Wydział Matematyki i Informatyki
Uniwersytet Wrocławski

Master Thesis

Directional Type Checker for Prolog

Jacek Śliwerski
jacek@sliwerski.net

Supervisor: Prof. Witold Charatonik

Wrocław, 2006

Abstract

Data types are one of the most common techniques of increasing program comprehension. At the same time logic programs are a convenient way of specifying problems and finding their solutions. Unfortunately, the most popular logic programming language – PROLOG – does not enforce type correctness.

The present thesis describes a directional type system for logic programs, an algorithm for checking type correctness of a program as well as offering the implementation of an appropriate type checker. In order to combine efficiency with type system expressiveness we have used a state of the art mechanism for the implementation of the underlying data structures.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Type Systems . . . 2
1.3 Quick Tour . . . 3
1.4 Contributions . . . 4
1.5 Structure . . . 4

2 Notations and Definitions **5**
2.1 General Concepts . . . 5
2.2 Tree Automata . . . 6
2.2.1 Basic Definitions . . . 7
2.2.2 Supplementary Definitions . . . 9
2.3 Directional Types . . . 10
2.4 Summary . . . 12

3 Type Checking **13**
3.1 Overview . . . 13
3.2 Magic-Set Transformation . . . 13
3.3 Sharp Automaton . . . 14
3.4 Non-emptiness . . . 15
3.5 Summary . . . 18

4 Data Structures **19**
4.1 Outline . . . 19
4.2 Example . . . 19
4.3 Advantages . . . 21
4.4 Efficiency Evaluation . . . 22
4.5 Summary . . . 24

5 Conclusions **25**
5.1 Typed Programs . . . 25

5.2 Limitations 28
5.3 Related Work 29

A User Documentation **30**
A.1 Tutorial 30
A.1.1 Append 30
A.1.2 Interface 31
A.1.3 Type Definitions 31
A.1.4 Verification 31
A.1.5 Determinism 32
A.1.6 Counterexamples 32
A.2 Reference Manual 33
A.2.1 Preprocessor 33
A.2.2 Type Definitions 35
A.2.3 Example 36
A.2.4 Type Annotations 36

B Implementation Details **37**
B.1 Binary Decision Diagrams 37
B.2 Resolution 39
B.3 Efficiency Evaluation 41

Bibliography **43**

Chapter 1

Introduction

Let us begin with a very simple example of several facts and rules:

1. *Jon Burrows is flying to Buenos Aires.*
2. *Elvis is known as Jon Burrows.*
3. *If X is flying to Y and Z is known as X, then Z is alive.*

The first two sentences describe facts. The third one is a rule that lets us draw conclusions not explicitly stated in the former ones. All three sentences constitute a knowledge database which may be later queried in order to find out whether it supports a given sentence. The set of rules and facts is a *logic program* and one may use appropriate tools to verify that the sentence "*Elvis is alive*" is indeed a logical consequence of the three sentences above.

1.1 Motivation

Prolog is one of the most popular *logic programming languages*. In order for our example to be a valid Prolog program, we would have to write it in the following form:

```
is_flying_to(jon_burrows,buenos_aires).
is_known_as(elvis,jon_burrows).
alive(Z) :- is_known_as(Z,X), is_flying_to(X,_).
```

Now we can run the query:

```
?- alive(elvis).
Yes
```

Prolog	SML

`rev([],[]).`
`rev([X|XS],Z) :-`
`    rev(XS,Y),`
`    append(Y,[X],Z).`
	`fun rev nil = nil`
	`  | rev (x::xs) =`
	`      rev xs @ [x]`

Table 1.1: List reversing in Prolog and SML.

and Prolog interpreter confirms that the king of rock 'n' roll is alive. However, if we would have accidentally exchanged `jon_burrows` with `buenos_aires` in the first clause, then our program would not work anymore. One of the most popular mechanisms that defend programmers against this kind of mistakes are type systems which (among others) provide:

safety A type system could detect invalid use of the predicates. It is obvious that the first argument of `is_flying_to` needs to be a person. Thus, an interpreter should be able to discover the mistake... if it knew that `buenos_aires` is not a person.

documentation Type annotations describe the intent of the programmer. An explicit information that the first argument of `is_flying_to` has to be a person helps disambiguate the possible uses of the predicate, especially for a developer who did not author it.

1.2 Type Systems

The most popular type systems are based on the functional perspective of the program, i.e. on the assumption that programmers write *functions* that take arguments and return results (or nothing). Prolog programs, in contrast, are composed of neither *functions* nor *procedures*, but of *clauses* and *predicates*. Let us consider a very simple task of reversing a list. Two implementations are previewed in Table 1.1. If we input the SML version to the interpreter, it will answer us with:

```
rev = fn : 'a list -> 'a list
```

which means: `rev` is a function that takes a list and returns a list. What makes Prolog different is that we can use the `rev` predicate not only this way:

```
rev([1, 2, 3], X).
```

but also as follows:

```
rev(X, [1, 2, 3]).
```

and even so:

```
rev(X, X).
```

depending on what we really need. For this reason it is not easy to answer what arguments have to be passed to the predicate `rev`, because they could be just anything. Therefore, instead of formulating the following:

> *Function f requires an argument of type T1 as an input and it returns a result of type T2 as an output.*

we express an (informal) definition of type correctness as:

> *If the predicate's argument before evaluation is T1 then it is transformed to type T2 after its evaluation.*

And this is why the following type is assigned to the `rev` predicate:

$$(list, \top) \to (list, list).$$

Notice, however, that this is not the ultimate type of the `rev` predicate, rather one of the many possible[1].

1.3 Quick Tour

Consider the `is_flying_to` predicate and the only clause of the program:

```
is_flying_to(jon_burrows,buenos_aires).
```

The process of type checking consists of the following steps:

1. A developer annotates the predicate with appropriate input and output types. Since types are defined by means of *tree automata*, let us use the notation $\mathcal{A}_{(person*place)}$ for an automaton which accepts only those terms that represent pairs of persons and places. The developer could then define the type of the `is_flying_to` clause as: $\mathcal{A}_{(everything)} \to \mathcal{A}_{(person*place)}$.

[1] See Section 5.2 for an explanation why $(\top, list) \to (list, list)$ is rejected by the type checker for the predicate `rev`.

2. The type checker uses the *magic-set transformation* to convert the original clause into the following one:
`is_flying_to`^{out}`(jon_burrows,buenos_aires):-`
`is_flying_to`^{in}`(jon_burrows,buenos_aires)`
which expresses the property of the original clause being *well-typed*.

3. Next, the appropriate tree automata are converted into a *sharp automaton* which accepts only those terms that violate type correctness. In our case the *sharp automaton* would accept terms of the following form:
$$\#(\neg(person * place), (everything))$$

4. Finally, we check whether the actual term (composed of terms from the definition of the clause) could be accepted by the sharp automaton. And since the term
$$\#((\texttt{jon_burrows}, \texttt{buenos_aires}), (\texttt{jon_burrows}, \texttt{buenos_aires}))$$
may not be accepted by the sharp automaton, the type is correct.

1.4 Contributions

The present thesis extends the research of Charatonik and Podelski [Cha00], [CP98] with a backtracking resolution algorithm that handles the full range of regular sets. For the purpose of this thesis the first (to our knowledge) PROLOG type checker that verifies correctness of regular directional types without any restrictions has been implemented. Due to the application of *binary decision diagrams* to the transition relation of tree automata, we have managed to reduce the response time, despite the exponential nature of the type checking algorithm.

1.5 Structure

The remainder of this thesis is organized as follows: Chapter 2 introduces some necessary definitions. The description of the algorithm and a correctness proof of the resolution are presented in Chapter 3 followed by Chapter 4 devoted to the data structures. Finally, Chapter 5 concludes with a short description of the related work and limitations of our approach. The document closes with two appendices: Part A contains the user documentation whereas Part B – implementation details of the tools prepared for this thesis.

Chapter 2

Notations and Definitions

This chapter introduces the most important notations and definitions. Section 2.1 offers some *preliminary notations* and presents the definitions of *logic programs*. Section 2.2 describes our main tool – *tree automata* while Section 2.3 discusses *directional types*. Finally, Section 2.4 closes with a short summary.

2.1 General Concepts

We fix the following notations:

- Σ is a term signature, i.e. a set of function symbols. Functors with arity 0 are called *constant symbols*.
  We will skip Σ in most of the remaining definitions even if they depend on its existence. This way the definitions become more compact (and thus easier to understand) and the signature is inferred during the execution from the environment.
- V is a set of variables.
- T is a set of *all* terms.
- T_Σ is a set of all *ground terms*, i.e. terms constructed only from functors.
- *Pred* is a set of all predicates. All predicates are unary.

Definition 1 (Logic Program) *A logic program P is a set of Horn clauses. We will use the following (*PROLOG*-like) notation of a clause:*

$$p_0(t_0) :- p_1(t_1), \ldots, p_n(t_n).$$

Example

A common example of a logic program is the following definition of the `append` predicate:

```
append([],L,L).
append([X|Xs],Y,[X|Zs]) :- append(Xs,Y,Zs).
```

In order for the above example to conform to our rules, we need to introduce a tuple functor (because predicates are unary) and replace the `[]`-notation with proper functors[1].

```
append(tuple3(nil,L,L)).
append(tuple3(cons(X,Xs),Y,cons(X,Zs)))
  :- append(tuple3(Xs,Y,Zs)).
```

In the above example:

- `append` $\in$ *Pred* is a unary predicate,
- `nil`, `cons`, `tuple3` $\in \Sigma$ are functors with arities 0, 2 and 3 respectively,
- `nil` is a constant symbol,
- `nil` $\in T_\Sigma$ is a ground term,
- `L`, `X`, `Xs`, `Y`, `Zs` $\in V$ are variables,
- `cons`(`X`, `Xs`) $\in T$ is not ground.

2.2 Tree Automata

Tree automata are a convenient abstraction to define types. Sets of terms accepted by tree automata are called *regular types*. In the subsequent chapters we will interchange the definition of a type with the definition of the corresponding automaton. Since the automata are heavily used in this thesis, we have further divided this section into two subsections: Subsection 2.2.1 offers the common definitions and properties whereas Subsection 2.2.2 provides those definitions and lemmas that are necessary for us to prove the correctness of our algorithm.

[1] Although PROLOG defines `.` and `[]` as list constructors, we have decided to use LISP-like ones – `cons` and `nil` – for greater readability.

2.2.1 Basic Definitions

This subsection describes tree automata, their runs and acceptance conditions illustrated with simple examples.

Definition 2 (Tree Automaton) *A tree automaton $\mathcal{A}$ is a triple (Q, S, F) where*

- *Q is a finite set of states,*
- *$F \subseteq Q$ is a set of final states,*
- *S is a set of transition rules of the form:*

  $$f(s_1, \ldots, s_m) \rightarrow s$$

  with $f \in \Sigma$ – an m-ary symbol and $\{s, s_1, \ldots, s_m\} \subseteq Q$.

Definition 3 (Bottom-Up Determinism) *An automaton is called* bottom-up deterministic *if there do not exist two transition rules with the same left-hand side and different right-hand sides. From now on, we will simply use the term* deterministic.

Definition 4 (Completeness) *An automaton $\mathcal{A} = (Q, S, F)$ is called* complete *if and only if*

$$\forall f \in \Sigma \forall q_1 \ldots, q_k \in Q \; \exists q \in Q \text{ s.t. } f(q_1, \ldots, q_k) \rightarrow q \in S$$

Example

Consider the following automaton $\mathcal{A}_{list}$=(Q,S,F):

- $Q = \{\top, list\}$
- $F = \{list\}$
- $S = \{$

$$\begin{aligned}
\texttt{nil} &\rightarrow list \\
\texttt{nil} &\rightarrow \top \\
\texttt{zero} &\rightarrow \top \\
\texttt{cons}(\top, list) &\rightarrow list \\
\texttt{cons}(\top, list) &\rightarrow \top \\
\texttt{cons}(\top, \top) &\rightarrow \top \\
\texttt{cons}(list, \top) &\rightarrow \top \\
\texttt{cons}(list, list) &\rightarrow \top
\end{aligned}$$

$\}$

Term	**Prolog Notation**
`nil`	`[]`
`cons(nil,nil)`	`[[]]`
`cons(zero,cons(zero,nil))`	`[0,0]`
`cons(nil,cons(zero,nil))`	`[[],0]`

Table 2.1: Examples of terms accepted by the automaton $\mathcal{A}_{list}$.

$\mathcal{A}_{list}$ defines the type of a list. It is clearly *complete* (for the inferred signature $\Sigma = \{\texttt{zero}, \texttt{nil}, \texttt{cons}\}$) and *nondeterministic* as there are transition rules $\texttt{nil} \rightarrow list$ and $\texttt{nil} \rightarrow \top$.

Definition 5 (Step) *Let $\mathcal{A} = (Q, S, F)$ be a tree automaton. We define a* step *function as:*

$$step_{\mathcal{A}}\Big(f(q_1, \ldots, q_k)\Big) = \Big\{q \Big| f(q_1, \ldots, q_k) \rightarrow q \in S\Big\}$$

where $f \in \Sigma$ is a k-ary function symbol.

$$step_{\mathcal{A}}(a) = \{q | a \rightarrow q \in S\}$$

is a special case for a constant symbol a, $(k = 0)$.

Definition 6 (Run) *A* run *of an automaton over a ground term is a transitive closure of the step function:*

$$run_{\mathcal{A}}\Big(f(t_1, \ldots, t_k)\Big) = \bigcup_{q_1 \in run_{\mathcal{A}}(t_1), \ldots, q_k \in run_{\mathcal{A}}(t_k)} step_{\mathcal{A}}\Big(f(q_1, \ldots, q_k)\Big).$$

We say that the automaton $\mathcal{A}$ accepts *(or* recognizes*) the term t (and write $t \in \mathcal{L}(\mathcal{A})$) if and only if $run_{\mathcal{A}}(t) \cap F \neq \emptyset$*

Example

Consider the previously defined automaton $\mathcal{A}_{list}$. Table 2.1 lists several examples of terms accepted by that automaton. Although the last one may seem unexpected, it definitely is a member of $\mathcal{L}(\mathcal{A}_{list})$. This brings us to one of the most important aspects of regular types:

> *Regular types (as opposed to the work by Rychlikowski and Truderung [RT01]) do not encode* parametric polymorphism*, i.e. one can define the type of «list of integers» or «list of strings», or even «list of list of integers»; however, it is not possible to define the type of a* generic list *which may hold items of exactly one type.*

2.2.2 Supplementary Definitions

In order to prove the correctness of our algorithm, we need one additional abstraction, which we call *generalized run*. In this subsection we limit our definitions to *complete* and *deterministic* automata.

A *generalized run* of an automaton is an extension of the traditional run in which we let an automaton run over non-ground terms given that there exists a *state valuation* mapping variables to the states of the automaton.

Definition 7 (State Valuation) *Let $\mathcal{A} = (Q, S, F)$ be a tree automaton. A partial function $\delta : V \to Q$ is called a* state valuation*. We say that δ'* extends *δ (denoted as $\delta \subseteq \delta'$) if and only if $\delta'(x) = \delta(x)$ for every x in the domain of δ.*

Definition 8 (Generalized Step) *Let $\mathcal{A} = (Q, S, F)$ be a complete, deterministic tree automaton and let $\delta : V \to Q$ be a* state valuation*. We define a* generalized step *function as:*

$$\begin{aligned} step_{\mathcal{A},\delta}\Big(f(q_1, \ldots, q_k)\Big) &= q \\ step_{\mathcal{A},\delta}(v) &= \delta(v) \end{aligned}$$

where $v \in V$, $f \in \Sigma$ is a k-ary function symbol and $f(q_1, \ldots, q_k) \to q \in S$.

Definition 9 (Generalized Run) *A* generalized run *of an automaton is a transitive closure of the generalized step function:*

$$run_{\mathcal{A},\delta}\Big(f(t_1, \ldots, t_k)\Big) = step_{\mathcal{A},\delta}\Big(f\Big(run_{\mathcal{A},\delta}(t_1), \ldots, run_{\mathcal{A},\delta}(t_k)\Big)\Big)$$

One may notice that generalized *step* and *run* functions may be undefined if $\delta(v)$ is not defined for some variable v in the term t. For this reason the following rule applies:

Whenever we write that $run_{\mathcal{A},\delta}(t) = q$, we restrict ourselves to those functions δ that are defined for every variable v in the term t.

Lemma 1 *If $run_{\mathcal{A},\delta}(t) = q$ and $\delta \subseteq \delta'$ then $run_{\mathcal{A},\delta'}(t) = q$.*

Proof

Let $\mathcal{A} = (Q, S, F)$. By induction over the structure of the term t:

1. Suppose t is a constant symbol g.
   $run_{\mathcal{A},\delta}(g) = q$ implies that $g \to q \in S$ which in turn implies that $run_{\mathcal{A},\delta'}(g) = q$.

2. Suppose t is a variable v
   $run_{\mathcal{A},\delta}(g) = q$ implies that $\delta(v) = q$. Furthermore, $\delta \subseteq \delta'$ implies $\delta'(v) = q$ and thus $run_{\mathcal{A},\delta'}(v) = \delta'(v) = q$.
3. Suppose t is a functor $f(t_1, \ldots, t_k)$ and $\forall 1 \leq i \leq k.run_{\mathcal{A},\delta}(t_i) = q_i$.
   $run_{\mathcal{A},\delta}(g) = q$ implies that $f(q_1, \ldots, q_k) \to q \in S$. By induction hypothesis, $\forall 1 \leq i \leq k.run_{\mathcal{A},\delta'}(t_i) = q_i$ hence $run_{\mathcal{A},\delta'}(t) = q$.

By induction theorem, $run_{\mathcal{A},\delta'}(t) = q$. ■

2.3 Directional Types

In this section we will present the definitions of the directional types. Below we identify *types* with *tree automata.*

Definition 10 (Type Judgment) *A* type judgment *is an implication*

$$t_1 : \mathcal{A}_1 \wedge \ldots \wedge t_n : \mathcal{A}_n \to t_0 : \mathcal{A}_0.$$

We say that such a judgment holds *if the implication*

$$t_1\theta \in \mathcal{L}(\mathcal{A}_1) \wedge \ldots \wedge t_n\theta \in \mathcal{L}(\mathcal{A}_n) \to t_0\theta \in \mathcal{L}(\mathcal{A}_0)$$

is true for all ground substitutions $\theta : V \to T_\Sigma$.

Definition 11 (Directional Type) *A* directional type

$$\mathcal{T} = (\mathcal{A}_p^I \to \mathcal{A}_p^O)_{p \in Pred}$$

assigns each predicate p a pair of an input and an output type.

Example

Consider the predicate `rev`. We have already mentioned one of its possible types:

$$(list, \top) \to (list, list)$$

For the above to be a directional type we need two tree automata:

1. An input type $\mathcal{A}_{\mathtt{rev}}^I$ which in our case is an automaton that accepts pairs of *list* and *anything*: $\mathcal{A}_{(list,\top)}$,
2. and an output type $\mathcal{A}_{\mathtt{rev}}^I$ which is an automaton that accepts pairs of lists: $\mathcal{A}_{(list,list)}$.

Definition 12 (Type Correctness) *A program is* well typed *with respect to the directional type* $\mathcal{T} = (\mathcal{A}^I_p \to \mathcal{A}^O_p)_{p \in Pred}$ *if the following type judgments hold for each clause* $p_0(t_0) :- p_1(t_1), \ldots, p_n(t_n)$.

$$\begin{array}{rcl} t_0 : \mathcal{A}^I_{p_0} & \to & t_1 : \mathcal{A}^I_{p_1} \\ t_0 : \mathcal{A}^I_{p_0} \wedge t_1 : \mathcal{A}^O_{p_1} & \to & t_2 : \mathcal{A}^I_{p_2} \\ & \vdots & \\ t_0 : \mathcal{A}^I_{p_0} \wedge t_1 : \mathcal{A}^O_{p_1} \wedge \ldots \wedge t_{n-1} : \mathcal{A}^O_{p_{n-1}} & \to & t_n : \mathcal{A}^I_{p_n} \\ t_0 : \mathcal{A}^I_{p_0} \wedge t_1 : \mathcal{A}^O_{p_1} \wedge \ldots \wedge t_n : \mathcal{A}^O_{p_n} & \to & t_n : \mathcal{A}^O_{p_n} \end{array}$$

Our definition of the *type correctness* expresses the type safety at every point of the left-to-right execution.

Example

Consider the second clause of the `rev` predicate (as presented in Table 1.1 on page 2):

```
rev([X|XS],Z) :- rev(XS,Y), append(Y,[X],Z).
```

In order for this clause to be *well-typed*, the following type judgments must hold:

$$\begin{array}{rcl} (\texttt{[X|XS]}, \texttt{Z}) : \mathcal{A}^I_{\texttt{rev}} & \to & (\texttt{XS}, \texttt{Y}) : \mathcal{A}^I_{\texttt{rev}} \\ (\texttt{[X|XS]}, \texttt{Z}) : \mathcal{A}^I_{\texttt{rev}} \wedge (\texttt{XS}, \texttt{Y}) : \mathcal{A}^O_{\texttt{rev}} & \to & (\texttt{Y}, \texttt{[X]}, \texttt{Z}) : \mathcal{A}^I_{\texttt{append}} \\ (\texttt{[X|XS]}, \texttt{Z}) : \mathcal{A}^I_{\texttt{rev}} \wedge (\texttt{XS}, \texttt{Y}) : \mathcal{A}^O_{\texttt{rev}} \wedge (\texttt{Y}, \texttt{[X]}, \texttt{Z}) : \mathcal{A}^O_{\texttt{append}} & \to & (\texttt{[X|XS]}, \texttt{Z}) : \mathcal{A}^O_{\texttt{rev}} \end{array}$$

Whether these judgments hold, depends on the definitions of the four automata: $\mathcal{A}^I_{\texttt{rev}}, \mathcal{A}^O_{\texttt{rev}}, \mathcal{A}^I_{\texttt{append}}$ and $\mathcal{A}^O_{\texttt{append}}$ or (in other words) on directional type $(\mathcal{A}^I_{\texttt{rev}} \to \mathcal{A}^O_{\texttt{rev}},\ \mathcal{A}^I_{\texttt{append}} \to \mathcal{A}^O_{\texttt{append}})$. Suppose that we want to verify correctness of the `rev` predicate with respect to the following types:

- `rev`: $\underbrace{(list, \top)}_{\mathcal{A}^I_{\texttt{rev}}} \to \underbrace{(list, list)}_{\mathcal{A}^O_{\texttt{rev}}}$,
- `append`: $\underbrace{(list, list, \top)}_{\mathcal{A}^I_{\texttt{append}}} \to \underbrace{(list, list, list)}_{\mathcal{A}^O_{\texttt{append}}}$.

The first judgment $([\mathtt{X}|\mathtt{XS}], \mathtt{Z}) : (\mathit{list}, \top) \rightarrow (\mathtt{XS}, \mathtt{Y}) : (\mathit{list}, \top)$ enforces that if `rev` is passed a correct term, it passes a correct term to the first predicate in the body of the clause (which in this example is a recursive call to `rev`). And since `Z` and `Y` may be anything, the type checker needs only to verify that if `[X|XS]` is a list, then so is `XS`. The verification procedure is described in detail in Chapter 3.

2.4 Summary

This chapter offered a description of logic programs, tree automata (and how they are related to types) as well as directional types. All these definitions will be utilized in the next chapter in order to introduce the type checking algorithm and prove its correctness.

Chapter 3

Type Checking

In this chapter we introduce the type checking procedure. Section 3.1 presents the list of steps that constitute the type checking algorithm and Sections 3.2, 3.3 and 3.4 describe them in detail. Finally, Section 3.5 summarizes the algorithm and offers an insight into the next chapters.

3.1 Overview

In order to check that the following clause

$$p_0(t_0) :- p_1(t_1), \ldots, p_n(t_n).$$

is well typed with respect to the directional type $(\mathcal{A}^I_{p_i} \rightarrow \mathcal{A}^O_{p_i})^n_{i=0}$, the type checker needs to take the following steps:

1. Convert the clause into $n+1$ clauses using the *magic-set transformation* (see Section 3.2).
2. For each of the resulting clauses take the following steps:
   (a) Create a *sharp-automaton* out of the type automata corresponding to the predicates in the clause (see Section 3.3).
   (b) Construct a *sharp term* out of the predicates' arguments and verify that there exists no ground substitution such that the resulting term is accepted by the sharp automaton (see Section 3.4).

3.2 Magic-Set Transformation

The *magic-set transformation* converts a clause into a set of clauses that describe the property of the original clause being well typed with respect

to an appropriate directional type. Following the Definition 12, every clause

$$p_0(t_0) :- p_1(t_1), \ldots, p_n(t_n).$$

is replaced with the following $n+1$ clauses:

$$\begin{array}{rcl} p_1^{in}(t_1) & :- & p_0^{in}(t_0) \\ p_2^{in}(t_2) & :- & p_0^{in}(t_0), p_1^{out}(t_1) \\ & \vdots & \\ p_n^{in}(t_n) & :- & p_0^{in}(t_0), p_1^{out}(t_1), \ldots, p_{n-1}^{out}(t_{n-1}) \\ p_0^{out}(t_0) & :- & p_0^{in}(t_0), p_1^{out}(t_1), \ldots, p_n^{out}(t_n) \end{array}$$

where p_i^{in} and p_i^{out} correspond to `in-` and `out-` automata describing types of the original clause.

3.3 Sharp Automaton

Consider the following clause c of the transformed program:

$$p_0(t_0) :- p_1(t_1), \ldots, p_n(t_n).$$

where every predicate p_i corresponds to a tree automaton $\mathcal{A}_i$. It is crucial to understand what it means for the program *not* to be well typed. This may only happen if there exists a substitution θ for variables in the clause such that:

$$\Big(\forall_{i=1}^{n}.t_i\theta \in \mathcal{L}(\mathcal{A}_i)\Big) \wedge \Big(t_0\theta \notin \mathcal{L}(\mathcal{A}_0)\Big)$$

In order to find θ or prove that there exists no such substitution we build a *sharp-automaton* which is bottom-up deterministic and recognizes the following language:

$$\mathcal{L}_{\#} = \Big\{\#(t_0, \ldots, t_n)\Big| t_0 \notin \mathcal{L}(\mathcal{A}_0) \wedge t_1 \in \mathcal{L}(\mathcal{A}_1) \wedge \ldots \wedge t_n \in \mathcal{L}(\mathcal{A}_n)\Big\}$$

where # is a fresh $(n+1)$-ary functor.

The sharp-automaton is constructed using the standard technique in automata theory [CDG+97], based on the power set construction for the determinization of tree automata. Let $\mathcal{A}_i = \{Q_i, S_i, F_i\}$. We describe the construction of the automaton $\mathcal{A}_{\#} = (Q_{\#}, S_{\#}, F_{\#})$:

- $Q_{\#} = \{p_R | R \in 2^{Q_0 \cup \ldots \cup Q_n}\} \cup \{p_{acc}\}$,

- $F_{\#} = \{p_{acc}\}$,
- $S_{\#}$ is constructed as follows:
  For every sequence $p_{R_1}, \ldots, p_{R_n} \in Q'$ and k-ary $f \in \Sigma$, $S_{\#}$ contains a transition $f(p_{R_1}, \ldots, p_{R_n}) \to p_R$ where:

$$R = \left\{ r \in \bigcup_{i=0}^{n} Q_i \middle| \exists r_1 \in R_1 \ldots \exists r_k \in R_k . f(r_1, \ldots, r_k) \to r \in \bigcup_{i=0}^{n} S_i \right\}$$

  Moreover, for every $(n+1)$-tuple $p_{R_0}, \ldots, p_{R_n} \in Q'$ such that

$$\left(R_0 \cap F_0 = \emptyset\right) \wedge \left(\forall_{i=1}^{n} . R_i \cap F_i \neq \emptyset\right)$$

  we add the transition $\#(p_{R_0}, \ldots, p_{R_n}) \to p_{acc}$.

A detailed description of this construction and the proof of its correctness may be found in [CP98].

3.4 Non-emptiness

Let us once again consider the clause c:

$$p_0(t_0) :- p_1(t_1), \ldots, p_n(t_n).$$

The source clause is well typed if and only if there exists no substitution θ for variables in the terms $t_0, \ldots, t_n$ such that $\#(t_0, \ldots, t_n)\theta \in \mathcal{L}(\mathcal{A}_{\#})$.

Since the automaton $\mathcal{A}_{\#}$ is bottom-up deterministic, finding a ground substitution is reduced to the problem of finding a *state valuation*, i.e. a mapping of variables to the reachable states of the sharp automaton so that it accepts the term. Such *state valuation* may be obtained by running the algorithm presented in Table 3.1.

Theorem 1 *Let $\mathcal{A}$ be a complete, deterministic tree automaton. The procedure* `Resolution`$(\mathcal{A}, t, q, \theta)$ *returns a set of* all *the valuations δ that satisfy the following conditions:*

1. $run_{\mathcal{A},\delta}(t) = q$,
2. $\theta \subseteq \delta$,
3. $Dom(\delta) \subseteq Dom(\theta) \cup Vars(t)$.

function `Resolution` (
  $\mathcal{A} = (Q, S, F)$: Deterministic Tree Automaton,
  t : Term,
  q : State,
  θ : Term $\rightarrow$ State
) : Set [Term $\rightarrow$ State]

1 **if** t **is** variable $v \in V$ **then**
2   **if not defined** $\theta(v)$ **then**
3   **return** $\{\theta_{v=q}\}$
4   **else if** $\theta(v) = q$ **then**
5   **return** $\{\theta\}$
6   **else**
7   **return** $\emptyset$
8   **end if**
9 **else if** t **is** functor $f(t_1, \ldots, t_k)$ **then**
10   result $\leftarrow \emptyset$
11   **for each** transition $f(q_1, \ldots, q_k) \rightarrow q \in S$ **do**
12   previous $\leftarrow \{\theta\}$
13   **for** $i = 1$ **to** k **do**
14   next $\leftarrow \emptyset$
15   **for each** $\theta' \in$ previous **do**
16   next $\leftarrow$ next $\cup$ `Resolution`$(\mathcal{A}, t_i, q_i, \theta')$
17   **end for each**
18   previous $\leftarrow$ next
19   **end for**
20   result $\leftarrow$ result $\cup$ previous
21   **end for each**
22   **return** result
23 **end if**

Table 3.1: Resolution algorithm.

Proof

Let $\mathcal{A} = (Q, S, F)$. We prove our theorem inductively over the structure of the term t.

1. Suppose that $t \in \Sigma$ is a constant symbol g.
   If $g \to q \in S$ (there may exist only one such transition because $\mathcal{A}$ is deterministic), then the resolution returns θ (notice that lines 13—19 are skipped because $k = 0$), otherwise it returns an empty set.
   Without a doubt, θ satisfies all three conditions of Theorem 1 and it is actually the only valuation that may satisfy them (due to the determinism of $\mathcal{A}$).
2. Suppose that $t \in V$ is a variable v.
   There are exactly three possibilities:
   (a) $\theta(v) \neq q$ (line 7). The resolution returns an empty set, which evidently satisfies our claims since there exists no valuation that would satisfy conditions 1 and 2.
   (b) $\theta(v) = q$ (line 5). The resolution returns θ, which obviously satisfies conditions 2, 3 as well as condition 1 since $run_{\mathcal{A},\theta}(v) = \theta(v) = q$.
   (c) $\theta(v)$ is undefined (line 3). The resolution returns θ extended with the mapping $v \to q$. Clearly, all three conditions of Theorem 1 are satisfied.
3. Now suppose that t is a functor $f(t_1, \ldots, t_k)$ and that the resolution returns correct results for the terms $t_1, \ldots, t_k$. We will prove that the resolution must also return the correct result for t.
   Let $f(q_1, \ldots, q_n) \to q \in S$. The loop between lines 13—19 defines a sequence $\Delta_0, \Delta_1, \ldots, \Delta_n$ of sets of state valuations such that:

$$\Delta_0 = \{\theta\} \tag{3.1}$$

$$\Delta_r = \bigcup_{\delta_{r-1} \in \Delta_{r-1}} \text{Resolution}(\mathcal{A}, t_r, q_r, \delta_{r-1}) \tag{3.2}$$

   For every $r = 1, \ldots, n$ easy induction shows the following conditions:

$$\forall \delta_r \in \Delta_r . run_{\mathcal{A},\delta_r}(t_r) = q_r \tag{3.3}$$

$$\forall \delta_r \in \Delta_r \exists \delta_{r-1} \in \Delta_{r-1} \begin{cases} \delta_r \subseteq \delta_{r-1} \\ and \\ Dom(\delta_r) \subseteq Dom(\delta_{r-1}) \cup Vars(t_{r-1}) \end{cases} \tag{3.4}$$

   We can even strengthen Equation 3.3 with the result of Lemma 1:

$$\forall \delta_r \in \Delta_r \forall 1 \leq j \leq r . run_{\mathcal{A},\delta_r}(t_j) = q_j \tag{3.5}$$

Eventually, the resolution returns the set Δ_n. We will demonstrate that every $\delta_n \in \Delta_n$ fulfils the conditions listed in Theorem 1:

(a) $run_{\mathcal{A},\delta_n}(t) = q$ is a consequence of 3.5 and the fact that $f(q_1, \ldots, q_n) \to q \in S$,

(b) $\theta \subseteq \delta_n$ is a direct consequence of 3.1 and transitivity of condition 3.4.

(c) $Dom(\delta_n) \subseteq Dom(\theta) \cup Vars(t)$ follows from the transitivity of condition 3.4.

By induction theorem, the resolution returns *all* valuations that satisfy conditions listed in Theorem 1. ■

Corollary 1 *Feeding the algorithm with the following arguments:*

1. $\mathcal{A}_\#$ *as* $\mathcal{A}$,
2. $\#(t_0, t_1, \ldots, t_n)$ *as* t,
3. p_{acc} *as* q,
4. *empty* θ,

will result in the list of all the possible state valuations *that violate type correctness.*

3.5 Summary

In this chapter we have presented the type checking algorithm and proven its correctness. A careful reader could have noticed that the construction of the *sharp automaton* in Section 3.3 is exponential. In order for the type checker to be effective enough, we had to choose an efficient data structure that would let us deal with immense tree automata. The next chapter presents such a compact representation – *binary decision diagrams.*

Chapter 4

Data Structures

Most of the computations of our type checker are performed on tree automata. In order to compress their size we have chosen *binary decision diagrams* [McM92] for the representation of their transition relation. Section 4.1 *outlines* how we have applied decision diagrams to states and transitions of tree automata. Section 4.2 presents an *example*, followed by Section 4.3 which lists the main *advantages* of decision diagrams. Section 4.4 presents *experiments* which we have conducted in order to evaluate the factual efficiency of diagrams. Finally Section 4.5 concludes with a short *summary* of the chapter.

4.1 Outline

Given an automaton we enumerate its individual states mapping them to consecutive integer numbers. A special state (called $\top$[1]) is used to represent the *sink state* of the automaton and is always assigned number 0. The automaton takes advantage of the binary representation of state numbers and stores transition relation as a set of *binary decision diagrams*. The diagrams represent a decision process which leads to the set of target states. Since this data structure is very complex we have prepared a simple example, which explains all the details.

4.2 Example

We have chosen a slightly modified version of $\mathcal{A}_{list}$ automaton described in Section 2.2.1 for an example. $\mathcal{A}_{list} = (Q, S, F)$:

[1] The type checker reserves the word `top` for this purpose. See Section A.2.2 for details.

- $Q = \{\top, \mathit{void}, \mathit{list}\}$
- $F = \{\mathit{list}\}$
- $S = \{$

$$\begin{aligned} \texttt{nil} &\rightarrow \mathit{list} \\ \texttt{unit} &\rightarrow \mathit{void} \\ \texttt{cons}(\top, \mathit{list}) &\rightarrow \mathit{list} \\ \texttt{cons}(\mathit{list}, \mathit{list}) &\rightarrow \mathit{list} \end{aligned}$$

$\}$

All the other transitions lead to the state $\top$, such that the automaton is complete and deterministic. First, the type checker enumerates the states. We present them in binary below:

$$\begin{aligned} \top &\rightarrow 00_2 \\ \mathit{void} &\rightarrow 01_2 \\ \mathit{list} &\rightarrow 10_2 \end{aligned}$$

Next, it builds an appropriate diagram for every functor $f \in \Sigma$. The inner-nodes of each diagram consist of two numbers identifying the argument (first) and its bit (second). The leaves of the diagram store target states of the transition relation. Suppose that while running over a term the automaton needs to determine the next step for $\texttt{cons}(\top, \mathit{list})$.

1. The automaton locates the diagram for the functor `cons`. The appropriate graph is presented in Figure 4.1.

2. Next, the arguments are transformed to binary numbers:

$$(\underbrace{00_2}_{\top}, \underbrace{10_2}_{\mathit{list}})$$

   Since the arithmetics is 0-based, the first argument becomes 0th one and the second one – 1st. The bits are enumerated in the same manner, starting from the least significant one.

3. Afterwards, the automaton traverses the diagram using the arguments to determine where to go in every subsequent step.

   (a) In the root the automaton needs to check the 0th bit of the 0th argument. It chooses the edge marked *low* because the value of the bit is zero.

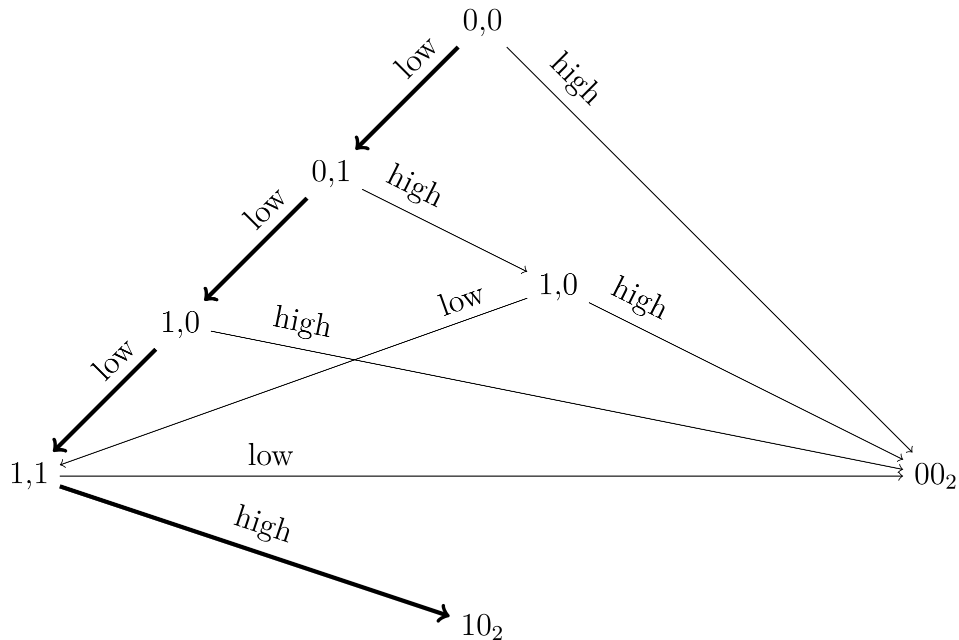

Figure 4.1: *Binary Decision Diagram* representing the transition relation for functor `cons` of the automaton $\mathcal{A}_{list}$.

(b) The next node is labeled $0, 1$ and so the automaton checks the 1st bit of the 0th argument. Again, it chooses the edge marked *low*.

(c) The third node is labeled $1, 0$. Once again, the 0th bit of the 1st argument is zero and the automaton chooses the *low* edge.

(d) Finally, the 1st bit of the 1st argument is one and the automaton chooses the edge labeled *high*, which leads to the leaf labeled with the state 10_2.

The above path has been marked in the Figure with thick arrows.

The automaton does not have to be deterministic – the leaves of the diagrams store lists of states, but it is always complete – the lists of states must not be empty.

4.3 Advantages

The main reasons why we have chosen this representation of the transition relation were:

1. The diagrams are reduced to *DAG*s using the linear time `reduce` procedure (see Appendix B.1 for the implementation) described by McMillan in his PhD Thesis [McM92]. Despite its simplicity, this heuristic proved to be very useful in the past. We presumed that it could compress the size of automata.

2. An important feature of the *binary decision diagrams* is that the states of an automaton are stored as arrays of bits – one of the most natural representations of characteristic functions. Since the states of the *sharp automaton* need to reflect the sets of states of original automata (see Section 3.3), we supposed that this representation could ease the implementation.

3. Last but not least, without any space penalty, the automaton becomes complete, which is an important precondition for it to be part of an input to the *sharp automaton* creation procedure.

All our three assumptions have been confirmed by the experiments described in the next section.

4.4 Efficiency Evaluation

In order to pre-evaluate the efficiency of our approach, we have compared the time it takes for two programs to determinize an automaton. One implementation (called further *BDD*) used the decision diagrams described in the previous section, while the other (called further *Naïve*) was based on the lists of states.

In our first experiment we have measured the influence of the maximal arity of a functor in the alphabet on the time needed to complete the determinization of a very simple automaton $\mathcal{A}_{list} = (Q, S, F)$:

- $Q = \{list, \top\}$
- $F = \{list\}$
- $S = \{$

$$\begin{aligned} \texttt{nil} &\rightarrow list \\ \texttt{cons}(\top, list) &\rightarrow list \end{aligned}$$

$\}$

symbol	arity
nil	0
cons	2
zero	0
succ	1
ala	n

Table 4.1: Alphabet Σ used in efficiency experiments.

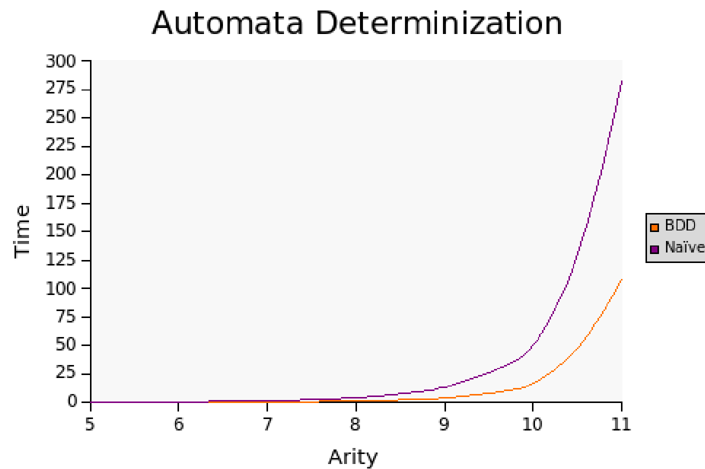

Figure 4.2: Relation between maximal arity of a functor and determinization time of the automaton $\mathcal{A}_{list}$.

Next, the automata have been completed to the alphabet Σ presented in Table 4.1, which consists of four ordinary functors (list and integer constructors) and one special with arity n – the experiment's parameter. The results are presented in Figure 4.2. Although BDD implementation seems slightly more efficient, the diagram does not show any clear evidence of its superiority over the Naïve one.

Despite the fact that the creation of the *sharp automaton* is very similar to automata determinization, it is not exactly the same. For this reason we have designed the second experiment that would closer reflect the nature of the actual task. The only thing we have changed in the setup of the experiment is that instead of determinizing the $\mathcal{A}_{list}$ automaton we determinize the automaton $\mathcal{A}_{list} \times \mathcal{A}_{int}$ where $\mathcal{A}_{int} = (Q, S, F)$ and:

- $Q = \{int, \top\}$
- $F = \{int\}$

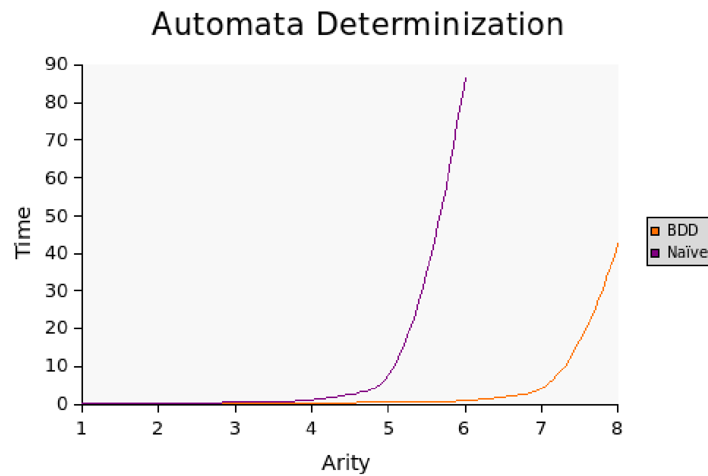

Figure 4.3: Relation between maximal arity of a functor and determinization time of the automaton $\mathcal{A}_{list} \times \mathcal{A}_{int}$.

- $S = \{$

$$\begin{array}{rcl} \texttt{zero} & \rightarrow & int \\ \texttt{succ}(int) & \rightarrow & int \end{array}$$

$\}$

Figure 4.3 presents the results of our second experiment. Evidently, the BDD implementation outperforms the Naïve one.

4.5 Summary

Even though the heuristics used in the BDD implementation of automata proved to be more efficient than the non-optimized Naïve solution, the exponential nature of the automata determinization problem does not let us expect any implementation to handle functors of arbitrary large arity. Fortunately, we have managed to move the boundary far enough to be able to deal with most of PROLOG programs as functors of arity 10 and more are very rare.

Chapter 5

Conclusions

We have implemented a type checker for regular directional types. Due to innovative implementation of underlying data structures we have achieved satisfactory efficiency without restricting expressiveness of the type system. Section 5.1 presents examples of programs that have been verified by our type checker. Section 5.2 documents one of the *limitations* of regular types and Section 5.3 presents *related work*.

5.1 Typed Programs

We have checked the type correctness of several predicates that have been assigned to students as exercises to the courses of Programming and Logic Programming at the Computer Science Department of the University of Wrocław.

append

```
append([],L,L).
append([X|XS],Y,[X|ZS]) :- append (XS,Y,ZS).
```

Valid Types	Invalid Types
$(list, list, \top) \to (list, list, list)$	$(\top, \top, \top) \to (list, list, list)$
$(\top, \top, list) \to (list, list, list)$	

rev

```
rev([],[]).
rev([X|XS],Z) :- rev(XS,Y), append(Y,[X],Z).
```

Valid Types	Invalid Types
$(list, list) \to (list, list)$	$(\top, list) \to (list, list)$
$(list, \top) \to (list, list)$	

member

```
member(X,[X,XS]).
member(X,[_,YS]) :- member(X,YS).
```

Valid Types	Invalid Types
$(\top, list) \rightarrow (\top, list)$	
$(list, list) \rightarrow (list, list)$	

add

```
add(X,zero,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

Valid Types	Invalid Types
$(int, int, \top) \rightarrow (int, int, int)$	$(\top, int, \top) \rightarrow (int, int, int)$
$(\top, \top, int) \rightarrow (int, int, int)$	

minus

```
minus(zero,_,zero).
minus(X,zero,X).
minus(s(X),s(Y),Z) :- minus(X,Y,Z).
```

Valid Types	Invalid Types
$(int, int, \top) \rightarrow (int, int, int)$	

mul

```
mul(X,zero,zero).
mul(X,s(Y),Z) :- mul(X,Y,W), add(X,W,Z).
```

Valid Types	Invalid Types
$(int, int, \top) \rightarrow (int, int, int)$	$(\top, int, int) \rightarrow (int, int, int)$

exp

```
exp(_,zero,s(zero)).
exp(X,s(Y),Z) :- exp(X,Y,T), mul(T,X,Z).
```

Valid Types	Invalid Types
$(int, int, \top) \rightarrow (int, int, int)$	

less

```
less(zero,s(_)).
less(s(X),s(Y)) :- sless(X,Y).
```

Valid Types	Invalid Types
$(int, int, \top) \rightarrow (int, int, int)$	

mod

```
mod(X,Y,X) :- less(X,Y).
mod(X,X,zero).
mod(X,Y,Z) :- minus(X,Y,T), mod(T,Y,Z).
```

Valid Types	Invalid Types
$(int, int, \top) \rightarrow (int, int, int)$	

length

```
length([],zero).
length([X|XS], s(Z)) :- length(XS,Z).
```

Valid Types	Invalid Types
$(\top, \top) \rightarrow (list, int)$	

sum

```
sum([],0).
sum([H],H).
sum([H1,H2|T],S) :- Z is H1 + H2, sum([Z|T],S).
```

Valid Types	Invalid Types
$(list[int], \top) \rightarrow (list[int], int)$	

qsort_div

```
qsort_div([],_,[],[]).
qsort_div([H|T],X,[H|A],B) :- H =< X, qsort_div(T,X,A,B).
qsort_div([H|T],X,A,[H|B]) :- H > X, qsort_div(T,X,A,B).
```

Valid Types	Invalid Types
$(list, \top, \top, \top) \rightarrow (list, \top, list, list)$	

qsort

```
qsort([],[]).
qsort([H|T],L) :-
        qsort_div(T,H,A,B),
        qsort(A,A1), qsort(B,B1),
        append(A1,[H|B1],L).
```

Valid Types	Invalid Types
$(list, \top) \rightarrow (list, list)$	

Although our type checker is able to verify the type correctness of many Prolog programs, the type system itself has several drawbacks. The next section presents one such limitation.

5.2 Limitations

Consider the predicate `rev`:

```
rev([],[]).
rev([X|XS],Z) :- rev(XS,Y), append(Y,[X],Z).
```

and its two calls:

```
rev([1,2,3],X)
rev(X,[1,2,3])
```

No matter which one we choose, a PROLOG interpreter will return the same result:

```
X = [3, 2, 1]
Yes
```

Unfortunately, our type checker is not able to reflect this property of the `rev` predicate. Let us analyze the following directional types:

$$\texttt{rev} \ : \ (\mathit{list}, \top) \to (\mathit{list}, \mathit{list}) \tag{5.1}$$

$$\texttt{rev} \ : \ (\top, \mathit{list}) \to (\mathit{list}, \mathit{list}) \tag{5.2}$$

Although they both seem reasonable, the `rev` predicate is correct only with respect to (5.1). Any attempt to verify the (5.2) type by the type checker will result in the following error message[1]:

```
Checking types...invalid.

Type checking failed for the clause:
in_rev (tuple2 (V_XS, V_Y)) <-- \
       in_rev (tuple2 (cons (V_X, V_XS), V_Z))

With the following counterexample:
V_Y = unit
V_X = nil
V_XS = unit
V_Z = nil
```

The type checker points the magic-set clause that caused the problem:

$$\texttt{rev}^{in}(\texttt{XS}, \texttt{Y}) :- \texttt{rev}^{in}([\texttt{X}|\texttt{XS}], \texttt{Z})$$

[1] The actual output may vary depending on the definition of types.

which means

$$\text{if} \quad ([\mathtt{X}|\mathtt{XS}], \mathtt{Z}) : (\top, \mathit{list}), \quad \text{then} \quad (\mathtt{XS}, \mathtt{Y}) : (\top, \mathit{list})$$

Obviously, there is nothing the type checker could reason about the type of the variable `Y` because it is first introduced in this call and thus its value may be anything.

5.3 Related Work

The PROLOG programming language was developed by Kowalski and Smoliar [KS82] around 1982. Since then it has been a subject of many researches and numerous textbooks have been devoted to this language (see [NM90] as an example).

The notion of *directional type* in logic programs has been introduced by Aiken and Lakshman [AL94], who presented an algorithm for checking the well-typedness of logic programs with respect to path-closed types. Their research has been extended by Charatonik and Podelski [CP98, Cha00], who addressed the open questions concerning the complexity of regular type checking. The present thesis provides an implementation of the type checking algorithm for general regular types, demonstrating that despite DEXPTIME-hardness of the problem, verification of the type correctness may still be feasible when the underlying data structures [McM92] are well chosen.

Another extension of *directional types* has been proposed by Rychlikowski and Truderung [RT01], who devised a system of *polymorphic directional types* and developed an appropriate type checker.

Appendix A

User Documentation

In this chapter we present the user documentation of the tools prepared for this thesis. It has been split into two sections: Section A.1 is a short tutorial and Section A.2 is a reference manual. The tutorial has been prepared for possibly broad audience, whereas the reference manual requires some understanding of the basic concepts like tree automata. This documentation may be found in the web at `http://sliwerski.net/dtc`.

A.1 Tutorial

In the following tutorial, we learn how to verify the correctness of the type of the `append` predicate.

A.1.1 Append

We start with the definition of the `append` predicate.

```
append ([], L, L).
append ([X|Xs], Y, [X|Zs]) :- append (Xs, Y, Zs).
```

This code will be first preprocessed to the following form:

```
append (tuple3 (nil, L, L)).
append (tuple3 (cons (X, Xs), Y, cons (X, Zs))) :-
    append (tuple3 (Xs, Y, Zs)).
```

This is due to the fact that the type checker accepts only unary predicates and that brackets are merely syntactic sugar used to denote list constructors: `cons` and `nil`. The preprocessor understands more extensions and is described in detail in Section A.2.1.

A.1.2 Interface

Since PROLOG operates exclusively on terms, there is no place for type annotations in the source file. Thus, the annotations have to be stored in a separate file with a `.pi`[1] extension. The annotation for the append predicate is very simple:

```
append : ain -> aout.
```

This simply says that the type checker should look into the type definitions and find appropriate states: `ain` and `aout`.

A.1.3 Type Definitions

Now we need to define the input and output types of the append predicate. First, we need to define which terms should be recognized as lists:

```
nil is list.
cons (top, list) is list.
```

As you can see in the example above, we say that `nil` should be recognized as list and `cons(x,y)` should also be recognized as list if `x` has been recognized to be anything (`top` is a special symbol to denote everything) and `y` has been recognized as list. Now it's time to uncover the type of our predicate.

```
tuple3 (list, list, top) is ain.
tuple3 (list, list, list) is aout.
```

Our claim is: whenever we feed append with two lists and something else, and the program terminates, we get three lists as a result.

A.1.4 Verification

In order to verify that this type is correct, simply save the examples in files: `append.pl`, `append.pi` and `typedefs.td` and run the type checker like this:

```
$ dtc typedefs.td append
Preprocessing append.pl...done.
Reading type definitions from typedefs.td...done.
Parsing /tmp/dtc-DrH1Yk.pl...done.
Parsing /tmp/dtc-DrH1Yk.pi...done.
Checking types...ok.
```

[1] Stands for *prolog interface.*

A.1.5 Determinism

As we know, `append` is not only used to append two lists:

```
append ([1, 2, 3], [4, 5, 6], X)
```

but also to split a list into two parts:

```
append (X, Y, [1, 2, 3, 4, 5, 6])
```

And there comes one of the features of our type checker: it accepts nondeterministic types too.

```
tuple3 (list, list, top) is ain.
tuple3 (top, top, list) is ain.
tuple3 (list, list, list) is aout.
```

In the example above we say: no matter whether you give two lists to append, or one list to split, you always end up with three lists. And since this type is valid, the type checker will accept it.

A.1.6 Counterexamples

Nondeterminism lets us define even more sophisticated types, like for example this one:

```
cons (top, olist) is olist.
cons (one, list) is olist.
zero is one.
```

The example above defines the type of the list with at least one occurrence of zero. We can now try to prove that `append` preserves the *minimum-one-element* property.

```
tuple3 (olist, list, top) is ain.
tuple3 (list, olist, top) is ain.
tuple3 (list, list, olist) is aout.
```

This says: if one of the arguments contains at least one occurrence of zero, the resulting list will contain at least one occurrence of zero as well. Although the reasoning is true, the type checker will fail:

```
$ dtc typedefs.td append
Preprocessing append.pl...done.
Reading type definitions from typedefs.td...done.
```

```
Parsing /tmp/dtc-CB5wPC.pl...done.
Parsing /tmp/dtc-CB5wPC.pi...done.
Checking types...invalid.

Type checking failed for the clause:
in_append (tuple3 (V_Xs, V_Y, V_Zs)) <-- \
in_append (tuple3 (cons (V_X, V_Xs), V_Y, cons (V_X, V_Zs)))

With the following counterexample:
V_Xs = nil
V_Y = nil
V_X = zero
V_Zs = cons (zero, nil)
```

The type checker was actually trying to prove that whenever you pass the correct term to the predicate, it will pass a correct term further, which does not have to be true in order for the type to be correct. As an example consider the following call:

```
append([zero],[],Z).
```

The bottommost recursive call to the `append` predicate will get two empty lists as arguments, which violates the type correctness according to Definition 12.

Notice that the type checker always produces a respective counterexample that violates type correctness, which eases program debugging.

A.2 Reference Manual

Identifiers (denoted below as `ID`) are composed of letters, digits and underscores and have to begin with a letter.

A.2.1 Preprocessor

The preprocessor understands a subset of PROLOG described with the following grammar:

```
<program> ::= <program> <clause>
            | <clause>
<clause> ::= <predicate> "."
           | <predicate> ":-" <predicate-alt> "."
<predicate-alt> ::= <predicate-alt> ";" <predicate-con>
```

```
                  | <predicate-con>
<predicate-con> ::= <predicate-con> "," <predicate-simple>
                  | <predicate-simple>
<predicate-simple> ::= <predicate>
                     | "(" <predicate-alt> ")"
<predicate> ::= ID "(" <term-list> ")"
              | ID
              | "!"
              | <term-relation>
<term-relation> ::= VAR "is" <term>
                  | VAR "=" <term>
                  | VAR ">" <term>
                  | VAR "<" <term>
                  | VAR "<=" <term>
                  | VAR ">=" <term>
<term-list> ::= <term-list> "," <term> | <term>
<compound-term> ::= <term>
                  | <term> "+" <term>
                  | <term> "-" <term>
<term> ::= VAR
         | "_"
         | ID
         | ID "(" <term-list> ")"
         | <list>
         | STRING
         | NUMBER
         | "(" <term-list> ")"
<list> ::= "[" <term-list> "|" <term> "]" |
           "[" <term-list> "]"
```

The preprocessor performs the following substitutions:

- Replaces list syntactic sugar with `cons` and `nil` constructors.
- Replaces numbers with the term `number`.
- Replaces strings with the term `string`.
- Adds `tupleX` constructors at the top of every predicate argument.
- Replaces underscores with fresh variables.
- Unfolds alternatives.

- Replaces infix operators with constructors:
  - `a + b` with `plus(a,b)`
  - `a - b` with `minus(a,b)`
- Replaces comparators with predicates:
  - `a = b` with `opeq(tuple2(a,b))`
  - `a < b` with `opsm(tuple2(a,b))`
  - `a > b` with `opgr(tuple2(a,b))`
  - `a <= b` with `opse(tuple2(a,b))`
  - `a >= b` with `opge(tuple2(a,b))`
- Ignores exclamations.

It is important to notice that type definitions may only use the core language, not the language understood by the preprocessor.

A.2.2 Type Definitions

Directional types are defined by tree automata. A file with type definitions contains only a list of transitions in the following format:

```
f(s1, s2,...,sn) is s0.
```

where `f` is an `n`-ary function symbol and `s0`,...,`sn` are states of the automaton. The following grammar has to be obeyed:

```
<transition-list> ::= <transition-list> <transition>
                    | <transition>
<transition> ::= <body> "is" ID "."
<body> ::= ID "(" <state-list> ")" | ID
<state-list> ::= <state-list> "," ID | ID
```

The set of states of the automaton is derived implicitly from the list of transitions. Additionally, the special state `top` is recognized by the parser as the sink-state. Formally speaking: if there exists no transition for some `n`-ary function symbol `f` and some states `s1,...,sn` then the automaton gets extended with the following transition:

```
f(s1,...,sn) is top.
```

A.2.3 Example

An example of a type definition:

```
nil is list.
cons (top, list) is list.
```

The above definition describes a tree automaton with the states `top` and `list`. The following terms are examples of terms recognized by the tree automaton with the accepting state list:

```
nil
cons (nil, nil)
cons (zero, cons (zero, nil))
cons (nil, cons (zero, nil))
```

One of the most important aspects of regular types is that there is no possibility of encoding parametric polymorphism (also known as generic types).

A.2.4 Type Annotations

Type annotations need to be given in a separate file so you do not have to modify your source files. They have to respect the following grammar:

```
<type-list> ::= <type-list> <type>
<type> ::= ID ":" ID "->" ID "."
```

Type annotations define the accepting states of the automata for input and output types of the predicates.

Appendix B

Implementation Details

The type checker has been implemented in the Nemerle programming language [MSO]. The complete program spans across 22 source files and consists of more than 7000 lines of code. This chapter previews several data structures and crucial algorithms. Section B.1 describes the implementation of the transition relation of our tree automata presented in Chapter 4. Section B.2 contains the implementation of the `Resolution` algorithm introduced in Section 3.4. Finally Section B.3 is devoted to the program used for efficiency evaluation described in Section 4.4. The complete solution may be downloaded from `http://sliwerski.net/dtc`.

B.1 Binary Decision Diagrams

Below we present the actual implementation of the *directed acyclic graph* together with the `Reduce` procedure. The *decision diagrams* in the form of *DAG*s are used to represent the transition relation in our implementation of tree automaton.

```
public variant DAG
{
    | Leaf
      {
          states : list [int];
      }
    | Node
      {
          arg  : int;
          bit  : int;
          high : DAG;
```

```
      low  : DAG;
    }

  [Nemerle.OverrideObjectEquals]
  public Equals (dag : DAG) : bool
  {
      | Leaf (dl) =>
        match (this){
            | Leaf (tl) => dl.Equals (tl)
            | _ => false
        }
      | Node (da, db, dh, dl) =>
        match (this){
            | Node (ta, tb, th, tl) =>
              ta == da && tb == db
              && object.ReferenceEquals (th, dh)
              && object.ReferenceEquals (tl, dl)
            | _ => false
        }
  }

  public GetReference () : int
  {
      base.GetHashCode ()
  }

  public override GetHashCode () : int
  {
      match (this){
          | Leaf ([s]) => 36 %^ s
          | Leaf (l) => 35 %^ l.GetHashCode ()
          | Node (a, b, h, l) =>
            a %^ b %^ h.GetReference () %^ l.GetReference ()
      }
  }

  public Reduce () : DAG
  {
      def nodes = Hashtable () : Hashtable [DAG, DAG];

      def traverse (dag : DAG) : DAG
```

```
        {
            | Leaf =>
              match (nodes.Get (dag)){
                  | Some (l) => l
                  | None =>
                    nodes.Add (dag, dag);
                    dag
              }
            | Node (a, b, high, low) =>
              def nh = traverse (high);
              def nl = traverse (low);
              if (object.ReferenceEquals (nh, nl)){
                  nl
              } else {
                  def node = Node (a, b, nh, nl);
                  match (nodes.Get (node)){
                      | Some (n) => n
                      | None =>
                        nodes.Add (node, node);
                        node
                  }
              }
        }

        traverse (this)
    }
}
```

B.2 Resolution

Below we present the actual implementation of the `Resolution` algorithm. We use *association lists* to represent *state valuations*.

```
private Resolution (term   : Term,
                    state  : int,
                    substs : list [string * int])
        : list [list [string * int]]
{

    def iter (terms, args, valuations)
```

```
    : list [list [string * int]]
{
    match ((terms, args)){
        | ([], []) => [valuations]
        | (x :: xs, y :: ys) =>
          def nsub = Resolution (x, y, valuations);
          nsub.FoldLeft ([], fun (sub, acc) {
               acc.RevAppend (iter (xs, ys, sub))
          })
        | _ =>
          throw System.ArgumentException (
              "List lengths don't match."
          );
    }
}

def loop (acc,
          pos : list [list [int]],
          terms : list [Term],
          valuations)
{
    match (pos){
        | [] => acc
        | x :: xs =>
          loop (acc.RevAppend (iter (terms, x,
                                         valuations)),
                xs, terms, valuations)
    }
}

match (term){
    | Variable (v) =>
      match (List.Assoc (substs, v)){
          | None =>
            [(v, state) :: substs]
          | Some (s) =>
            if (state.Equals (s)){
                [substs]
            } else {
                []
            }
```

```
          }
        | Functor (head, args) =>
          def reachable = automaton.ReachableStates ();
          def pos = Combinatorics.SizeSubsets (reachable,
                                               args.Length);
          def dag = automaton.transitions[head];
          def filtered = pos.Filter (fun (l) {
              EndsOK (state, dag, l)
          });
          def reversed = filtered.Map (fun (l) { l.Rev () });
          loop ([], reversed, args.Rev (), substs);
    }
}
```

B.3 Efficiency Evaluation

Below we present the actual implementation of the program we have used for experiments described in Section 4.4.

```
public module Benchmark
{
    private mutable arity : int;

    private MakeAlphabet () : Alphabet
    {
        def a = Alphabet ();

        a.Add (Symbol ("nil", 0));
        a.Add (Symbol ("cons", 2));
        a.Add (Symbol ("zero", 0));
        a.Add (Symbol ("s", 1));
        a.Add (Symbol ("ala", arity));

        a
    }

    private MakeListAutomaton () : TreeAutomaton * DAGAutomaton
    {
        def slist = State ("list");
        def stop  = State ("top");
```

```
    def trans1 = Transition ("nil", [], slist);
    def trans2 = Transition ("cons", [stop, slist], slist);

    def ta = TreeAutomaton([slist, stop], [trans1, trans2],
                           [slist]);

    def alphabet = MakeAlphabet ();

    (ta.MakeComplete (alphabet, stop),
     DAGAutomaton (ta, stop, alphabet))
}

private MakeIntAutomaton () : TreeAutomaton * DAGAutomaton
{
    def sint = State ("int");

    def trans1 = Transition ("zero", [], sint);
    def trans2 = Transition ("s", [sint], sint);

    def ta = TreeAutomaton ([sint], [trans1, trans2],
                            [sint]);

    def alphabet = MakeAlphabet ();

    (ta.MakeComplete (alphabet),
     DAGAutomaton (ta, alphabet))
}

public Main (args : array[string]) : void
{
    arity = int.Parse(args[0]);

    def (lta, lda) = MakeListAutomaton ();
    def (ita, ida) = MakeIntAutomaton ();

    if (bool.Parse(args[1])){
        _ = (ita * lta).Determinize ();
    } else {
        _ = (ida * lda).Determinize ();
    }
```

```
    }
}
```

Bibliography

[AL94] Alexander Aiken and T. K. Lakshman. Directional type checking of logic programs. In Baudouin Le Charlier, editor, *SAS*, volume 864 of *Lecture Notes in Computer Science*, pages 43–60. Springer, 1994.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997.

[Cha00] Witold Charatonik. Directional type checking: Beyond discriminative types. In *Proceedings of the European Symposium on Programming*, pages 72–87, 2000.

[CP98] Witold Charatonik and Andreas Podelski. Directional type inference for logic programs. In *Proceedings of the Fifth International Static Analysis Symposium*, pages 278–294, Pisa, Italy, September 1998.

[KS82] Robert Kowalski and Steve Smoliar. Logic for problem solving. *SIGSOFT Softw. Eng. Notes*, 7(2):61–62, 1982.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, May 1992.

[MSO] Michał Moskal, Kamil Skalski, and Paweł Olszta. Nemerle programming language. Available on: `http://nemerle.org`.

[NM90] Ulf Nilsson and Jan Małuszyński. *Logic, Programming and Prolog.* John Wiley & Sons Ltd., 1990.

[RT01] Pawel Rychlikowski and Tomasz Truderung. Polymorphic directional types for logic programming. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and*

practice of declarative programming, pages 61–72, Florence, Italy, September 2001. ACM.