n, Q, F, \Delta)$ *is the smallest binary relation* $\rightarrow_A \subseteq T(\Sigma_n^\square) \times Q$ *with*

- $a \rightarrow_A q$ *if* $a \rightarrow q \in \Delta$
- $f(t_1, \ldots, t_n) \rightarrow_A q$ *if* $f(q_1, \ldots, q_n) \rightarrow^c q \in \Delta$, $f(t_1, \ldots, t_n)$ *satisfies* c, $t_1 \rightarrow_A q_1 \ldots, t_n \rightarrow_A q_n$.

The automaton A accepts *a tree* t *if* $t \rightarrow_A q$ *for some accepting state* $q \in F$. *The set* L_A recognized *by* A *is the set of all trees accepted by* A.

Emptyness of the language accepted by a tree automaton with arbitrary equality constraints is undecidable [Mon81]. The essential restriction here lies in the function ω which ensures that there is an a priori bound (not depednding on the input tree) on the number of tests that can be performed on any path of any input tree. This restriction makes it possible to show

Theorem 75 ([CCD93, DCC95]) *The class of languages recognized by deterministic and complete RAEC is effectively closed under Boolean operations, and emptiness of the language recognized by a deterministic and complete RAEC is decidable.*

The main interest of RAEC is that one can easily define, for any term $l \in T(\Sigma, X)$, a complete and deterministic RAEC that recognizes the language of ground terms that are reduced by l. Equality constraints are used here since l may be non-linear (that is, contain several occurrences of the same variable). If the term l is linear then classical tree automata are sufficient.

Example

Let $\Sigma = \{\epsilon, a, f\}$ where ϵ is constant, a is unary, and f binary. The set of trees reduced by $f(x, a(x))$ is recognized by the RAEC with $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, and $\omega(q_i) = i$ with the natural order:

$$\begin{array}{lll}
\epsilon \to q_0 & f(q_0, q_1) \to^{1=21} q_2 & a(q_2) \to q_2 \\
a(q_0) \to q_1 & f(q_1, q_1) \to^{1=21} q_2 & f(q_2, _) \to q_2 \\
a(q_1) \to q_2 & & f(_, q_2) \to q_2
\end{array}$$

Hence, it follows with Theorem 75 that the so-called theory of *reducibility* is decidable:

Theorem 76 ([CCD93, DCC95]) *Let Σ be a finite signature, $t_1, \ldots, t_n \in T(\Sigma)$ and $r_1, \ldots, r_m \in T(\Sigma, X)$. The first-order theory of the structure with universe $T(\Sigma)$, unary predicates $x = t_i$ (interpreted as equality of the argument to t_i) and red_{r_i} (interpreted as the argument being reducible by r_i) is decidable.*

Note that RAEC are, by construction, not closed under cylindrification, and as we have seen above this is not a problem for decidability of the first-order theory of reducibility since this theory is monadic. Is it possible to close the class of CRAEC under cylindrification and retain the nice properties given in Theorem 75? The answer to that question is negative, as we will show in Theorem 77 below. First we will define the new class of automata in Definition 19. This definition does not contain the function ω of Definition 17; in fact this is not necessary since our undecidability result holds even in case the only constraint checking occurs only at the very last step of the automaton.

Definition 19 *Given a finite signature Σ, a* component-wise reduction automaton with equality constraints (short: CRAEC) *is given by (n, Q, F, Δ) where*

- *$n \geq 1$,*
- *Q is a finite set of states,*
- *$F \subseteq Q$ is the set of accepting states,*
- *Δ is a set of transition rules of one of the following forms:*
  - *$(\perp \ldots \perp) \to q$, $q \in Q$.*
  - *$f(q_1, q_2) \to^c q$ where c is a conjunction of tests of the form $\pi.i = \mu.j$ with $1 \leq i, j \leq n$, $\pi, \mu \in \{1, 2\}^*$, $f \in \Gamma^{(n)} - (\perp \ldots \perp)$, $q, q_1, q_2 \in Q$.*

Definition 20 (Acceptance by a CRA) *A tree t* satisfies *the test $\pi.i = \mu.j$ iff $(t[\pi]).i = (t[\mu]).j$, this extends in the canonical way to conjunctions of tests.*

The rewriting relation of a CRA $A = (\Gamma, n, Q, F, \Delta)$ is the smallest binary relation $\to_A \subseteq T(\Gamma^{(n)}) \times Q$ with

- *$a \to_A q$ if $a \to q \in \Delta$*
- *$f(t_1, t_2) \to_A q$ if $f(q_1, q_2) \to^c q \in \Delta$, $f(t_1, t_2)$ satisfies c, $t_1 \to_A q_1$ and $t_2 \to_A q_2$.*

The automaton A accepts *a tree t if $t \to_A q$ for some accepting state $q \in F$. The set L_A* recognized *by A is the set of all trees accepted by A.*

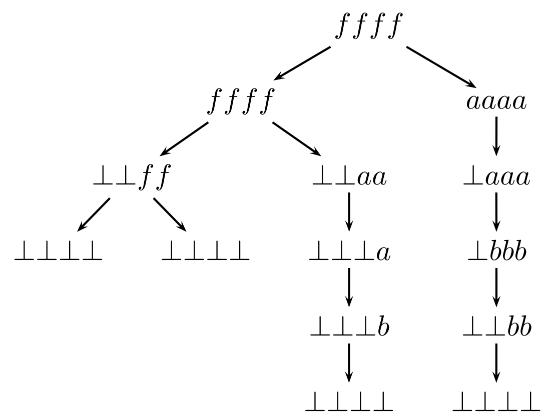


Figure 10.1: Example of the representation of the solution $((\epsilon, \epsilon), (a, aab), (aabb, aabb))$ of the PCP instance $((a, aab), (abb, b))$.

Theorem 77 ([Tre01, Su02]) *The emptiness problem for CRAEC (even when restricted to the class of CRAEC that perform one single test, where this test is at the root of the tree) is undecidable.*

Proof: We reduce the Post Correspondence Problem to the emptiness problem of CRA. Let $P = ((p_i, q_i))_{i=1,\dots,m}$ be an instance of PCP. We use the signature $\Sigma = \{\perp, a, b, f\}$ where $\perp$ is a constant, a and b are unary, and f is binary.

A word $w \in \{a, b\}^*$ is encoded as a tree $c(w) \in T(\Sigma)$ as follows:

$$\begin{array}{rcl} c(\epsilon) & := & \perp \\ c(aw) & := & a(c(w)) \\ c(bw) & := & b(c(w)) \end{array}$$

A quadruple of words $(w_1, w_2, w_3, w_4) \in (\{a, b\}^*)^4$ is encoded as the tree $t = c(w_1, w_2, w_3, w_4)$ such that $t.i = c(w_i)$ (see Proposition 9.2). For instance the encoding of the tuple $(a, aab, aabb, aabb)$ is the right subtree of the tree depicted in Figure 10.1.

We define the signature of our automaton as $\{f, a, b\}^{(4)}$. The automaton L_P is constructed in several steps:

For any word $v \in \Gamma^*$, let $A_v^{i,j}$ for $i \neq j$ be the automaton that accepts all encodings of tuples (w_1, w_2, w_3, w_4) where $w_j = w_i v$. If $v = v_1 \cdots v_n$ then the rules of the automaton are (shown here for $i = 2$ and $j = 4$), where α stands for any symbol from $\{a, b\}$ and where every occurrence of $\_$ stands for any symbol from $\{a, b, \perp\}$ (different occurrences of $\_$ may denote different symbols):

$$\begin{array}{rcl} (\_\perp\_\perp) & \to & q_n \\ (\_\perp\_v_i)(q_n, q_i) & \to & q_{i-1} \\ (\_\alpha\_\alpha)(q_n, q_0) & \to & q_0 \end{array}$$

The only accepting state is q_0.

From the boolean closure of (classical) tree automata we get an automaton A_1 which accepts all encodings of tuples of words of the form (v, w, vp_i, wq_i) with $1 \leq i \leq n$, that is such that the last two words of the quadruple are obtained from the first two words by application of one step of the given instance of PCP. Let Q_1 be the set of accepting states of A_1.

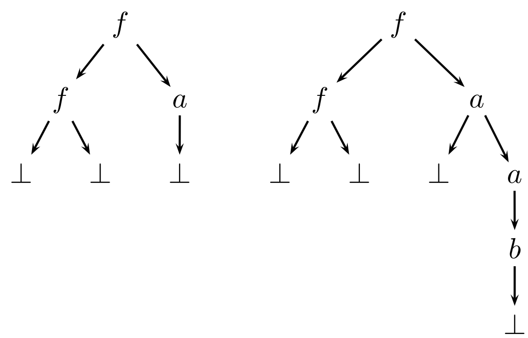


Figure 10.2: The first and second components of the tree of Figure 10.1, which are identical to the third and forth component of its left subtree of the tree of Figure 10.1.

The next step is to construct an automaton which accepts all encodings of tuples (w_1, w_2, w_3, w_4) where $w_3 = w_4$:

$$\begin{array}{rcl} (\_\_\perp\perp) & \to & q \\ (\_\_\alpha\alpha) & \to & q \end{array}$$

where q is the only accepting state. Let A_2 be the automaton which recognizes its intersection with the automaton A_1, and Q_2 its set of accepting states.

Note that these first two steps used only classical tree automata. The third step is now to build the final automaton which uses the automata A_1 and A_2 as sub-automata (we hence assume their state-spaces to be disjoint) to accept a solution sequence of P. This is where equality tests come in.

$$\begin{array}{rcll} (\perp\perp\perp\perp) & \to & p_\perp & \\ (\perp\perp ff)(p_\perp, q_\perp) & \to & p_0 & \\ (ffff)(p_0, q) & \to & p_0 & q \in Q_1 \\ (ffff)(p_0, q) & \to^{\epsilon.1=1.3 \wedge \epsilon.2=1.4} & p_f & q \in Q_2 \end{array}$$

where p_f is the only accepting state. An example of the representation of the solution of an instance of PCP as accepted by this automaton is given in Figure 10.1. □

Note that the proof can be easily adapted to trees with only two components and to the class of automata that perform only one *atomic* equality test by coding the first and second, respectively the third and forth component into one component.

Chapter 11

A Comparison of Methods for Deciding First-Order Theories

These lecture notes have presented two important families of methods for deciding first-order theories: syntactic and semantic methods. What makes the distinction between these two? After all, syntactic objects play an important role in both of them (formulas, and the syntactic objects that are recognized by the various classes of automata), and both use semantic arguments (talking about properties of the structure) in order to show their correctness.

11.1 Syntactic Methods

When designing syntactic methods, in particular methods based on simplification of formulas, our primary objects of discourse are formulas, that is syntactic objects. It is still true that we have to use semantic arguments to show the correctness of the transformations, but this is only part of the proof obligations. Termination and completeness are based on purely syntactic arguments.

A great advantage of syntactic methods lies in the fact that one can profit from the techniques developed the last two decades in the field of term rewrite systems. First-order formulas are not really first-order terms, for several reasons:

- Some algebraic properties of logical connectives are implicit, for instance logical conjunction and disjunction are associative and commutative, and these properties are used without saying when designing transformation rules. This can be modeled in term rewrite theory by considering terms modulo some equational theory.
- Quantifiers are beyond a first-order term modelisation, in particular issues like renaming of bound variables.

Classical term rewrite techniques are still useful when applied with care. Quantifiers, for instance, do often not play a central role when designing simplification rules for basic formula. The most useful thing from term rewrite theory here is probably the "toolbox" of termination proofs.

It is instructive to investigate how syntactic methods have to be adapted when we change the underlying structure. We have seen an example of this in the chapter on quantifier elimination when we passed from finite feature trees to infinite feature trees. In case of FT the change was minimal, the only difference in the quantifier elimination procedure for FT over finite and infinite trees consists in the presence or absence of an occur check. Things are arguably more complicated in case of CFT, but still syntactic methods are more robust to "small" changes to the structure since for most simplification rules we just have to check that they still describe equivalence transformations, and modify them accordingly if they don't. The completeness and termination arguments are not affected by he shift to a different structure.

There is additional perk of syntactic methods described by a finite set of transformation rules: they allow us to derive a complete axiomatization of the theory of the structure. The axiomatization can be obtained from the transformation system simply by turning any transformation rule $\frac{\phi}{\psi}$ into a logical equivalence $\tilde{\forall}\phi \leftrightarrow \psi$. In some sense, the transformation system is just a strategy for applying these equivalence. The axiomatization is not necessarily finite (in case the set of rules is not finite but given by a set of rule schemata), but it is "as regular" as the set of transformation rules (which means at the very least that is a decidable set).

To close this section we would like to mention that there is at least one example of a syntactic method based on automata. In [MNT98], a method for deciding entailment between $\mathcal{A}_{FT\leq}$ constraints was given that relies on a characterization of of the semantics of constraints by recognizable sets of path constraints. In this case, the entailment problem between constraints reduces to the inclusion problem between recognizable languages.

11.2 Semantic Methods

When designing semantic methods, in particular methods based on automata, our primary objects of discourse are (tuples of) values taken form the semantic structure, and how these can be described by the mechanism at hand (automata). It is still true that we have to use syntactic arguments since the syntax of formulas guides the translation process, but this is generic, and the core of the argument consists in showing that the used class of automata has the necessary closure properties, and that the essential properties are decidable.

Trying to show that a certain structure is automatic may be too strong. The reason for this is that, for given a L_δ and ν, any predicate that is represented by a recognizable set is automatic. If one shows that a certain structure is automatic, one obtains in reality a much stronger result: The extension of the structure by all (countable many !) predicates that are represented by some recognizable language is automatic, and hence its first-order theory is decidable. These additional predicates are often not expressible in the original structure, and this may make proving the fact that the structure is automatic much harder than mere decidability.

Another reason why showing a structure automatic is stronger than just showing that its first-order theory is decidable has been given in Theorem 60: one obtains in fact decidability of a (slight) extension of first-order logic in case the representation function is one-one.

An advantage of the automata method is that all predicates in the structure can be considered in isolation, in contrast to syntactic methods where one has to study the properties of solved forms that in general combine all possible predicates allowed in the structure (except when one can eliminate a certain predicate completely from solved forms, as for instance the equality predicate or some predicate that can be defined in terms of other predicates). Of course one still has to find some L_δ and ν that "work" for all the predicates of the structure. A consequence of this is that combinations of structures are for free provided one has the same L_δ and ν: If (A, P_1) and (A, P_2) are automatic with the same L_δ and ν then $(A, P_1 \cup P_2)$ is also automatic. This may be useful (see, for instance, Subsection 9.2.2), but on the other hand it shows again why the property of being automatic may be too strong when one is in fact only interested in decidability of first-order theories: The above combination property does not hold when one replaces "being automatic" by " having a decidable first-order theory": There are structures (A, P_1) and (A, P_2) with decidable first-order theories such that the theory of $(A, P_1 \cup P_2)$ is not decidable, as it is for instance the case for Presburger arihmetic and Skolem arihmetic. It is true that both structures are automatic (Subsection 9.1.2 and Exercice 14), but using different classes of automata and completely different representation fucntions that do not combine.

Another example is obtained from the theory of finite words. Lets consider the following three structures $\mathcal{A}_{Wp}$, $\mathcal{A}_{Ws}$ and $\mathcal{A}_{Wps}$:

- all three structures have an infinite set of constants, namely all finite words that can be formed over the alphabet $\{a, b\}$;
- there are no other functions symbols;
- all three structures have the same universe, namely $\{a, b\}^*$, and every constant is interpreted as itself;
- $\mathcal{A}_{Wp}$ has as only predicate symbol $x \prec_{pre} y$, interpreted as x is a prefix of y; $\mathcal{A}_{Ws}$ has as only predicate symbol $x \prec_{suf} y$, interpreted as x is a suffix of y, and $\mathcal{A}_{Wps}$ has both predicate symbols (with the same interpretation).

It is easy to show that $\mathcal{A}_{Wp}$ is automatic, by representing a word w as the word w itself. It is equally simple to show that $\mathcal{A}_{Ws}$ is automatic, but this requires that we change the representation and represent a word w by its mirror $\bar{w}$ (that is by inversing its order). The combination of the two structures, $\mathcal{A}_{Wps}$, has however an undecidable first-order theory since it allows to express that y is a factor of x:

$$\exists z(z \prec_{pre} y \wedge x \prec_{suf} z)$$

and this all it takes to make the undecidability proof of Theorem 79 work.

In the section of syntactic methods we have argued that these are quite robust against "small" changes to he underlying structure. This is not the case for semantic methods, as we have seen the move form finite trees to infinite trees necessitates a move from classical tree automata to Rabin tree automata, which is quite dramatic since the fundamental theorems about Rabin tree automata rely on deep game-theoretic arguments.

How would semantic and syntactic methods compare when modifying the syntax of formulas? This is more difficult to answer as we do not have many examples of such a modification, since the language of first-order logic is not open to much essential modification. Still we got some taste of it when talking about the $\exists^\infty$ quantifier in the chapter on automatic structures: all we needed to do in order to extend our results on automatic structures to his new quantifier was to show an additional closure property of recognizable languages, which was in some sense "orthogonal" to the preceding closure properties. In case of syntactic methods, on the other hand, an essential modification of the syntax of formulas is more likely to necessitate transversal changes in the whole simplification procedure.

Chapter 12

Undecidable Theories

12.1 Coding the Post Correspondence Problem via Sets

12.1.1 Basic Concepts

We present a method for constructing undecidability proofs of first-order theories of concrete structures [Tre92].

Our method for showing undecidability of the theory of some structure $\mathcal{A}$ consists in constructing, for a given instance P of the Post Correspondence Problem (see Section 3.7), a formula $\texttt{solvable}_P$ that is valid in $\mathcal{A}$ iff P is solvable. For a given structure $\mathcal{A}$ we will have to construct formulas expressing properties of words and sets of pairs of words. Then, the general scheme for the formula $\texttt{solvable}_P$ is

$$\exists z, x\ ((x, x)\ \texttt{in}\ z \wedge \neg\texttt{epsilon}(x) \wedge \texttt{construction}_P(z))$$

and the formula $\texttt{construction}_P(z)$ is

$$\begin{aligned}
&\forall x_l, x_r\ ((x_l, x_r)\ \texttt{in}\ z \rightarrow \\
&\qquad (\texttt{epsilon}(x_l) \wedge \texttt{epsilon}(x_r)) \vee \\
&\qquad \exists y_l, y_r\ ((y_l, y_r)\ \texttt{in}\ z \wedge \bigvee_{(p,q)\in P} (\texttt{append}_p(y_l, x_l) \wedge \texttt{append}_q(y_r, x_r))))
\end{aligned}$$

The intended meaning of a formula $\texttt{append}_w(x, y)$ is that y represents the word represented by x with the word w appended at the end. This can obviously be expressed by using a more primitive relation which denotes only appedning one letter:

$$\texttt{append}_{(c_1\cdots c_n)}(x, y) = \exists z_1, \ldots, z_{n-1}(\texttt{append}_{c_1}(x, z_1) \wedge \bigwedge_{i=2}^{n-1} \texttt{append}_{c_i}(z_{i-1}, z_i) \wedge \texttt{append}_{c_n}(z_{n-1}, y)$$

Theorem 78 *([Tre92]) Let $\mathcal{A}$ be a structure and functions*

$$\begin{aligned}
&\phi\colon \{a, b\}^* \rightarrow \mathcal{A} && \textit{one-to-one} \\
&\psi\colon \textit{FinSets}(\{a, b\}^* \times \{a, b\}^*) \rightarrow \mathcal{A}
\end{aligned}$$

Let there be formulas in the language of $\mathcal{A}$ $\texttt{epsilon}(x)$, (x, y) $\texttt{in}$ z and for each $c \in \{a, b\}$ a formula $\texttt{append}_c(x, y)$ such that

1. $\mathcal{A} \models \textit{epsilon}[c]$ *iff* $c = \phi(\epsilon)$

2. $\mathcal{A} \models \textit{append}_c[\phi(v), d]$ *iff* $d = \phi(vc)$

3. $\mathcal{A} \models [c, d]\ \textit{in}\ [\psi(m)]$ *iff there exists* $(v, w) \in m$ *with* $c = \phi(v)$ *and* $d = \phi(w)$

4. The relation binary relation $\sqsubset$ *on* $\mathcal{A}$*, defined by* $a_1 \sqsubset a_2$ *if* $\mathcal{A} \models \textit{append}_c[a_1, a_2]$ *for some* $c \in \{a, b\}$*, is well-founded.*

Then the first-order theory of $\mathcal{A}$ *is undecidable.*

Proof: We show that any instance P of PCP is solvable iff $\mathcal{A} \models \texttt{solvable}_P$.

Let $i_1, \ldots, i_m$ be a solution of P. Let

$$\begin{aligned} d &:= \phi(p_{i_1} \cdots p_{i_m}) \\ s &:= \psi(\{(p_{i_1} \cdots p_{i_j}, q_{i_1} \cdots q_{i_j}) \mid 0 \leq j \leq n\}) \end{aligned}$$

The formula $\texttt{solvable}_P$ is obviously true in $\mathcal{A}$ by choosing the values d for x and s for z.

Let $s \in \mathcal{A}$ such that $\mathcal{A} \models \texttt{construction}_P[z]$. By the last condition of the theorem the relation $\sqsubset$ is well-founded. Hence, we can prove by well-founded induction that for every pair (c, d): if $\mathcal{A} \models [c, d]\ \texttt{in}\ [s]$ then there is a P-constructable pair of words (v, w) with $c = \phi(v)$ and $d = \phi(w)$.

Hence, and since ϕ is one-to-one, if $\mathcal{A} \models \texttt{solvable}_P$ then there is non-empty word v such that (v, v) is P-constructable. □

When proving undecidability of a theory one is of course interested to formulate a theorem as sharp as possible, that is to find an undecidable fragment of the theory as simple as possible. One common way to define fragments of a theory is to classify formulas by the number of quantifier alternations of their prenex normal form: The Σ_n*-fragment* contains all formulas whose prenex normal form starts with $\exists$ and has $n - 1$ quantifier alternations, and the Π_n*-fragment* contains all formulas whose prenex normal form starts with Π and has $n - 1$ quantifier alternations.

With the method given in this section we can at best show undecidability of the Σ_3-fragment of the theory. An improvement of the method using encodings of sequences instead of sets, and that can yield undecidability results for the Σ_2-fragment of a theory, is given in [Tre92].

A variant of the method applies to the case where the relation $\sqsubset$ (as defined in the above proof) is not well-foundeed in general but is well-founded at least on the representation of words. In this case we have to provide a formula $\texttt{finite}(x)$ such that in the structure $\mathcal{A}$

1. $\mathcal{A} \models \texttt{finite}(\phi(v))$ for all words $v \in \{a, b\}^*$

2. there is no infinite sequence of letters $d_1, d_2, \ldots$ and sequence $c_1, c_2, \ldots$ of elements of $\mathcal{A}$ with

$$\mathcal{A} \models \texttt{finite}[c_1], \mathcal{A} \models \texttt{append}_{v_1}[c_2, c_1], \mathcal{A} \models \texttt{append}_{v_2}[c_3, c_2], \ldots$$

Then we can change the definition of scheme for the formula $\texttt{solvable}_P$ into

$$\exists z, x\ ((x, x)\ \texttt{in}\ z \wedge \texttt{finite}(x) \wedge \neg\texttt{epsilon}(x) \wedge \texttt{construction}_P(z))$$

and the proof for this variant is exactly as the proof of Theorem 78.

12.1.2 Example: The Theory of Words

Theorem 79 *([Qui46]) The first-order theory of words is undecidable.*

We can show this classical result now easily by using Theorem 78.

Proof: For simplicity we assume that the alphabet contains four letters a, b, c, d. The proof for a binary alphabet $\{a, b\}$ is obtained by encoding letters over $\{a, b, c, d\}$ as words over $\{a, b\}$, for instance $a \mapsto aba, b \mapsto abba, c \mapsto abbba, d \mapsto abbbba$.

$$\begin{array}{rcl}
\phi(v) & := & v \\
\texttt{epsilon}(x) & := & x = \epsilon \\
\texttt{append}_v(x, y) & := & y = xv \\
\psi(\{(v_1, w_1), \ldots, (v_n, w_n)\}) & := & cv_1dw_1c \cdots cv_ndw_nc \\
(x, y)\ \texttt{in}\ z & := & \exists r, s\ z = rcxdycs
\end{array}$$

□

Corollary 80 *The positive first-order theory of words is undecidable in case of a finite alphabet containing at least two symbols.*

Proof: This follows immediately from Theorem 79 and the fact that inequations can be expressed equivalently by positive formulas in case of a finite alphabet (Proposition 8). □

Related Results

The undecidability result of Theorem 79 can be sharpenend to

Theorem 81 *([Dur95a]) It is undecidable whether a formula of the form*

$$\exists x\, \forall y_1, y_2, y_3\, e$$

where e is an equation, is valid in $\mathcal{A}_W$.

We have the following results about decidable special cases:

Theorem 82 *The theory of words over a unary alphabet is decidable.*

Proof: The theory of the structure of words over a unary alphabet reduces to the theory of Presburger Arithmetic (where ϵ is mapped to 0 and concatenation is mapped to $+$) and hence is decidable by Theorem 31. □

Recall that the Σ_1-fragment of $\mathcal{A}_W$ is decidable in case of a finite alphabet (Corollary 52).

See Section 8.2 for more on decidable special cases.

12.1.3 Example: Trees Modulo AC

Theorem 83 *([Tre92]) The Σ_3-fragment of the theory of ground terms modulo AC is undecidable in case of the signature $\Sigma_0 = \{\epsilon\}$, $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{f, +(AC)\}$.*

The proof is similar to the proof of Theorem 79.

Proof:

$$\begin{array}{rcl}
\phi(\epsilon) & := & \epsilon \\
\phi(wa) & := & a(\phi(w)) \\
\phi(wb) & := & b(\phi(w)) \\
\texttt{epsilon}(x) & := & x = \epsilon \\
\texttt{append}_a(x,y) & := & y = a(x) \\
\texttt{append}_b(x,y) & := & y = b(x)
\end{array}$$

Finally, we encode sets by

$$\begin{array}{rcl}
\psi(\{(v_1,w_1\},\ldots,(v_n,w_n)\}) & := & f(\phi(v_1),\phi(w_1)) + \ldots + f(\phi(v_n),\phi(w_n)) \\
(x,y) \texttt{ in } z & := & \exists r\; z = f(x,y) + r
\end{array}$$

□

Related Results

The undecidability result can be sharpend to simpler tree signatures:

Theorem 84 *([Tre92]) The Σ_3-fragment of the theory of ground terms modulo AC is undecidable in case of a signature containing at least one constant, one binary AC function symbol, and one non-constant function symbol (AC or not).*

Proof: This is a simple variation of the proof of Theorem 83. □

If the signature gets further simplified then the theory becomes decidable:

Theorem 85 *The theory of ground terms modulo AC is decidable in case of a signature consisiting of one binary AC function symbol and a finite set of constants.*

Proof: Let the set of constants be $\{c_1,\ldots,c_n\}$. In this case the structure has an n-dimensional interpretation (see Definition 2 and Proposition 3) in Presburger Arithmetic where n is the number of constants:

$$\begin{array}{rcl}
\delta(x^1,\ldots,x^n) & = & (x^1 \neq 0 \vee \ldots \vee x^n \neq 0) \\
\phi(p^1,\ldots,p^n) & = & \underbrace{c_1+\ldots+c_1}_{p^1} + \ldots + \underbrace{c_n+\ldots+c_n}_{p^n} \\
P_=(x,y) & = & \bigwedge_{i=1}^{i=n} x^i = y^i \\
P_{c_i}(x^1,\ldots,x^n) & = & (x^1 = 0 \wedge \ldots \wedge x^{i-1} = 0 \wedge x^i = 1 \wedge x^{i+1} = 0 \wedge \ldots \wedge x^n = 0) \\
P_+(x,y,z) & = & \bigwedge_{i=1}^{i=n} x^i + y^i = z^i
\end{array}$$

Informally this means that a term can now simply be seen as a multiset of constants, and can hence be simulated by a n-tuple giving the number of occurrences of any constant in the multiset. In this sense the AC operator is just component-wise addition. □

We have the following results about fragments characterized by the number of quantifier eliminations:

Theorem 86 *([Mar97]) The Σ_2-fragment of the theory of ground terms modulo AC is undecidable.*

Theorem 87 *([Com88]) The Σ_1-fragment of the theory of ground terms modulo AC is decidable.*

12.1.4 Example: Peano Arithmetic

The encoding of sets of pairs of words uses the β predicate introduced in [Göd31]. A proof of the theorem we restate below can also be found in textbooks on mathematical logic, for instance [End72].

$$\beta(x_1, x_2, l, x) := x = x_1 \% (1 + (l+1) * x_2)$$

where $y \% z$ stands for the rest of the division of y by z, that is

$$x = y\%z \qquad := \qquad \exists q\,(y = q * z + x \land x < z)$$

Theorem 88 *([Göd31]) For every sequence $x_0, \ldots, x_{n-1}$ of natural numbers there exist c_1, c_2 such that for all $0 \leq i < n$:*

$$\mathcal{A}_{Pe} \models \beta(c_1, c_2, i, e) \quad \Leftrightarrow \quad e = x_i$$

Proof: Let $l = \max(n, x_0, \ldots, x_{n-1})$. We define for $0 \leq i < n$

$$a_i = 1 + (i+1)l!$$

All numbers a_i are relatively prime: Suppose that some prime number p divides a_i and a_j for some $i \neq j$. Then p divides $a_i - a_j = (i-j)l!$, and hence p divides $l!$ since $|i - j| < n \leq l$. This is impossible since p divides $a_i = 1 + (i+1)l!$.

By the Chinese Remainder Theorem (Theorem 6) there is a number x such that $x \equiv x_i \pmod{a_i}$ for all $i = 0, \ldots, n-1$. Since $a_i > x_i$ we obtain that $x_i = x\%a_i$. Hence, $c_1 = x$ and $c_2 = l!$ does what is required. □

Theorem 89 *([Göd31]) The theory of Peano Arithmetic is undecidable.*

Proof: Words are encoded into numbers by reading a word as the binary representation for a number, where a stands for 0 and b stands for 1.

$$\begin{array}{rcl}
\phi(\epsilon) & := & 1 \\
\phi(wa) & := & 2\phi(w) \\
\phi(wb) & := & 2\phi(w) + 1 \\
\texttt{epsilon}(x) & := & x = 1
\end{array}$$

Note that we could have chosen any value for $\phi(\epsilon)$ except 0 (in case of $\phi(\epsilon) = 0$ the function ϕ would not be one-to-one). We finally define

$$\begin{array}{rcl}
\texttt{append}_a(x, y) & := & y = 2x \\
\texttt{append}_b(x, y) & := & y = 2x + 1
\end{array}$$

Sets of pairs of words are encoded as *triples* of natural numbers. As a consequence, the membership predicate has now the form (x, y) `in` (z_1, z_2, n).

$$\psi((u_i, v_i)_{i=0,\ldots,n-1})(c_1, c_2, n)$$

where c_1, c_2 are the values corresponding to the sequence

$$(\phi(u_0), \phi(v_0), \ldots, \phi(u_{n-1}), \phi(v_{n-1}))$$

whose existence is guaranteed by Theorem 88. Finally, we define

$$(x, y) \texttt{ in } (z_1, z_2, n) := \exists i \, (i < n \wedge \beta(z_1, z_2, 2i, x) \wedge \beta(z_1, z_2, 2i + 1, y)) \qquad \square$$

Related Results

Gödel's seminal paper [Göd31] contains in fact a much stronger results than Theorem 89. Gödel was interested in general proof systems that can prove sentences about aritmethic. In particular,

Theorem 90 *[Göd31] There is computable function that yields for every recursive set T of axioms with $\mathcal{A}_{Pe} \models T$ a set c such that neither $T \models c$ nor $T \models \neg c$.*

A much celebrated result is

Theorem 91 *[Mat70] It is undecidable whether a polynominal with integer coefficients has a root in $\mathbf{N}^n$.*

As an immediate consequence the Σ_1-fragment of the theory of $\mathcal{A}_{Pe}$ is undecidable.

We recall that the first-order theory of arithmetic is decidable in case one restricts oneself to formulas without multiplication (Presburger arithmetic) or to formulas without addition (Skolem arithmetic).

Theorem 92 ([Rob49]) *The theory of the structure $\mathcal{A}'_{Sk}$, that is Skolem Arithmetic enriched by the successor function, is undecidable.*

Proof: The structure $\mathcal{A}_{Pe}$ of Peano Arithmetic has a one-dimensional interpretation (see Definition 2) in $\mathcal{A}'_{Sk}$ as follows: δ is simply `True`, the interpretation function ϕ is the identity function, multiplication is interpreted as multiplication, and $x + y = z$ is interpreted as

$$\begin{aligned} P_+(x, y, z) \quad = \quad & (z = 0 \wedge x = 0 \wedge y = 0) \\ & \vee (z \neq 0 \wedge s(x * z) * s(y * z) = s(s(x * y) * z * z) \end{aligned}$$

Since the theory of $\mathcal{A}_{Pe}$ is undecidable (Theorem 89), the theory of $\mathcal{A}'_{Sk}$ is undecidable as well as consequence of Proposition 3. $\square$

Theorem 93 ([Rob49]) *The theory of $\mathcal{A}_{Pr|}$, that is Presburger Arithmetic enriched by the predicate "x divides y" is undecidable.*

Proof: This can be shown by an interpretation of the structure $\mathcal{A}'_{Sk}$ in the structure $\mathcal{A}_{Pr|}$, see [Rob49]. $\square$

There are however decidable fragment of the theory of $\mathcal{A}_{Pr|}$:

Theorem 94 ([Lip78]) *The Σ_1-fragment of the theory of $\mathcal{A}_{Pr|}$ is decidable.*

Theorem 95 ([BI05]) *The following restrictions of the theory of $\mathcal{A}_{Pr|}$ are decidable:*

1. *Sentences of the form $QzQ_1x_1 \ldots Q_nx_n\phi$ where ϕ is quantifier free, and the only variable allowed to occur as first argument of a divisibility predicate is z.*

2. *Sentences of the form $\exists z_1, \ldots, z_n\ Q_1x_1 \ldots Q_nx_n\phi$ where ϕ is quantifier free and positive, and the only variables allowed to occur as first argument of a divisibility predicate are $z_1, \ldots, z_n$.*

Proposition 96 *The predicate "x is an integer" cannot be defined in the theory of $\mathcal{A}_{\mathbb{R}}$.*

Proof: If this predicate were definable in $\mathcal{A}_{\mathbb{R}}$ then there would exist a one-dimensional interpretation of $\mathcal{A}_{Pe}$ in $\mathcal{A}_{\mathbb{R}}$, in contradiction to the decidability of the theory of $\mathcal{A}_{\mathbb{R}}$ (Theorem 29), the undecidability of the theory of $\mathcal{A}_{Pe}$ (Theorem 89), and Proposition 3. □

Theorem 97 ([Rob49]) *The theory of $\mathcal{A}_{\mathbb{Q}}$ is undecidable.*

Proof: There is a one-dimensional interpretation of $\mathcal{A}_{Pe}$ in $\mathcal{A}_{\mathbb{Q}}$ since the predicate "x is an integer" can be defined in $\mathcal{A}_{\mathbb{Q}}$ (the definition of that predicate is non-trivial, see Section 3 of [Rob49]). □

12.1.5 Example: Variants of $S2S$

Various extensions and variants of $\mathcal{A}_{S2S}$ have an undecidable first-order theory.

Theorem 98 *The first order theory of the structure $\mathcal{A}'_{S2S}$, which is defined as $\mathcal{A}_{S2S}$ except that $0(v)$ is interpreted as $0v$, is undecidable.*

We write, when talking about the structure $\mathcal{A}'_{S2S}$, the two operations simply as $y = 0x$ and $y = x1$. In the logic of this structure we can express that y can be obtained from x by adding some 1 at the right, respectively by adding some 0 on the left:

$$
\begin{array}{rcl}
x \leq_0 y & := & \forall X(x \in X \wedge \forall z : z \in Y \to 0z \in Y) \to y \in X \\
x \leq_1 y & := & \forall X(x \in X \wedge \forall z : z \in Y \to z1 \in Y) \to y \in X
\end{array}
$$

We can now easily express that a word is composed of 0, resp. 1, only:

$$
\begin{array}{rcl}
x \in 0^* & := & \epsilon \leq_0 x \\
x \in 1^* & := & \epsilon \leq_1 x
\end{array}
$$

Proof: For this proof we use two sets to represent a word, and four sets to represent a set of pairs of words. More precisely, a word is represented as the set of positions labeled with a, and the set of positions labeled with b, where positions are sequences of 1:

$$
\begin{array}{rcl}
\phi(w) & = & (\{1^n \mid w(n) = a\}, \{1^n \mid w(n) = b\}) \\
\mathtt{epsilon}(X_a, X_b) & = & \neg\exists x(x \in X_a \vee x \in X_b) \\
\mathtt{append}_{\mathtt{a}}(X_a, X_b, Y_a, Y_b) & = & \forall x.x \in Y_a \leftrightarrow (x \in X_a \vee \exists y.y = \max(X_a, X_b) \wedge x = y1) \wedge Y_b = X_b \\
\mathtt{append}_{\mathtt{b}}(X_a, X_b, Y_a, Y_b) & = & \forall x.x \in Y_b \leftrightarrow (x \in X_b \vee \exists y.y = \max(X_a, X_b) \wedge x = y1) \wedge Y_a = X_a
\end{array}
$$

where $y = max(X_a, X_b)$ can be easily expressed as

$$(y \in X_a \vee y \in X_b) \wedge \forall x((x \in X_a \vee x \in X_b) \rightarrow x \leq_1 y)$$

Finally, we represent a set of pairs of words similarly to the encoding of words, by additionally prefixing the representation of each word with a sequence of 0, the length of which indicates the index. More precisely, $\psi((v_1, w_1), \ldots, (v_n, w_n))$ is defined as

$$(\{0^i 1^j \mid v_i(j) = a\}, \{0^i 1^j \mid v_i(j) = b\}, \{0^i 1^j \mid w_i(j) = a\}, \{0^i 1^j \mid w_i(j) = b\})$$

And we can defined accordingly (X_a, X_b, Y_a, Y_b) `in` (Z_1, Z_2, Z_3, Z_4) as

$$\begin{array}{rcl} \exists p \in 0^* & . & \forall x.(x \in X_a \leftrightarrow x \in 1^* \wedge \exists y(y \in Z_1 \wedge x \leq_0 y \wedge p \leq_1 y)) \\ & \wedge & \forall x.(x \in X_b \leftrightarrow x \in 1^* \wedge \exists y(y \in Z_2 \wedge x \leq_0 y \wedge p \leq_1 y)) \\ & \wedge & \forall x.(x \in Y_a \leftrightarrow x \in 1^* \wedge \exists y(y \in Z_3 \wedge x \leq_0 y \wedge p \leq_1 y)) \\ & \wedge & \forall x.(x \in Y_b \leftrightarrow x \in 1^* \wedge \exists y(y \in Z_4 \wedge x \leq_0 y \wedge p \leq_1 y)) \end{array}$$

□

There are other ways to prove this result. One way, indicated in [Tho90], is to represent a grid by observing that words from the set 0^*1^* represent positions on the grid where we can express movement on both axes (by adding and removing an 0 on the left or an 1 on the right). Set variables can be used to simulate different colorings of these position on the grid. Using this we can express the halting problem of a Turing machine by representing a grid where each line corresponds to a configuration of the Turing machine.

In fact, a word of the form $0^i 1^j$ represents a pair of natural numbers (i, j) where we can express easily in the logic the operations of successor arithmetic on each of the components like incrementing and decrementing by one, and comparison with 0. This is precisely what we need to simulate the halting problem of Minsky Machines.

Exercice 18
Show Theorem 98 by reduction of the halting problem of Minsky Machines (Theorem 5). Hint: use one set variable for each possible value of the program counter.

Theorem 99 *The theory of the structure $\mathcal{A}_{WS2S}$, extended by the predicate "x and y have the same length", is undecidable.*

It has been shown in [Tho90] that the equal-length predicate allows to encode the $x = 0y$, when y is restricted to 0^*1*.

12.1.6 Example: The Theory of $\mathcal{A}_{FT\leq}$

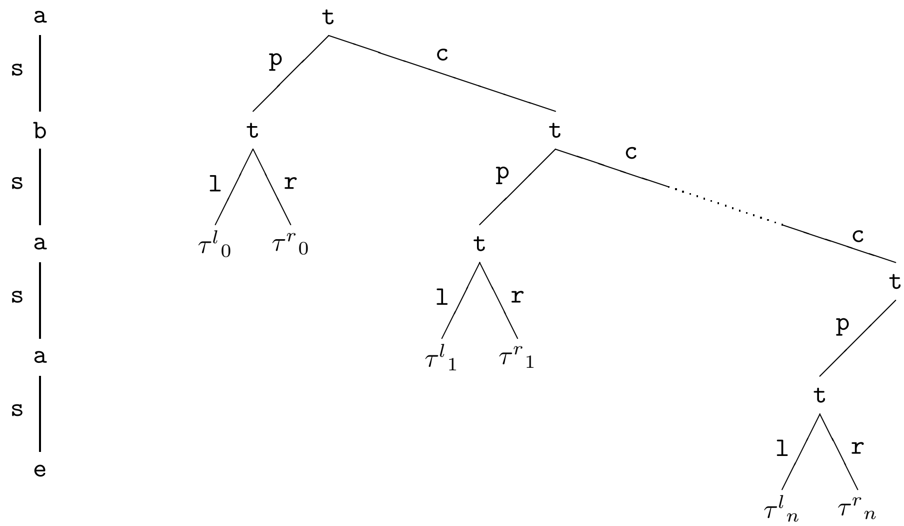
Theorem 100 ([MNT98, MNT01]) *The first-order theories of $\mathcal{A}_{FT\leq}$ is undecidable.*

For this result we have to apply the variant of our method given at the end of Section 12.1.1 since our model contains infinite trees.

Proof: We assume w.l.o.g. label symbols $\mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{t} \in \mathcal{L}$ and feature symbols $\mathtt{s}, \mathtt{c}, \mathtt{p}, \mathtt{l}, \mathtt{r} \in \mathcal{F}$.

We define the encoding of words by $\phi(c_0 \cdots c_n) = \tau$ with (see Figure 12.1(a))

$$\begin{array}{rcl} D_\tau & = & \{\mathtt{s}^i \mid 0 \leq i \leq n\} \\ L_\tau(\mathtt{s}^i) & = & c_{n-i} \end{array}$$



(a) The string $aaba$.

(b) The set $\{(v_0, w_0), \ldots, (v_n, w_n)\}$.

Figure 12.1: Representation of strings and of solution sequences.

The formulas $\texttt{epsilon}(x)$ and $\texttt{append}_w(x, y)$ are as always easy to define, where $\texttt{append}_w(x, y)$ is defined by induction on w:

$$\begin{array}{rcl} \texttt{epsilon}(x) & := & \texttt{e}(x) \\ \texttt{append}_{\texttt{a}}(x, y) & := & \texttt{a}y \wedge y[\texttt{s}]x \\ \texttt{append}_{\texttt{b}}(x, y) & := & \texttt{b}y \wedge y[\texttt{s}]x \end{array}$$

The representation of a set of pairs of words is obtained by arranging the representations of the words in a list. More precisely, $\psi(\{(v_0, w_0), \ldots, (v_n, w_n)\}) = \sigma$ with (see Figure 12.1(b))

$$\begin{array}{rcl} D_\sigma & = & \{\texttt{c}^i, \texttt{c}^i\texttt{p} \mid 1 \leq i \leq n\} \\ & \cup & \{\texttt{c}^i\texttt{pl}\pi \mid 1 \leq i \leq, \pi \in D_{\phi(v_i)}\} \\ & \cup & \{\texttt{c}^i\texttt{pr}\pi \mid 1 \leq i \leq, \pi \in D_{\phi(w_i)}\} \end{array}$$

The tricky part is the definition of the formula (x, y) $\texttt{in}$ z. For this purpose we define, for any formula ϕ, a formula $\mu x \phi$ expressing that x denotes the *smallest* element satisfying ϕ. This formula is stronger than $\min x \phi$ in that it requires the existence of a smallest tree satisfying ϕ in addition:

$$\mu x \phi := \phi \wedge \forall y\, (\phi\{x \mapsto y\} \rightarrow x \leq y)$$

We can define in an analogous way:

$$\nu x \phi := \phi \wedge \forall y\, (\phi\{x \mapsto y\} \rightarrow x \geq y)$$

If x denotes a tree as given in Figure 12.1(b), then the formula $\mathsf{one\text{-}branch}(x, x')$ given below expresses that x' denotes a tree as given in Figure 12.2.

$$\begin{array}{rcl} \mathsf{one\text{-}branch}(x, x') & := & \exists x_c\, (\max x_c (\mathsf{string}(x_c) \wedge x_c < x) \\ & & \quad \wedge x_c < x' \leq x \\ & & \quad \wedge \max x' (\exists z\, (\mu z (x_c \leq z \leq x')))) \end{array}$$

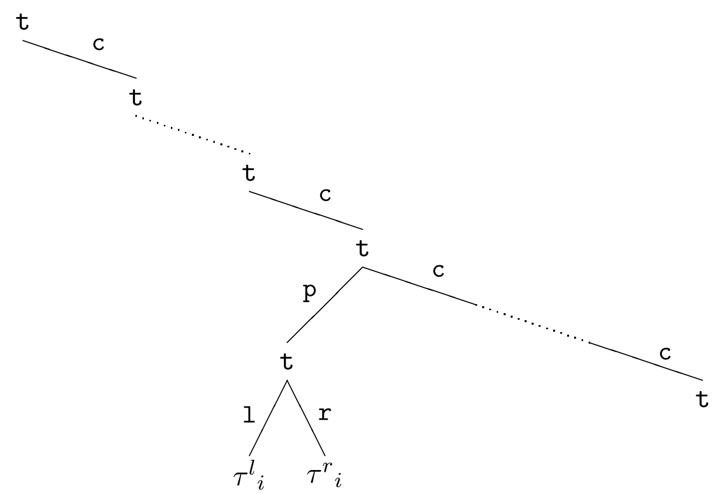


Figure 12.2: A possible value for x' such that $\mathsf{one\text{-}branch}(x, x')$, where x is as in Figure 12.1(b).

In this formula, x' is smaller than x but is strictly greater than the c-spine x_c of x. The tree x' can have only one of the p-edges of x since the set of trees between x_c and x' must have a smallest element. By the maximality of x', the tree x' contains x_c plus exactly one of the subtrees of x starting with a p-edge (see Figure 12.2).

The following formula $\mathsf{select}(\tau^l, \tau^r, \sigma)$, where σ is as in Figure 12.2, expresses that τ^l is the tree $\tau^l{}_i$ and τ^r is the tree $\tau^r{}_i$:

$$\begin{aligned}\mathsf{select}(y_l, y_r, x') \quad &:= \quad \exists x'' \, (\mu x''(x' \leq x'' \wedge \exists x''' \, (x''[c]x''' \wedge x''' \leq x'')) \\ &\qquad \wedge \exists z \, (x''[\mathtt{p}]z \wedge z[\mathtt{l}]y_l \wedge z[\mathtt{r}]y_r))\end{aligned}$$

From a tree σ' as given in Figure 12.2, we get the tree σ'' (denoted by x'') containing at all nodes c^j with $j \leq i$ a pair $(\tau^{l'}{}_j, \tau^{r'}{}_j)$ such that $\tau^l{}_i \leq \tau^{l'}{}_j$ and $\tau^r{}_i \leq \tau^{r'}{}_j$ (by Lemma 11). By the minimality of σ'' we get that $\tau^l{}_i = \tau^{l'}{}_j$ and $\tau^r{}_i = \tau^{r'}{}_j$ for all $j \leq i$, hence in particular for $j = 0$ (see Figure 12.3). Combination of the two formulas yields

$$\mathsf{in}(y_l, y_r, x) \quad := \quad \exists x{}'(\mathsf{one\text{-}branch}(x, x') \wedge \mathsf{select}(y_l, y_r, x'))$$

Now, it is easy to verify that the predicate (x, y) `in` z behaves as required by Theorem 78.

Finally, the predicate `finite`(x) is now defined by expressing that either x is one node only (representing the empty string), or the set of feature trees that are *strictly* smaller than x possesses a greatest element:

$$\mathtt{finite}(x) \quad := \quad \mathsf{atom}(x) \vee \exists y \, (\nu y(y < x))$$

$\square$

The original proof in [MNT98, MNT01] was given for a slightly different setting in which feature trees are only *partially* labeled, that is where the labelling function of a feature tree τ is any partial function $L_\tau \colon D_\tau \to \mathcal{L}$. In that case, the proof is the same as the one given here (up to minor technical differences).

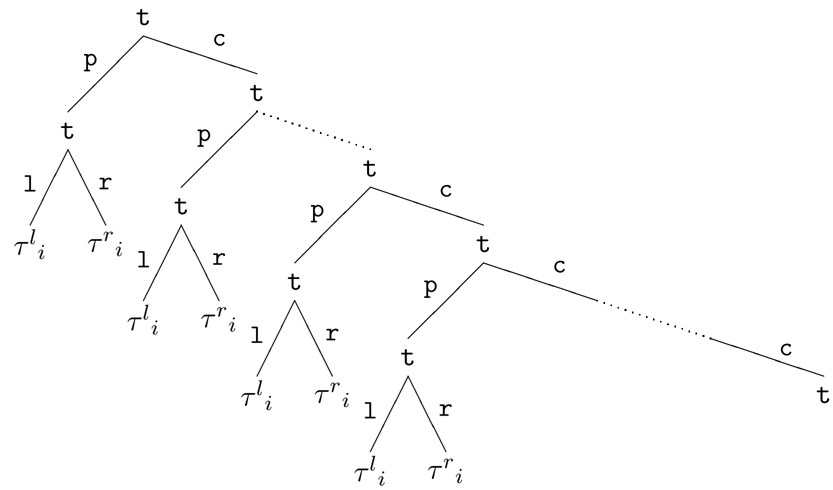


Figure 12.3: The value of x' in the formula $\mathsf{select}(y_l, y_r, x)$ where x is as in Figure 12.2.

12.1.7 Example: The Theory of Non-Structural Subtyping

Theorem 101 ([SAN$^+$02]) *The first-order theories of $\mathcal{A}_{\mathrm{NSub}}$ and $\mathcal{A}_{\mathrm{FNSub}}$ are undecidable.*

The proof is very similar to the one of Theorem 100. Recall that the first-order theory of *structual* subtyping, in contrast to the *non-structural* subtyping considered in Theorem 101, is decidable (Theorem 47).

12.1.8 Example: The Theory of One-Step Rewriting

12.2 Coding the Post Correspondence Problem via Lists

12.2.1 Example: The Theory of a Lexicographic Path Ordering

Theorem 102 ([CT97]) *Let Σ be a finite signature and $>$ a precedence (not necessarily total) such that*

1. *there exists a constant* 0 *which is minimal among the constants,*
2. *there exists a binary f which is minimal in $\Sigma \setminus \{0\}$,*
3. *there exists a unary g which is mininmal in $\{h \mid h > f\}$.*

Then the Σ_2-fragment of the theory of $\mathcal{A}_{\mathrm{lpo}}$ is undecidable.

The proof is quite technical. Recall that the Σ_1-fragment is decidable for any total precedence (Corollary 22).

Chapter 13

Constraint Systems not Covered Here

There are several classes of constraint systems that are not discussed in this survey for various reasons.

13.1 Unification

Unification is the problem of solving basic constraints in the structure of ground terms, often modulo some set of equational axioms. Solving unification problems is of paramount importance for logic programming and automated deduction. There are so many results on the decidability and complexity of the unification problem for various equational axioms that we cannot discuss all of them in the present work. The interested reader should consult [BS01] for an overview.

13.2 Constraints for Cryptographic Protocol Analysis

Solving constraints for the analysis of cryptographic protocols, as introduced in Section 2.2.9, stands a bit apart from the constraint systems discussed in this paper since the underlying constraint system is defined by an inference system. As a consequence, techniques for solving these constraints are based on an analysis of the proof trees permitted by these inference systems. The interested reader might, for instance, consult [RT03, CLS03, DLLT06]. The paper [CDL05] gives on overview of the different equational theories that are important for symbolic protocol analysis.

13.3 Combination of Constraint Systems

The problem of combination can be phrased as follows: given two constraint systems, each of them with a decision algorithm, construct a decision algorithm for the combination of these two constraint systems. The notion of combination of two constraint systems is usually defined through axiomatizations, that is given axiomatizations of the two constraint systems, their combination is given as the syntactic union of these two axiom systems. These axiom systems are typically not complete in the sense of first-order logic, in this case one also has to define whether one is interested in validity or satisfiability with respect to an axiom system.

There exist various approaches to this problem, depending on

- the class of formulas one is interested in;

- the question whether the solvers are "black boxes" or whether one may make assumptions on their implementation;
- the kind of computational service one is interested in. This could for instance be deciding satisfiability or validity, computation of solutions (in case of non-closed formulas), construction of proof objects or human-readable explanations.

Milestones in this domain are the works of Nelson and Oppen [NO79], Shostak [Sho82], and Baader and Schulz [BS96]. The habilitation thesis of Christophe Ringeissen [Rin09] gives a good overview, and pointers to the literature.

13.4 Satisfiability Modulo Theories (SMT)

This recent research direction aims at developing practically fast algorithms for deciding satisfiability of quantifier-free constraints in conjunctive normal form, given a satisfiability algorithm for conjunctions of positive and negative literals. This is achieved by extending the solver for conjunctions of literals using the techniques that have made Boolean satisfiability (SAT) so successful, like backjumping and conflict driven clause learning [MZ09]. See [NORCR07] for a survey.

Appendix A

A Catalogue of Results

We summarize the decidability and non-decidability results mentionend in these lectures. Theories are understood to be first-order unless mentionend otherwise.

A.1 Numbers (natural, integer)

Language	Decidability
0, s, $=$	Yes (Theorem 28 page 66)
0, s, $+$, $=$, $<$	Yes (Theorem 31 page 71 Lower bound NTIME($2^{2^{cn}}$) [FR74] Upper bound SPACE($2^{2^{2^{cn}}}$) [FR79]
0, $+$, $*$, $=$, $<$	No (Theorem 89 page 113) No for positive $\exists^*$ fragment [Mat70]
0, s, $*$, $=$	No (Theorem 92 page 114)
0, s, $\vert$, $=$	No (Theorem 93 page 114) Yes for Σ_1 (Theorem 94 page 115, [Lip78]) Yes for restricted quantification (Theorem 95 page 115, [BI05])
$0, 1, \ldots, *, =$	Yes

A.2 Rational numbers

Language	Decidability
0, 1, -1, $+$, $*$, $<$, $=$	No, Therorem 97 page 115 [Rob49]

A.3 Real numbers

Language	Decidability
0, 1, -1, $+$, $*$, $<$, $=$	Yes, Theorem 29 page 67 [Tar51]

A.4 Words

Language	Decidability
ϵ, a, b, $\cdot$, $=$	No (Theorem 79 page 111) No for fragment $\exists\forall\forall\forall$*equation* [Dur95a] Yes for $\exists^*$ fragment [Mak77], [BS86]
ϵ, xa, xb, ax, bx, $=$	Yes [RV01], Exercice 12

A.5 Words - (Weak) Second Order

Language	Decidability
ϵ, xa, xb	Yes (Theorem 71 page 100)
ϵ, xa, bx	No [Tho90]

A.6 Ground Terms

Language	Decidability
Σ, $=$	Yes [Mal71] Non-elementary complexity [Vor96]
Σ, $=$, $\prec_{\mathrm{lpo}}$ total precedence	Yes for the Σ_1-fragment (Corollary 22, page 63) Yes for monadic signatures (Theorem 68, page 97) No for the Σ_2-fragment [CT97]
Σ, $=$, $\prec_{\mathrm{kbo}}$	Yes (Theorem 46, page 78) NP-complete for the Σ_1-fragment (Theorem 24, page 64).
Σ, $=$, $\prec_{\mathrm{struct}}$	Yes (Theorem 47, page 78) PSPACE for basic constraints (Theorem 48, page 78)
Σ, $=$, $\prec_{\mathrm{non-struct}}$	No (Theorem 101, page 119)
$x = t_i$, $\mathrm{red}_{r_i}(x)$	Yes (Theorem 76, page 103)

A.7 Ground Terms Modulo E

Equational Theory	Decidability
AC	No for the Σ_3-fragment (Theorem 83 page 111) No for the Σ_2-fragment [Mar97] Yes for the Σ_1-fragment [Com88]

A.8 Feature Trees

Language	Decidability
Ax, $x[f]y$, $xf\uparrow$, $=$	Yes (Theorem 42 page 76) [BS95] Non-elementary complexity [Vor96] Satisfiability and Entailment: quasi-linear [ST94]
Ax, $x[f]y$, $xf\uparrow$, $\leq$	No [MNT01] Satisfiability in n^3 [MNP97] Entailment PSPACE-complete [MNT01]
Ax, $x[f]y$, $x\{\cdots\}$, $=$	Yes (Theorem 43 page 77) [Bac95] Non-elementary complexity [Vor96] Satisfiability and Entailment: quasi-linear [ST94]
Ax, $x[v]y$, $x\{\cdots\}$, $=$ (first-class features)	No [Tre93] Satisfiability NP-complete (Theorem 19 page 59) [Tre93]

Bibliography

[Ack54] Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* North-Holland, Amsterdam, 1954.

[AFM96] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On confluence of constraint handling rules. In Eugene Freuder, editor, *Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

[Apt90] Krzysztof R. Apt. Logic programming. In van Leeuwen [vL90], chapter 10, pages 493–574.

[Bac95] Rolf Backofen. A complete axiomatization of a theory with feature and arity constraints. *Journal of Logic Programming*, 24(1&2):37–71, July/August 1995.

[BC93] Alexandre Boudet and Hubert Comon. About the theory of tree embedding. In M. C. Gaudel and J.-P. Jouannaud, editors, *Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 376–390, Orsay, France, April 1993. Springer-Verlag.

[BC96] Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In Hélène Kirchner, editor, *CAAP'96*, volume 1059 of *Lecture Notes in Computer Science*, Linköping, Sweden, April 1996. Springer-Verlag.

[BCR98] Jacek Bochnak, Michel Coste, and Marie-Franoise Roy. *Real Algebraic Geometry.* Springer-Verlag, 1998.

[BG00] Achim Blumensath and Erich Grädel. Automatic structures. In *15th Annual IEEE Symposium on Logic in Computer Science*, pages 51–62, Santa Barbara, CA, June 2000. IEEE Computer Society.

[BI05] Marius Bozga and Radu Iosif. On decidability within the arithmetic of addition and divisibility. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 425–439. Springer-Verlag, April 2005.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BS86] J. Richard Büchi and Steven Senger. Coding in the existential theory of concatenation. *Archiv math. Logik*, 26:101–106, 1986.

[BS95] Rolf Backofen and Gert Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268, July 1995.

[BS96] Franz Baader and Klaus Schulz. Unification in the union of disjoint equational theories: ,combining decision procedures. *Journal of Symbolic Computation*, 21(2):211–243, February 1996.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier and MIT Press, 2001.

[Büc60a] J. Richard Büchi. On a decision method in restricted second order arithmetic. In E. Nagel et. al., editor, *International Congr. on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[Büc60b] J. Richard Büchi. Weak second order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[CCC+94] Anne-Cécile Caron, Hubert Comon, Jean-Luc Coquidé, Max Dauchet, and Florent Jacquemard. Pumping, cleaning and symbolic constraints solving. In Serge Abiteboul and Eli Shamir, editors, *21st International Colloquium on Automata, Languages and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 436–449, Jerusalem, Israel, July 1994. Springer-Verlag.

[CCD93] Anne-Cécile Caron, Jean-Luc Coquide, and Max Dauchet. Encompassment properties and automata with constraints. In Claude Kirchner, editor, *5th International Conference on Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 328–342, Montreal, Canada, June 1993. Springer-Verlag.

[CDG+97] Hubert Comon, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 1997.

[CDL05] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 2005. To appear.

[CJ99] Hubert Comon and Yan Jurski. Counter automata, fixpoints and additive theories. `http://www.lsv.ens-cachan.fr/~comon/ftp.articles/mca.ps`, December 1999.

[CK90] C. C. Chang and H. J. Keisler. *Model Theory*, volume 73 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, third edition, 1990.

[CKRP73] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un système de communication homme-machine en Français. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[CL89] Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[CLS03] H. Comon-Lundh and V. Shmatikov. Constraint solving, exclusive or and the decision of confidentiality for security protocols assuming a bounded number of sessions. In *18th IEEE Symposium on Logic in Computer Science* [IEE03], pages 271–280.

[CMT01] Hubert Comon, Claude Marché, and Ralf Treinen, editors. *International Summer School on Constraints in Computational Logics (CCL'99)*, volume 2002 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2001.

[Com88] Hubert Comon. Unification et disunification. Théorie et applications, 1988. Doctoral Thesis, Institut National Polytechnique de Grenoble.

[Com90a] Hubert Comon. Solving inequations in term algebras. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 62–69. IEEE Computer Society Press, 1990.

[Com90b] Hubert Comon. Solving symbolic ordering constraints. *Int. J. Found. Comput. Sci.*, 1(4):387–412, 1990.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7, 1972.

[CT94] Hubert Comon and Ralf Treinen. Ordering constraints on trees. In Sophie Tison, editor, *CAAP'94*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, April 1994. Springer-Verlag. (Invited Lecture).

[CT97] Hubert Comon and Ralf Treinen. The first order theory of lexicographic path orderings is undecidable. *Theoretical Computer Science*, 176(1–2), April 1997.

[DCC95] Max Dauchet, Anne-Cécile Caron, and Jean-Luc Coquidé. Automata for reduction properties solving. *Journal of Symbolic Computation*, 20(2):215–233, 1995.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 7:279–301, 1982.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[Dia07] Daniel Diaz. The gnu prolog web site, 1999-2007. `http://www.gprolog.org/`.

[Die02] Volker Diekert. Makanin's algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of Mathematics and Its Applications*, chapter 12, pages 387–442. Cambridge University Press, 2002.

[DLLT06] Stéphanie Delaune, Pascal Lafourcade, Denis Lugiez, and Ralf Treinen. Symbolic protocol analysis in presence of a homomorphism operator and *xclusive or*. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 132–143, Venice, Italy, July 2006. Springer-Verlag.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proccedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, January 1982. ACM.

[DSH90] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:75–93, 1990.

[Dur95a] Valery G. Durnev. Undecidability of the positive $\forall\exists^3$-theory of a free semigroup. *Siberian Mathematical Journal*, 36(5):917–929, September–October 1995. English translation of [Dur95b].

[Dur95b] Valery G. Durnev. Undecidability of the positive $\forall\exists^3$-theory of a free semigroup. *Sibirskiĭ Matematicheskiĭ Zhurnal*, 36(5):1067–1080, September 1995. English translation as [Dur95a].

[DY83] D. Dolev and A.C. Yao. On the security of public-key protocols. In *Transactions on Information Theory*, volume 29, pages 198–208. IEEE Computer Society Press, March 1983.

[End72] Herbert B. Enderton. *Mathematical Introduction to Logic*. Academic Press, 1972.

[Fef86] Solomon Feferman, editor. *Kurt Gödel, Collected Works*. Oxford University Press, 1986. 2 volumes.

[FR74] Michael. J Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In R. M. Karp, editor, *Complexity of Computation*, volume 7, pages 27–41. American Mathematical Society, Providence, RI, 1974.

[FR79] Jeanne Ferrante and Charles W. Rackoff. *The computational complexity of logical theories*. Number 718 in Lecture Notes in Mathematics. Springer-Verlag, 1979.

[Fre02] Alexandre Frey. Satisfying subtype inequalities in polynomal space. *Theoretical Computer Science*, 277:105–117, April 2002.

[GN01] Harald Ganzinger and Robert Nieuwenhuis. Constraints and theorem proving. In Comon et al. [CMT01], pages 159–201.

[Göd31] Kurt Gödel. Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. Reprinted in [Fef86].

[Gun92] Carl A. Gunter. *Semantics of Programming Lamguages*. MIT Press, 1992.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[HJ90] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 197–209, San Francisco, CA, January 1990. ACM.

[Hod93a] Wilfrid Hodges. *Model Theory*. Encyclopedia of Mathematics and its Applications 42. Cambridge University Press, 1993.

[Hod93b] Wilfried Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.

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

[IEE03] IEEE Computer Society. *18th IEEE Symposium on Logic in Computer Science*, Ottawa, Canada, June 2003. IEEE Computer Society.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Conference on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. Satisfiability of systems of ordinal notation with the subterm property is decidable. In Javier Leach Albert, Burkhard Monien, and M. Rodriguez Artalejo, editors, *18th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 510, pages 455–468, Madrid, Spain, 1991. Springer Verlag.

[JT01] Jean-Pierre Jouannaud and Ralf Treinen. Constraints and constraint solving: An introduction. In Comon et al. [CMT01], pages 1–46.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Kel93] Bill Keller. *Feature Logics, Infinitary Descriptions and Grammar.* CSLI Lecture Notes No. 44. Center for the Study of Language and Information, 1993.

[KKR90] Claude Kirchner, Hélène Kirchner, and Michaël Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990.

[Kow74] Robert A. Kowalski. Logic as a programming language. In Jack L. Rosenfeld, editor, *Information Processing 74*, pages 569–574, Stockholm, Sweden, August 1974. North-Holland.

[KR03] Viktor Kuncak and Martin C. Rinard. Structural subtyping of non-recursive types is decidable. In *18th IEEE Symposium on Logic in Computer Science* [IEE03], pages 96–107.

[KV01] Konstantin Korovin and Andrei Voronkov. Knuth-Bendix constraint solving is NP-complete. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 979–992, Crete, Greece, July 2001. Springer-Verlag.

[Lip78] L. Lipshitz. The Diophantine problem for addition and divisibility. *Transactions of the AMS*, 235:271–283, January 1978.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Löw15] Leopold Löwenhein. Über Möglichkeiten im Relativkalkül. *Mathematische Annalen*, 76(4):447–470, June 1915.

[LS87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification.* Wiley/Teubner, 2nd edition, 1987.

[Mah88] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *LICS*, pages 348–357, Edinburgh, Scotland, UK, July 1988. IEEE Computer Society.

[Mak77] G. S. Makanin. The problem of solvability of equations in a free semi-group. *Akad. Nauk SSSR*, 232(2), 1977.

[Mal71] Anatolií Ivanovič Malc'ev. Axiomatizable classes of locally free algebras of various type. In III Benjamin Franklin Wells, editor, *The Metamathematics of Algebraic Systems: Collected Papers 1936–1967*, chapter 23, pages 262–281. North Holland, 1971.

[Mar97] Jerzy Marcinkowski. Undecidability of the first order theory of one-step right ground rewriting. In Hubert Comon, editor, *8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 241–253, Sitges, Spain, June 1997. Springer-Verlag.

[Mat70] Yu Matijacevič. Enumerable sets are diophantine. *Dokl. Akad. Nauk. SSSR*, 191:279–282, 1970.

[Mie04] Pawel Mielniczuk. Basic theory of feature trees. *ACM Transactions on Computational Logic*, 15(3):385–402, July 2004.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[MNP97] Martin Müller, Joachim Niehren, and Andreas Podelski. Ordering constraints over feature trees. In Gert Smolka, editor, *Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 297–311, Linz, Austria, October/November 1997. Springer-Verlag.

[MNT98] Martin Müller, Joachim Niehren, and Ralf Treinen. The first-order theory of ordering constraints over feature trees. In Vaughan Pratt, editor, *Thirteenth IEEE Annual Symposium on Logic in Computer Science*, pages 432–443, Indianapolis, IN, June 1998. IEEE Computer Society.

[MNT01] Martin Müller, Joachim Niehren, and Ralf Treinen. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics and Theoretical Computer Science*, 4(2):193–234, September 2001.

[Mon81] J. Mongy. *Transformation de noyaux reconnaissables d'arbres*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Villeneuve d'Ascq, France, 1981.

[MS98] Kim Marriott and Peter Stuckey. *Programming With Constraints: An Introduction*. The MIT Press, 1998.

[MS01] Jonathan Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 166–175, 2001.

[MZ09] Sharad Malik and Lintao Zhang. Boolean satisfiability - from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, August 2009.

[Nie93] Robert Nieuwenhuis. Simple LPO-constraint solving methods. *Information Processing Letters*, 47:65–69, 1993.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NORCR07] Robert Nieuwenhuis, Albert Oliveiras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in satisfiability modulo theories. In Franz Baader, editor, *18th International Conference on Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 2–18, Paris, France, June 2007. Springer-Verlag.

[NR99] Robert Nieuwenhuis and José Miguel Rivero. Solved forms for path ordering constraints. In Paliath Narendran and Michael Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, pages 1–15, Trento, Italy, July 1999. Springer-Verlag.

[NR00] Paliath Narendran and Michael Rusinowitch. The theory of a total unary rpo is decidable. In *Proceedings of the First International Conference on Computational Logic*, pages 660–672, London, UK, July 2000.

[Opp73] Derek C. Oppen. Elementary bounds for Presburger arithmetic. In STOC73 [STO73], pages 34–37.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of the 24th Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997. ACM.

[Pla99] Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *40th Annual Symposium on Foundations of Computer Science*, pages 721–725, New York, NY, USA, October 1999. IEEE Computer Society.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the AMS*, 52:264–268, 1946.

[Pre29] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt. In *Comptes Rendus du I Congres de Mathematiciens des Pays Slaves*, pages 92–101, 1929.

[PWO97] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[Qui46] W. V. Quine. Concatenation as a basis for arithmetic. *Journal of Symbol Logic*, 11(4):105–114, 1946.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Rin09] Christophe Ringeissen. Equational reasoning and combination methods: from programs to proofs. Habilitation Thesis, November 2009. `http://www.loria.fr/~ringeiss/HDR-CR/hdr-cr-en.pdf`.

[RL78] C. R. Reddy and Donald W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *STOC*, pages 320–325, San Diego, CA, USA, May 1978. ACM.

[Rob49] Julia Robinson. Definability and decision problems in arithmetic. *Journal of Symbolic Computation*, 14(2):98–114, June 1949.

[RT03] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with finite number of sessions, composed keys is NP-complete. *Theoretical Computer Science*, 299(1–3):451–475, April 2003.

[RV01] Tatiana Rybina and Andrei Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*, 1(4):1–28, April 2001.

[SAN$^+$02] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *POPL'02*, pages 203–216, Portland, OR, USA, January 2002. ACM.

[SC06] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In *Handbook of Constraint Programming*, chapter 14, pages 495–526. Elsevier, 2006.

[Sch92] Klaus U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In Klaus U. Schulz, editor, *IWWERT'90*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150. Springer-Verlag, 1992.

[Sho82] R.E. Shostak. Deciding combinations of theories. In D. W. Loveland, editor, *6th International Conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, pages 209–222, New York, USA, 1982. Springer-Verlag.

[SM73] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. In STOC73 [STO73], pages 1–9.

[Smo92] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.

[SR90] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 232–245. The MIT Press, June 1990.

[SR91] Vijay Saraswat and Martin Rinard. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th Symposium on Principles of Programming Languages*, pages 333–351, Orlando, FL, January 1991. ACM.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[STO73] ACM. *Symposium on the Theory of Computing*. ACM, 1973.

[Sto74] Larry J. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report MAC TR-133, Poject MAC, MIT, Cambridge, MA, 1974.

[Su02] Zhendong Su. *Algorithms for and the Complexity of Constraint Entailment*. PhD thesis, University of California, Berkeley , CA, October 2002. `http://www.cs.ucdavis.edu/~su/publications/thesis.ps.gz`.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. Reprinted in volume 3 of Alfred Tarski: Collected Papers, edited by Steven R. Givant and Ralph N. McKenzie, Birkäuser 1986.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In van Leeuwen [vL90], chapter 4, pages 133–191.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume III, pages 389–455. Springer-Verlag, 1997.

[Tiu92] Jerzy Tiuryn. Subtype inequalities. In *LICS'92*, pages 308–315, Santa Cruz, CA, USA, June 1992. IEEE Computer Society Press.

[Tre92] Ralf Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–457, November 1992.

[Tre93] Ralf Treinen. Feature constraints with first-class features. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 734–743. Springer-Verlag, August/September 1993.

[Tre97] Ralf Treinen. Feature trees over arbitrary structures. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*, chapter 7, pages 185–211. CSLI Publications and FoLLI, 1997.

[Tre01] Ralf Treinen. Undecidability of the emptiness problem of reduction automata with component-wise tests, June 2001. `http://www.lsv.ens-cachan.fr/~treinen/publi/cra-emptiness.ps.gz`.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to the decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.

[vL90] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B - Formal Models and Semantics. Elsevier Science Publishers and The MIT Press, 1990.

[Vor96] Sergei Vorobyov. An improved lower bound for the elementary theories of trees. In M. A. McRobbie and J. K. Slaney, editors, *CADE'96*, volume 1104 of *Lecture Notes in Computer Science*, pages 275–287, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[vRH04] Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[ZSM05] Ting Zhang, Henny Sipma, and Zohar Manna. The decidability of the first-order theory of Knuth-Bendix order. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 131–148, Tallinn, Estonia, July 2005. Springer-Verlag.