FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Description Logic Based Reasoning on Programming Languages

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Rahmen des Erasmus-Mundus-Studiums

Computational Logic

eingereicht von

Ronald de Haan
Matrikelnummer 1128130

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dr. Bernhard Gramlich
Mitwirkung: Dr. Mikhail Roshchin

Wien, 28.09.2012

(Unterschrift Verfasser)

(Unterschrift Betreuung)

TU WIEN

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Description Logic Based Reasoning on Programming Languages

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science (M.Sc.)

in

Computational Logic

by

Ronald de Haan
Registration Number 1128130

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ. Prof. Dr. Bernhard Gramlich
External advisor: Dr. Mikhail Roshchin

Vienna, 28.09.2012

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ronald de Haan
Tivoligasse 7-9/2/11, 1120 Wien, Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

The people that have helped make this thesis possible shall not go unmentioned. I would like to take this chance to perform a speech act expressing gratitude towards all those people who have helped create the environment in which I wrote this thesis.

I would like to thank my supervisor, Bernhard Gramlich, for his feedback and comments. His directions were invaluable in the development of the ideas in this thesis. Thanks also go to Mikhail Roshchin, with whom I worked together at Siemens. He introduced me to a whole new side of theoretical computer science, namely the applied side. His ideas and comments have also proved extremely beneficial for the work in this thesis.

Last, but certainly not least, I would like to extend my thanks to my friends, family, class mates, and all the other people that have made my life during the last two years truly great. Thanks to all of you in Dresden, Bozen-Bolzano, Munich, Sydney, Vienna, the Netherlands, and everywhere else. Thanks for all the beers, all the laughs, all the discussions, all the jokes, all the travelling, and everything else.

Abstract

Imperative programming languages are ubiquitous in virtually all fields of technology, with programs specifying all sorts of computational behavior. For many practical reasons, an automated analysis of semantic properties of programs, such as termination and equivalence, is desirable. We provide a new approach to the automated semantic analysis of programs by encoding their behavior into formal logic. We consider a few syntactically simple imperative programming languages, and we encode programs of these languages into expressions of the description logic $\mathcal{ALC}(\mathcal{D})$ for a particular domain $\mathcal{D}$. We do this in such a way that models of these encodings correspond to executions of the source programs. In other words, essentially, we assign a model-theoretic semantics to imperative programs. This encoding makes it possible to express semantic properties of programs (most notably termination and equivalence) in the formal logic language. Effectively, in this fashion, we reduce reasoning problems defined on the programs to description logic reasoning. Practically, this directly results in algorithms to perform automated reasoning on a number of restricted fragments of the programming languages (i.e. loop-free programs, or programs restricted to a finite numerical domain). Theoretically, our approach makes it possible to identify further, less restricted fragments of the programming languages for which certain reasoning tasks are decidable. We identify one such fragment, based on finite partitionings of the state space, and illustrate what class of programs belongs to this fragment.

Kurzfassung

Imperative Programmiersprachen sind in praktisch allen technologischen Bereichen verbreitet, wobei Programme verschiedene Arten von Berechnungen spezifizieren. Für viele praktische Zwecke ist die automatisierte Analyse semantischer Eigenschaften von Programmen, wie die Terminierung und Äquivalenz, nützlich. Wir stellen ein neues Vorgehen zur automatisierten semantischen Analyse von Programmen bereit durch die Kodierung ihres Verhaltens in die formale Logik. Wir betrachten einige syntaktisch einfache, imperative Programmiersprachen, und wir kodieren Programme dieser Sprachen in Ausdrücke der Beschreibungslogik $\mathcal{ALC}(\mathcal{D})$, für einen bestimmten Bereich $\mathcal{D}$. Wir machen das in einer Weise, in der Modelle dieser Kodierungen den Durchführungen der Programme entsprechen. Mit anderen Worten, wir weisen imperativen Programmen eine modelltheoretische Semantik zu. Diese Kodierung ermöglicht es semantische Eigenschaften von Programmen (vor allem Terminierung und Äquivalenz) in der Sprache der formalen Logik auszudrücken. Auf diese Weise reduzieren wir das Schlussfolgern diverser semantischer Eigenschaften von Programmen zu Reasoning-Verfahren der Beschreibungslogik. In praktischer Hinsicht führt dieses Vorgehen direkt zu Algorithmen für das automatisierte Schlussfolgern für einige Fragmente der Programmiersprachen (d.h. Zyklusfreie Programme oder Programme beschränkt auf endliche numerische Bereiche). In theoretischer Hinsicht ermöglicht das Vorgehen weitere, weniger eingeschränkte Fragmente der Programmiersprachen zu identifizieren, wofür einige Aufgaben des Schlussfolgerns entscheidbar sind. Wir identifizieren Eins solcher Fragmente, welches auf endliche Aufteilungen des Zustandsraums basiert, und wir illustrieren, welche Klasse von Programmen zu diesem Fragment gehört.

Contents

CHAPTER 1

Introduction

1.1 Problem Description

Programming languages are formal languages designed to specify instructions for a computer to perform a particular computation. Programs, which are statements in these programming languages, thus are recipes for computations. The main semantic property of such programs is the input-output relation (i.e. computing the output of the computation specified by the program, based on the inputs given). Programming languages are designed to allow for efficient evaluation of this property. However, in many cases it is very useful to consider other semantic properties of programs and semantic relations between different programs. For instance, knowing whether a program terminates on all inputs, or knowing whether two programs give exactly the same output on all possible inputs, are very useful pieces of knowledge, both theoretically and practically. However, the general question of deciding semantic properties of programs is not restricted to such canonical questions. To give an example of a more intricate semantic property of programs, it might also be useful to be able to decide whether for a program, if it is presented with the value of input variable x of at least 10, will terminate with the value of output variable y at most twice the value of output variable z.

Unfortunately, in the general case, we know that many such semantic properties are undecidable. For instance, the halting problem – the problem whether a given program in a Turing-complete computational model (i.e. a programming language that is general enough to be equivalent to Turing machines) terminates – is well-known to be undecidable. Also, many semantic properties of programs and many semantic relations between programs, in the general case, can be reduced to the undecidable question whether a Diophantine equation has an integer solution – the decidability of which is the topic of Hilbert's famous tenth problem. However, there are a number of different ways in which we can restrict the programming languages so that the reasoning problems corresponding to these semantic properties and relations become decidable.

In order to be able to algorithmically decide such properties for these restricted fragments of programming languages, the semantics of programming languages needs to be formalized. A lot of research has been done investigating the formal semantics of programming languages (cf. [19, 20]). This has resulted in a number of different approaches to defining semantics of programs (e.g. operational semantics, denotational semantics, axiomatic semantics). However, the possibilities of automatically and algorithmically deciding semantic properties of programs of general-purpose programming languages remain largely unexplored.

In this thesis, we will investigate how a model-theoretic semantics can be assigned to programming languages, and how this can be used to perform automated reasoning over programs of the programming languages. In order to do so, concretely, we aim to achieve the following three goals. Our first goal is to devise a way to assign to programs of programming languages a model-theoretic semantics. Of course, this has to be done in a sensible way, i.e. the model-theoretic semantics needs to correspond to previously formalized semantics of the programming languages. Secondly, we set out to achieve the goal of using this model-theoretic semantics to automatically decide certain semantic properties of programs (such as termination and equivalence of programs), for restricted fragments of the programming languages that are already known to allow these semantic properties to be decided. We will show how existing algorithms (based on the model-theoretic semantics) can be used to decide such semantic properties. The third goal we aim to achieve is to use the model-theoretic semantics to identify previously unidentified (non-trivial) fragments of programming languages that allow the aforementioned semantic properties to be decided algorithmically. In addition, for the identified fragments with this property, we give an indication of the computational complexity of deciding these semantic properties.

1.2 Methodology

In order to achieve our goals, we employ the following methodology. Firstly, we consider a number of several simple programming languages, representing various programming paradigms (imperative programming, logic programming, functional programming). For these programming languages, we consider formal semantics as defined previously in the literature (e.g. for imperative languages we consider the operational semantics). We will encode programs of these programming languages into expressions of formal logic, in such a way that the semantic properties of the programs correspond to model theoretic properties of the logic expressions. In addition, we will specify how relevant semantic properties of programs can be expressed in the formal logic, using the encoding of programs into this logic. In this way, we are able to reduce the reasoning problems on programs of the programming languages (i.e. deciding the relevant semantic properties of programs) to reasoning problems in the model-theoretic semantics of the formal logic.

The formal logic we will make use of, is the description logic $\mathcal{ALC}(\mathcal{D})$, which consists of the prototypical description logic $\mathcal{ALC}$ extended with concrete domains. In short, description logics are formal logics designed to allow for decidable reasoning algorithms with high efficiency, in combination with high expressive power. Since we will consider programming languages that operate on concrete domains (i.e. numerical values), we need a logic whose expressive power includes concrete domains (to represent the relation between programs and concrete values), as well as the usual abstract domain (to represent the complex internal structure of programs). The description logic $\mathcal{ALC}(\mathcal{D})$ offers enough expressivity to be a suitable target language of our encoding.

1.3 Structure of the Thesis

The thesis is structured as follows. We start with the formal definition of the syntax and semantics of the description logic $\mathcal{ALC}(\mathcal{D})$ in Chapter 2, as well as some background on the working of available reasoning algorithms for this logic and related logics. Then, in Chapter 3, we formally define the syntax and semantics of a number of simple, representative programming languages we consider in the thesis. These include imperative programming languages, as well as a functional and a logic programming language. In addition, in Chapter 3, we determine a number of semantic properties of programs that we focus on in the remainder of the thesis. With all preliminary definitions in place, we can start with the technical work. In Chapter 4, we specify how arbitrary programs of the programming languages defined before are encoded into TBoxes of the description logic $\mathcal{ALC}(\mathcal{D})$. Along with specifying these encodings, we also prove the exact correspondence between the semantics of the programs and the model theoretic semantics of their encodings into $\mathcal{ALC}(\mathcal{D})$. Furthermore, in this chapter, we illustrate how a number of reasoning problems, by means of these encodings, can be flexibly specified using the expressive power of the description logic. After these first technical results, we elaborate on the benefits of this technique of encoding programs into logic, in Chapter 5. Finally, with the framework in place, we are ready to investigate the decidability and complexity of fragments of the programming languages by means of this framework, which we will do in Chapter 6. This entire endeavour is motivated by a real-world, industrial case of a domain-specific imperative programming language, for which automated reasoning algorithms are desirable. In Chapter 7, we elaborate on this motivating instance of the general problem and an implementation of reasoning algorithms developed on the basis of the framework developed in this thesis.[1] In Chapter 8, finally, we draw our conclusions and suggest directions for further research.

The general outline of the thesis, as described above, is depicted graphically in Figure 1.1. The boxes in the diagram represent the different chapters, and the arrows between these boxes represent a dependency relation between the content of the chapters.

[1] The use cases that offer a practical motivation are provided by Siemens AG, in München. Also, Siemens AG has helped make the research in this thesis possible by providing financial support.

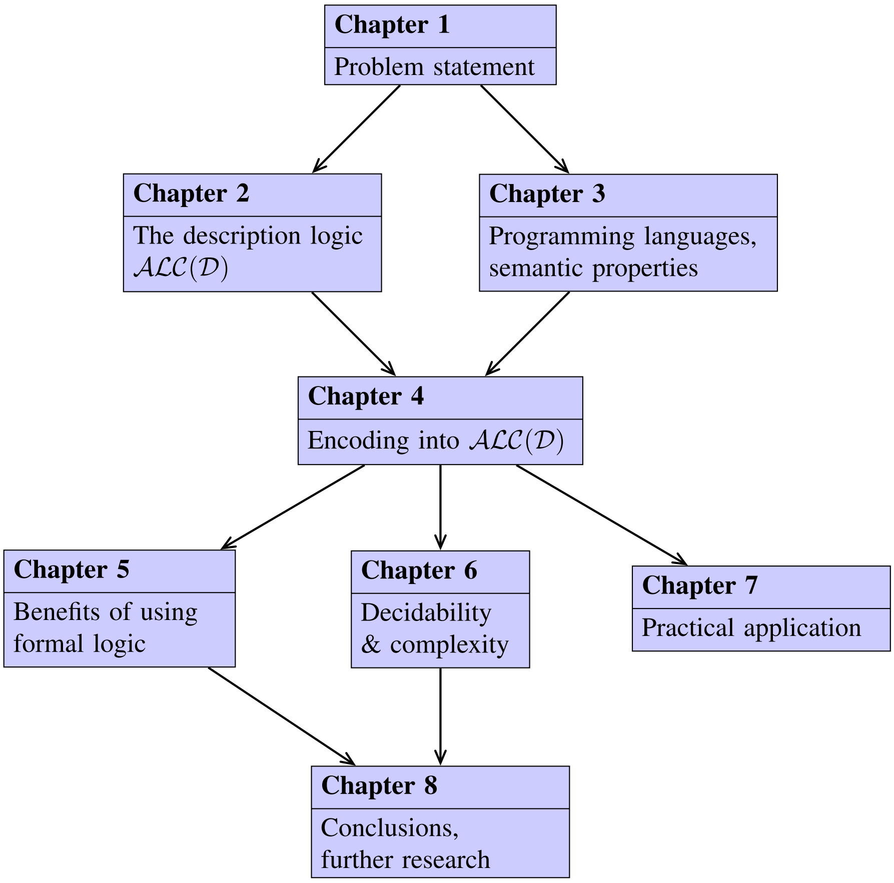
Chapter 1
Problem statement
Chapter 2
The description logic $\mathcal{ALC}(\mathcal{D})$
Chapter 3
Programming languages, semantic properties
Chapter 4
Encoding into $\mathcal{ALC}(\mathcal{D})$
Chapter 5
Benefits of using formal logic
Chapter 6
Decidability & complexity
Chapter 7
Practical application
Chapter 8
Conclusions, further research

Figure 1.1: Graphical representation of the outline of the thesis. The arrows represent dependency relations between the chapters.

CHAPTER 2

Preliminaries

In this chapter, we repeat the formal definitions of the syntax and semantics of the description logic $\mathcal{ALC}(\mathcal{D})$, which is the basic description logic $\mathcal{ALC}$ extended with concrete domains. Furthermore, we will give some general background on (the working of) tableau algorithms for description logics, and in particular for the logic $\mathcal{ALC}(\mathcal{D})$. These tableau based algorithms are the most often used algorithm to solve reasoning problems for description logics such as $\mathcal{ALC}(\mathcal{D})$.

2.1 The Description Logic $\mathcal{ALC}(\mathcal{D})$

We repeat some basic definitions about the description logic $\mathcal{ALC}(\mathcal{D})$. For more details on this particular description logic, see [16]. Description logics are a family of formal logics, often used to reason about various application domains. Its main ingredients are concepts and roles, which intuitively represent categories of objects and relations between objects, respectively. What makes description logics particularly useful for these purposes is that they are more expressive than propositional logic and that the decision problems associated with reasoning are of a lower complexity than full first-order logic (and in many cases, reasoning problems for description logics are decidable whereas similar problems for first-order logic are undecidable). Description logic knowledge bases usually consist of two types of knowledge: terminological knowledge (stored in a TBox) and assertional knowledge (stored in an ABox). Terminological knowledge expresses general relations between the different concepts and roles in the knowledge base. Assertional knowledge describes concrete individual objects, and the relation between these objects and the concepts and roles in the knowledge base.

The description logic $\mathcal{ALC}$ [21] is a prototypical description logic, and is at the basis of many more expressive description logics. In fact, the description logic $\mathcal{ALC}(\mathcal{D})$, that we consider in this thesis, is the extension of $\mathcal{ALC}$ with concrete domains (such as numerical values). We give a formal definition of the syntax and semantics of $\mathcal{ALC}(\mathcal{D})$, but before we can do so, we must firstly formally define what (admissible) concrete domains are.

Definition 2.1.1 (Concrete domains)**.** *A* concrete domain $\mathcal{D}$ *is a pair* $(\Delta_\mathcal{D}, \Phi_\mathcal{D})$*, where* $\Delta_\mathcal{D}$ *is a set and* $\Phi_\mathcal{D}$ *a set of predicate names. Each predicate name* $P \in \Phi_\mathcal{D}$ *is associated with an arity* n *and an* n*-ary predicate* $P^\mathcal{D} \subseteq \Delta_\mathcal{D}^n$*. Let* V *be a set of variables. A predicate conjunction of the form*

$$c = \bigwedge_{i \leq k} (x_0^{(i)}, \ldots, x_{n_i}^{(i)}) : P_i,$$

where P_i *is an* n_i*-ary predicate for* $i \leq k$ *and the* $x_j^{(i)}$ *are variables from* V*, is called satisfiable iff there exists a function* δ *mapping the variables in* c *to elements of* $\Delta_\mathcal{D}$ *such that* $(\delta(x_0^{(i)}), \ldots, \delta(x_{n_i}^{(i)})) \in P_i^\mathcal{D}$ *for* $i \leq k$*. Such a function is called a* solution *for* c*. A concrete domain is called* admissible *iff*

1. *its set of predicate names is closed under negation and contains a name* $\top_\mathcal{D}$ *for* $\Delta_\mathcal{D}$*, and*

2. *the satisfiability problem for finite conjunctions of predicates is decidable.*

By $\overline{P}$*, we denote the name for the negation of the predicate* P*, i.e., if the arity of* P *is* n*, then* $\overline{P}^\mathcal{D} = \Delta_\mathcal{D}^n \backslash P^\mathcal{D}$*.*

In particular, we consider the admissible concrete domain of linear arithmetic over the natural numbers.

Example 2.1.2. *Let* $\mathcal{N}_{lin} = (\mathbb{N}, \Phi)$ *be the concrete domain of linear arithmetic over the natural numbers, where* $\mathbb{N}$ *denotes the set of natural numbers, and* Φ *consists of the following predicates:*

$$\begin{array}{rcll}
\top & = & \mathbb{N} & \\
\bot & = & \emptyset & \\
\rho_n & = & \{x \in \mathbb{N} \mid x\ \rho\ n\} & \textit{for } \rho \in \{<, \leq, =, \neq, \geq, >\} \textit{ and } n \in \mathbb{N} \\
\rho & = & \{(x, y) \in \mathbb{N}^2 \mid x\ \rho\ y\} & \textit{for } \rho \in \{<, \leq, =, \neq, \geq, >\} \\
\pi & = & \{(x, y, z) \in \mathbb{N}^3 \mid x \pi y = z\} & \textit{for } \pi \in \{+, -\} \\
\overline{\pi} & = & \{(x, y, z) \in \mathbb{N}^3 \mid x \pi y \neq z\} & \textit{for } \pi \in \{+, -\} \\
\cdot_n & = & \{(x, y) \in \mathbb{N}^2 \mid x \cdot n = y\} & \\
\overline{\cdot_n} & = & \{(x, y) \in \mathbb{N}^2 \mid x \cdot n \neq y\} &
\end{array}$$

It can straightforwardly be verified that this concrete domain is admissible.

Another example of an admissible concrete domain we could consider is the restriction $\mathcal{N}_{lin}^{\leq k}$ of the previous concrete domain $\mathcal{N}_{lin}$ to a finite number of numbers $\{0, \ldots, k\}$ for some $k \in \mathbb{N}$. We define $\mathcal{N}_{lin}^{\leq k} = (\mathbb{N}^{\leq k}, \Phi^{\leq k})$, where $\mathbb{N}^{\leq k} = \{0, \ldots, k\}$ and $\Phi^{\leq k}$ is the restriction of Φ to $\mathcal{N}^{\leq k}$. In fact, also concrete domains representing non-linear arithmetic, when restricted to a finite number of values, are admissible.

With this notion of concrete domains in place, we are ready to formally define the syntax of the logic $\mathcal{ALC}(\mathcal{D})$, and afterwards define its formal, model-theoretic semantics.

Definition 2.1.3 ($\mathcal{ALC}(\mathcal{D})$ syntax)**.** *Let* $\mathsf{N_C}$ *and* $\mathsf{N_R}$ *be disjoint and countably infinite sets of concept and role names. Let* $\mathsf{N_{aF}}$ *be a countably infinite subset of* $\mathsf{N_R}$ *such that* $\mathsf{N_R} \backslash \mathsf{N_{aF}}$ *is also countably infinite. Elements of* $\mathsf{N_{aF}}$ *are called* abstract features*. Let* $\mathsf{N_{cF}}$ *be a countably infinite set of* concrete features *such that* $\mathsf{N_R} \cap \mathsf{N_{cF}} = \emptyset$ *and* $\mathsf{N_C} \cap \mathsf{N_{cF}} = \emptyset$*. A concrete path is a sequence* $f_1 \ldots f_n g$*, where* $f_1, \ldots, f_g \in \mathsf{N_{aF}}$ *and* $g \in \mathsf{N_{cF}}$*. Let* $\mathcal{D} = (\Delta_\mathcal{D}, \Phi_\mathcal{D})$ *be a concrete domain.*

Concepts *The set of $\mathcal{ALC}(\mathcal{D})$ concepts is the smallest set such that:*

1. *every concept name $A \in \mathsf{N_C}$ is an $\mathcal{ALC}(\mathcal{D})$ concept,*

2. *if C and D are $\mathcal{ALC}(\mathcal{D})$ concepts and $R \in \mathsf{N_R}$, then $\neg C$, $C \sqcap D$, $C \sqcup D$, $\exists R.C$, and $\forall R.C$ are $\mathcal{ALC}(\mathcal{D})$ concepts, and*

3. *if $g \in \mathsf{N_{cF}}$, $u_1, \ldots, u_n$ are concrete paths, and $P \in \Phi^{\mathcal{D}}$ is a predicate with arity n, then $\exists u_1, \ldots, u_n.P$ and $g\uparrow$ are $\mathcal{ALC}(\mathcal{D})$ concepts.*

We use $\top$ as an abbreviation for some fixed propositional tautology such as $A \sqcup \neg A$, $\bot$ for $\neg\top$, $C \rightarrow D$ for $\neg C \sqcup D$, and $C \leftrightarrow D$ for $(C \rightarrow D) \sqcap (D \rightarrow C)$. We use $u\uparrow$ as an abbreviation for $\forall f_1 \ldots \forall f_k.g\uparrow$ if $u = f_1 \ldots f_k g$ is a concrete path.

ABoxes *Let $\mathsf{O_a}$ and $\mathsf{O_c}$ be disjoint and countably infinite sets of* abstract *and* concrete objects. *Let C be an $\mathcal{ALC}(\mathcal{D})$ concept, $R \in \mathsf{N_R}$ a role (possibly an abstract feature), g a concrete feature, $a, b \in \mathsf{O_a}$, $x, x_1, \ldots, x_n \in \mathsf{O_c}$ and $P \in \Phi^{\mathcal{D}}$ with arity n. Then*

$$a : C, \quad (a, b) : R, \quad (a, x) : g, \quad \text{and} \quad (x_1, \ldots, x_n) : P$$

*are $\mathcal{ALC}(\mathcal{D})$-*assertions. *An $\mathcal{ALC}(\mathcal{D})$-*ABox *is a finite set $\mathcal{A}$ of $\mathcal{ALC}(\mathcal{D})$-assertions.*

TBoxes *Let C and D be $\mathcal{ALC}(\mathcal{D})$ concepts. Then an expression of the form $C \doteq D$ is called a* concept equation. *A finite set $\mathcal{T}$ of concept equations is called a* general TBox. *An expression of the form $A \doteq C$, where A is a concept name and C is a concept, is called a* concept definition. *For $\mathcal{T}$ a finite set of concept definitions, we say that a concept name A directly uses a concept name B in $\mathcal{T}$ if there is a concept definition $A \doteq C \in \mathcal{T}$ such that B occurs in C. We let the relation "uses" denote the transitive closure of "directly uses." A finite set of concept definitions $\mathcal{T}$ is called an* acyclic TBox *if*

1. *there is no concept name A such that A uses itself, and*

2. *the left-hand sides of all concept definitions in $\mathcal{T}$ are pairwise distinct.*

Definition 2.1.4 ($\mathcal{ALC}(\mathcal{D})$-semantics)**.** *An $\mathcal{ALC}(\mathcal{D})$-interpretation $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set called the* abstract domain *and $\cdot^{\mathcal{I}}$ is an* interpretation function *that maps*

- *every concept name $A \in \mathsf{N_A}$ to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$,*

- *every role name $R \in \mathsf{N_R} \backslash \mathsf{N_{aF}}$ to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$,*

- *every abstract feature $f \in \mathsf{N_{aF}}$ to a partial function $f^{\mathcal{I}} : \Delta^{\mathcal{I}} \rightarrow \Delta^{\mathcal{I}}$, and*

- *every concrete feature $g \in \mathsf{N_{cF}}$ to a partial function $g^{\mathcal{I}} : \Delta^{\mathcal{I}} \rightarrow \Delta^{\mathcal{D}}$.*

If $u = f_1 \ldots f_k g$ is a concrete path, then $u^{\mathcal{I}}$ is defined as the composition of the path's components: $g^{\mathcal{I}}(f_k^{\mathcal{I}}(\cdots(f_1^{\mathcal{I}}(\cdot))))$. The interpretation function $\cdot^{\mathcal{I}}$ is extended to complex concepts as follows:

$$\begin{array}{rcl}
(\neg C)^{\mathcal{I}} & := & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} & := & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} & := & C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\exists R.C)^{\mathcal{I}} & := & \{d \mid \text{there exists } e \in \Delta^{\mathcal{I}} \text{ such that } (d,e) \in R^{\mathcal{I}} \text{ and } e \in C^{\mathcal{I}}\} \\
(\forall R.C)^{\mathcal{I}} & := & \{d \mid \text{for all } e \in \Delta^{\mathcal{I}}, (d,e) \in R^{\mathcal{I}} \text{ implies } e \in C^{\mathcal{I}}\} \\
(\exists u_1, \ldots, u_n.P)^{\mathcal{I}} & := & \{d \in \Delta^{\mathcal{I}} \mid \text{there exists } x_1, \ldots, x_n \in \Delta^{\mathcal{D}} \text{ such that} \\
 & & \;u_i^{\mathcal{I}}(d) = x_i \text{ for } 1 \leq i \leq n \text{ and } (x_1, \ldots, x_n) \in P^{\mathcal{D}}\} \\
(g\uparrow)^{\mathcal{I}} & := & \{d \in \Delta^{\mathcal{I}} \mid g^{\mathcal{I}}(d) \text{ undefined}\}
\end{array}$$

Interpretations $\mathcal{I}$ can be extended to ABoxes and TBoxes as follows. $\mathcal{I}$ maps each $a \in \mathsf{O_a}$ to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and each $x \in \mathsf{O_c}$ to an element $x^{\mathcal{I}} \in \Delta^{\mathcal{D}}$. Then, $\mathcal{I}$ satisfies:

$$\begin{array}{rcl}
a : C & \text{iff} & a^{\mathcal{I}} \in C^{\mathcal{I}} \\
(a,b) : R & \text{iff} & (a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}} \\
(a,x) : g & \text{iff} & g^{\mathcal{I}}(a^{\mathcal{I}}) = x^{\mathcal{I}} \\
(x_1, \ldots, x_n) : P & \text{iff} & P^{\mathcal{D}}(x_1^{\mathcal{I}}, \ldots, x_n^{\mathcal{I}})
\end{array}$$

An interpretation $\mathcal{I}$ satisfies an ABox $\mathcal{A}$ iff it satisfies all assertions $A \in \mathcal{A}$.

An interpretation $\mathcal{I}$ satisfies a concept equation/definition $C \doteq D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$, and it satisfies a TBox $\mathcal{T}$ iff it satisfies all $T \in \mathcal{T}$.

A concept C is satisfiable if there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$. Such an interpretation is called a model *of C. A concept C* subsumes *a concept D (written $C \sqsubseteq D$) iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all interpretations $\mathcal{I}$. Two concepts are* equivalent *(written $C \equiv D$) iff they mutually subsume each other.*

A concept C is satisfiable w.r.t. a TBox *$\mathcal{T}$ iff there exists a model of (both) C and $\mathcal{T}$. A concept C* subsumes *a concept D* w.r.t. a TBox *$\mathcal{T}$ (written $C \sqsubseteq_{\mathcal{T}} D$) iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{T}$. Two concepts C and D are* equivalent w.r.t. a TBox *$\mathcal{T}$ (written $C \equiv_{\mathcal{T}} D$) iff $C \sqsubseteq_{\mathcal{T}} D$ and $D \sqsubseteq_{\mathcal{T}} C$.*

2.1.1 Reasoning Problems

Description logic knowledge bases can be used to express concepts in a particular application domain, the relations between these concepts, and the relations between individual objects and these concepts. Once such knowledge is described, it would be desirable to be able to solve certain reasoning problems on this description. Important description logic reasoning problems include concept satisfiability, concept subsumption and concept equivalence. Depending on what terminological knowledge is present (e.g., no TBox, an acyclic TBox, a general TBox) different variants of these reasoning problems can be distinguished.

For the description logics we consider, these problems can all be reduced to each other (or the corresponding co-problems). In particular, concept subsumption and equivalence can be reduced to concept (un)satisfiability. For two concepts C and D, we have that $C \sqsubseteq D$ holds iff

$C \sqcap \neg D$ is unsatisfiable, and we have that $C \equiv D$ holds iff $(C \sqcap \neg D) \sqcup (\neg C \sqcap D)$ is unsatisfiable. This can be straightforwardly verified using the model-theoretic semantics.

Another important description logic reasoning problem, that is a generalization of concept satisfiability, is ABox satisfiability. Also for this problem, different variants (depending on the type of terminological knowledge present) can be distinguished. To see that ABox satisfiability is more general than concept satisfiability, it suffices to check that a concept C is satisfiable iff the ABox $\{a : C\}$, for some $a \in \mathsf{O_a}$, is satisfiable.

All the reasoning problems mentioned above can thus be reduced to the problem of ABox satisfiability (possibly with respect to terminological knowledge). An algorithm for this latter problem would thus be extremely useful. In Section 2.2, we will see that there are conceptually simple algorithms available for this purpose.

2.1.2 Nominal Concepts

In this thesis, we will also consider description logics with nominal concepts. Nominal concepts are concepts that require an interpretation of size 1. In other words, for every interpretation $\mathcal{I}$ and every nominal concept C_o, we have that $|C_o^{\mathcal{I}}| = 1$. The extension of $\mathcal{ALC}(\mathcal{D})$ with nominal concepts is denoted $\mathcal{ALCO}(\mathcal{D})$.

2.2 Tableau Algorithms for Description Logics

As mentioned in Section 2.1.1, reasoning algorithms that can solve the problem of ABox satisfiability are highly desirable in the field of description logics. We discuss the family of algorithms that is most commonly used for this purpose: tableau algorithms (cf. [2]).

The method of semantic tableaux can also be used for other logics, such as propositional logic, modal logics and first-order logic, and has been introduced by Beth in [4]. Tableau algorithms involve an exhaustive semantic search for a witness of satisfiability of a given logic sentence.

First, we shortly discuss the general working of tableau algorithms in the setting of description logics. Then, we elaborate briefly on the extension of this general tableau algorithm to extensions of the basic description logic $\mathcal{ALC}$. Finally, we summarize the advantages that this type of reasoning algorithm offers, both conceptually and practically.

2.2.1 General Working

The algorithm starts with a tableau (or table) containing a number of description logic (assertional) statements, and tries to find a model witnessing the (simultaneous) satisfiability of these statements. This is done by means of nondeterministic search for such a witness model, guided by the semantics of the statements present in the tableau. For instance, when a statement $a : (C \sqcup D)$ is present in the tableau, either $a : C$ or $a : D$ is also added, since for the first statement to be true in a model, one of the latter two statements must also be true. Another example is that for the statement $a : \exists R.C$ the statements $(a, b) : R$ and $b : C$ are added, for a fresh object name b. In other words, the algorithm tries to make the statements more explicit by adding semantically more explicit statements. This procedure of nondeterministically adding more explicit statements might result in a “clash,” which corresponds to an inconsistency in the constructed candidate witness. For instance, a tableau containing statements $a : C$ and $a : \neg C$ is obviously inconsistent.

The algorithm thus traverses the search tree corresponding to the different possibilities of adding statements to the tableau, and succeeds if there is a branch of the search tree that contains no clash and for which no further statements can be added (i.e. the tableau is as explicit as possible). The soundness of this algorithm results from the fact that such maximally explicit tableaux directly correspond to models of the original statements. The algorithm is complete by virtue of the exhaustiveness of the search algorithm (i.e. all possibilities are eventually tried).

2.2.2 Extensions

The algorithm as described above, is the simplest form of tableau algorithms used for the description logic $\mathcal{ALC}$ without TBoxes. For extensions of this basic reasoning setting, the algorithm needs to be extended as well. Tableau algorithms have been developed for description logics with transitive roles, number restrictions, functional roles, and many more additions to the expressivitiy of the logic (cf. [2]). However, here we restrict ourselves to the extension of tableau algorithms to TBoxes and concrete domains.

When reasoning with respect to terminological axioms, additional statements can be added to the tableaux. For instance, for a statement $a : C$ in the tableau, in the presence of an axiom $C \sqsubseteq D$, the statement $a : D$ needs to be added as well. For acyclic TBoxes, this can be done straightforwardly. General TBoxes, however, can introduce cyclic behavior. An example of such cyclic behavior results from the combination of the statement $a : C$ and the terminological axioms $C \sqsubseteq \exists R.D$ and $D \sqsubseteq \exists R.C$. Blocking conditions need to be introduced to the tableau algorithms to circumvent this behavior. These conditions prevent cyclic behavior, and detect situations in which a model for the original statements can be constructed from the statements in the tableau.

For the extension of description logics to concrete domains, the tableau algorithms also need to be adapted. This adaptation involves modularly using an external reasoning algorithm that decides the satisfiability of finite conjunctions of the concrete domain predicates. This external algorithm is used to detect clashes based on the statements referring to the concrete domain. Such extensions of the tableau algorithm to concrete domains, in the general case, do not work for general TBoxes, however.

2.2.3 Efficient Implementations

Most modern, efficient implementations solving description logic reasoning problems are based on tableau algorithms. As a result of this, much work has been done on optimizing tableau algorithms. For instance, sophisticated data structures have been developed for tableau algorithms, that result in a more efficient execution of the search procedure.

2.2.4 Conceptual Advantages

The tableau algorithms are conceptually easy to understand. Essentially, the only task of these algorithms is to find a model for a number of statements. The search for such a model is directly guided by the model-theoretic semantics of the statements present in the tableau and the terminological axioms present. E.g., there is a direct relation between the tableau rules applicable for a statement and the semantics of this statement. This makes designing such algorithms conceptually very straightforward. As a result, it is conceptually relatively easy to extend the algorithm to more expressive logics, containing more complicated statements. The additional rules and mechanisms needed for such extensions namely are dictated by the semantics of the introduced operators and statements. The design of algorithms for such extensions of the language is thus guided by the semantics.

An illustration of this relation and its guidance in the design of algorithms concerns blocking conditions. The situations in which such blocking conditions are applicable (and required) directly correspond to cases in which finite models of the corresponding statements are acyclic. The witness models that correspond to these tableaux for which blocking conditions apply are exactly such cyclic finite models. In essentially all cases of extensions of the description logics to more expressive variants, the adaptation of the tableau algorithms to these variants are based on the model-theoretic semantics of the introduced language constructs.

In fact, the tableau algorithms that have been developed for expressive description logics are often extremely sophisticated, resulting from the complex model-theoretic semantics of these expressive logics. The development of such sophisticated algorithms is made possible by the straightforward relation between these algorithms and the semantics of the logic.

CHAPTER 3

Programming Languages

The definitions of programming languages – of their syntax and the way programs specify computation – are heterogeneous. Traditionally, programming languages were mainly imperative, i.e. programs directly specified the procedure to be followed in the process of computation. Examples of such traditional imperative languages are FORTRAN and C. Many popular modern programming languages are also based on this imperative paradigm (examples are Java, C++, C#, etc). Besides languages from this imperative paradigm, there are also declarative programming languages, with the main subparadigms of logic programming languages and functional programming languages. Declarative languages specify what is to be computed, without specifying the control flow of how to compute it. Stereotypical examples of functional and logic programming languages, are Haskell and Prolog, respectively.

In this thesis, we consider a number of simple programming languages that are representative for the (sub)paradigms of programming languages mentioned above. In particular, we consider two imperative languages (*While* in Section 3.1.1, and *Goto* in Section 3.1.2), as well as a functional (*FPN* in Section 3.2.1) and a logic programming language (*LPN* in Section 3.2.2). For the sake of simplicity, and in order to be able to straightforwardly define reasoning problems over programs of different programming languages, we restrict all programming languages to the same domain of values (namely numerical values).

The *Goto* language shows parallels to early, low-level imperative programming languages, such as early versions of FORTRAN, while the *While* language is closer to later imperative languages such as C, that uses (while) loops instead of goto statements. The programming language *FPN*, as mentioned above, is a functional language, and shows many parallels to the general-purpose functional programming language Haskell. Similarly, the language *LPN* is a logic programming language, and shows many parallels to the prototypical logic programming language Prolog.

Below, we formally define the syntax and semantics of the programming languages *While*, *Goto*, *FPN* and *LPN*. For the imperative languages *While* and *Goto*, the (operational) semantics of a program is defined as a (partial) mapping from states (which are themselves mappings from variables to numerical values) to states, corresponding to the input-output relation induced by the program. For the declarative languages *FPN* and *LPN*, the semantics of a program is defined as relations over numerical values for the different syntactical elements (function or predicate symbols) of the program.

3.1 Imperative Languages

We consider two examples of imperative programming languages. Before we do so, we introduce some formal machinery that is common to the semantic formalization of both languages.

Given a countably infinite set of variables $\mathcal{X}$, and a finite subset of variables $X \subseteq \mathcal{X}$, we define the set of states over X, denoted with $\mathcal{S}_X$, as the set of total mappings $s : X \rightarrow \mathbb{N}$. A state over a set of variables X is then simply a mapping $s \in \mathcal{S}_X$. Imperative programming languages define programs that semantically are (partial) functions $\mathcal{S}_X \rightarrow \mathcal{S}_X$, over a particular set of variables X.

3.1.1 The Programming Language *While*

3.1.1.1 Syntax

We define the syntax of the simple representative imperative programming language *While* (defined and used for similar purposes in [19, 20]) with the following context-free grammar in Backus-Naur form (we use right-associative bracketing). We let n range over values of $\mathbb{N}$, x over the set of variables $\mathcal{X}$, a over expressions of category **AExp**, b over expressions of category **BExp**, and p over expressions of category **While**.

$$\begin{array}{lcl} a & ::= & n \mid x \mid a + a \mid a \star a \mid a - a \\ b & ::= & \top \mid \bot \mid a = a \mid a \leq a \mid \neg b \mid b \wedge b \\ p & ::= & x := a \mid \mathit{skip} \mid p ; p \mid \textbf{if } b \textbf{ then } p \textbf{ else } p \mid \textbf{while } b \textbf{ do } p \end{array}$$

We consider programs as expressions of category **While**. We denote the set of variables occurring in a program p with $Var(p)$, the set of subterms of p of category **BExp** with $Bool(p)$, and the set of subterms of p of category **AExp** with $Arith(p)$.

Furthermore, we define a notion of 'executive closure' on programs. Intuitively, for a given program p, its closure $cl(p)$ contains all subprograms that could in principle occur in any execution of the program p. For instance, if a (sub)program of the form $((\textbf{while } b \textbf{ do } q); r)$ can occur in an execution, then also **if** b **then** $(q; (\textbf{while } b \textbf{ do } q); r)$ **else** $(skip; r)$. Formally, with $cl(p)$ we denote the smallest set of programs such that:

- $p \in cl(p)$;
- if $(\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2) \in cl(p)$, then $\{p_1, p_2\} \subseteq cl(p)$;
- if $((\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2); q) \in cl(p)$, then $\{(p_1; q), (p_2; q)\} \subseteq cl(p)$;
- if $(\textbf{while } b \textbf{ do } q) \in cl(p)$, then $(\textbf{if } b \textbf{ then } (q; (\textbf{while } b \textbf{ do } q)) \textbf{ else } skip) \in cl(p)$;
- if $((\textbf{while } b \textbf{ do } q); r) \in cl(p)$, then $(\textbf{if } b \textbf{ then } (q; (\textbf{while } b \textbf{ do } q); r) \textbf{ else } (skip; r)) \in cl(p)$; and

- if $(p_1; p_2) \in cl(p)$ and p_1 is of the form $skip$ or of the form $x := e$, then $p_2 \in cl(p)$.

3.1.1.2 Operational Semantics

In order to define the operational semantics of *While* programs, we must firstly introduce the interpretation of Boolean and arithmetical expressions. We define the function $\mathcal{B}^X$ that interprets expressions of category **BExp** as a function from $State_X$ to $\mathbb{B}$. Formally, $\mathcal{B}^X$ takes a state $s \in \mathcal{S}_X$ and an expression of category **BExp** as arguments, and returns a boolean value.

$$\begin{array}{rcll}
\mathcal{B}^X(s, a_1\,\sigma\,a_2) & = & \mathcal{A}^X(s, a_1)\,\sigma\,\mathcal{A}^X(s, a_2) & \text{for } \sigma \in \{=, \leq\} \\
\mathcal{B}^X(s, \neg b) & = & \neg\mathcal{B}^X(s, b) & \\
\mathcal{B}^X(s, b_1 \wedge b_2) & = & \mathcal{B}^X(s, b_1) \wedge \mathcal{B}^X(s, b_2) &
\end{array}$$

We define the function $\mathcal{A}^X$ that interprets expressions of category **AExp** as a function from $State_X$ to $\mathbb{N}$. Formally, $\mathcal{A}^X$ takes a state $s \in \mathcal{S}_X$ and an expression of category **AExp** as arguments, and returns a natural number.

$$\begin{array}{rcll}
\mathcal{A}^X(s, n) & = & n & \text{for } n \in \mathbb{N} \\
\mathcal{A}^X(s, x) & = & state(x) & \text{for } x \in X \\
\mathcal{A}^X(s, a_1\,\rho\,a_2) & = & \mathcal{A}^X(s, a_1)\,\rho\,\mathcal{A}^X(s, a_2) & \text{for } \rho \in \{+, \star\} \\
\mathcal{A}^X(s, a_1 - a_2) & = & \mathcal{A}^X(s, a_1) - \mathcal{A}^X(s, a_2) & \text{if } \mathcal{A}^X(s, a_1) - \mathcal{A}^X(s, a_2) \geq 0 \\
\mathcal{A}^X(s, a_1 - a_2) & = & 0 & \text{if } \mathcal{A}^X(s, a_1) - \mathcal{A}^X(s, a_2) < 0
\end{array}$$

Next, we define the modification of values that states assign to variables. For $s \in State_X$, $x \in X$ and $n \in \mathbb{N}$, we define

$$s[x \mapsto n](y) = \begin{cases} n & \text{if } x = y \\ s(y) & \text{otherwise} \end{cases}$$

For a program p and a set X such that $var(p) \subseteq X \subseteq \mathcal{X}$, we define the operational semantics as follows. We consider the transition system $(\Gamma, T, \Rightarrow)$, where $\Gamma = \{(q, s) \mid q \in cl(p), s \in State_X\} \cup T$, $T = State_X$, and $\Rightarrow \subseteq \Gamma \times \Gamma$.

We define the relation $\Rightarrow$ as the smallest relation such that for each $s \in State_X$, for each $a \in$ **AExp**, and for each $b \in$ **BExp**:

- we have $(skip, s) \Rightarrow s$;
- we have $(x := a, s) \Rightarrow s[x \mapsto \mathcal{A}^X(a, s)]$;
- we have $(p_1; p_2), (p_1', p_2) \in cl(p)$ and $(p_1, s) \Rightarrow (p_1', s')$ imply $(p_1; p_2, s) \Rightarrow (p_1'; p_2, s')$;
- we have $(p_1; p_2) \in cl(p)$ and $(p_1, s) \Rightarrow s'$ imply $(p_1; p_2, s) \Rightarrow (p_2, s')$;
- we have $(\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2, s) \Rightarrow (p_1, s)$, if $\mathcal{P}^X(b, s) = \top$;
- we have $(\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2, s) \Rightarrow (p_2, s)$, if $\mathcal{P}^X(b, s) = \bot$; and
- we have $(\textbf{while}\ b\ \textbf{do}\ p, s) \Rightarrow (\textbf{if}\ b\ \textbf{then}\ (p; \textbf{while}\ b\ \textbf{do}\ p)\ \textbf{else}\ skip, s)$.

Note that $\Rightarrow$ is deterministic, i.e., for any s, t, t', if $s \Rightarrow t$ and $s \Rightarrow t'$, then $t = t'$. We say that p terminates on s with outcome t if $(p, s) \Rightarrow^* t$ for $t \in T$. We say that p does not terminate on s if there is no $t \in T$ such that $(p, s) \Rightarrow^* t$. Note also that if p does not terminate on s, then there is an infinite sequence $(p, s) \Rightarrow (p', s') \Rightarrow \ldots$ starting from (p, s).

3.1.1.3 Normal Form

We define a notion of normal forms for programs. Without loss of generality, we assume any *While* program p is of the form $(p_1; \ldots; p_n; \mathit{skip})$. we consider the following substitutions that are applied on the separate p_i and that preserve the operational semantics of programs. Let x denote a fresh variable, let ρ range over $\{+, \star, -\}$, and π over $\{=, \leq\}$, and let $\varphi[\cdot]$ range over contexts (**if** $\cdot$ **then** p_1 **else** p_2) and (**while** $\cdot$ **do** p).

$$\begin{array}{rclcll}
x := a_1 \,\rho\, a_2 & \rightsquigarrow & x_1 := a_1 & ; & \varphi[x := x_1 \,\rho\, a_2] & \text{if } a_1 \notin \mathcal{X} \\
x := a_1 \,\rho\, a_2 & \rightsquigarrow & x_2 := a_2 & ; & \varphi[x := a_1 \,\rho\, x_2] & \text{if } a_2 \notin \mathcal{X} \\
\varphi[a_1 \,\pi\, a_2] & \rightsquigarrow & x := a_1 & ; & \varphi[x \,\pi\, a_2] & \text{if } a_1 \notin \mathcal{X} \\
\varphi[a_1 \,\pi\, a_2] & \rightsquigarrow & x := a_2 & ; & \varphi[a_1 \,\pi\, x] & \text{if } a_2 \notin \mathcal{X}
\end{array}$$

Using the transformations on programs given by the above substitutions, we can transform any program p to a program p' that is operationally equivalent (when considering only the variables occurring in p) and such that the following holds:

- each subexpression of p' of category **AExp** is either of the form $x \;\rho\; y$, for $x, y \in \mathcal{X}$ and $\rho \in \{+, \star, -\}$, or of the form n for $n \in \mathbb{N}$;
- for each subexpression of p' of category **BExp** of the form $t \;\rho\; s$, for $\rho \in \{=, \leq\}$, holds $t, s \in \mathcal{X}$; and
- either $p' = \mathit{skip}$, or p' is of the form $e; \mathit{skip}$, for some expression e of category **While**.

We will say that programs that satisfy this particular condition are in normal form. In the remainder of this thesis we will often assume that programs are in normal form, which will make it easier to specify the encoding of programs into description logic.

3.1.1.4 Running Example

In order to illustrate the programming language *While*, and concepts and definitions related to *While* programs, we will introduce a running example. We will use this example more often in the remainder of this thesis.

Example 3.1.1. *Consider the following* While *program p. For input values given in variables x and y, it returns the value* $\min\{x, 3\} \cdot y$ *as the value of variable z.*

$$p = (z := 0; w := 0; \overbrace{\textbf{\textit{while}}\ (w < x \wedge w < 3)\ \textbf{\textit{do}}\ (z := z + y; w := w + 1); \mathit{skip}}^{p'})$$

Note that we use p' as an abbreviation for the indicated subprogram of p.

Note that this example program p is not in normal form. For the sake of brevity, and in order to keep the examples uncluttered, we will transform p into normal form only when needed.

An example derivation for p for a state $s \in \mathcal{S}_{\{w,x,y,z\}}$, with $s(w) = 2$, $s(x) = 2$, $s(y) = 5$ and $s(z) = 3$, is given in Figure 3.1. As you can see, in fact, the program returns the value $\min\{s(x), 3\} \cdot s(y) = 10$ as the value of variable z.

3.1.1.5 Stepwise division of derivations

An auxiliary definition that we will use in later chapters of this thesis is the stepwise division of (finite) derivations. With this notion, we divide derivations into partial derivations in such a way that the execution of an assignment (sub)program happens at (each and only at) the end of a partial derivation. The reason for introducing this notion of stepwise division of derivations is purely technical. When encoding derivations into description logic interpretations, in Chapter 4, we will let typical models consist of partial derivations that are distinguished in the stepwise division. This allows us to encode actual state changes (i.e. variable assignments) with transitions, and the conditional behavior of (sub)programs in a derivation with properties of program-state pairs.

In the formal definition of stepwise divisions we distinguish several cases.

Definition 3.1.2 (Stepwise division of finite derivations)**.** *For any finite derivation* $d = (p_1, s_1) \Rightarrow \dots \Rightarrow (p_n, s_n) \Rightarrow s_{n+1}$ *we define the* stepwise division *of this derivation to be the sequence of subderivations* $d_1, \dots, d_m$ *such that:*

- $d = [d_1 \Rightarrow \dots \Rightarrow d_m]$,
- *for the last subderivation* $d_m = [(p'_1, s'_1) \Rightarrow \dots \Rightarrow (p'_r, s'_r) \Rightarrow s']$ *none of the* p'_j *for* $1 \leq j \leq r$ *is of the form* $(x := e; q)$*; and*
- *for all* $1 \leq i < m$ *the subderivation* $d_i = [(p'_1, s'_1) \Rightarrow \dots \Rightarrow (p'_r, s'_r)]$ *satisfies that* p'_r *is of the form* $(x := e; q)$ *and for all* $1 \leq j < r$ *it holds that* p'_j *is not of the form* $(x := e; q)$.

The stepwise division of any finite derivation is uniquely defined.

Definition 3.1.3 (Stepwise division of infinite derivations)**.** *For an infinite derivation* $d = (p_1, s_1) \Rightarrow (p_2, s_2) \Rightarrow \dots$ *where infinitely many* p_i *are of the form* $(x := e; q)$*, we define the* stepwise division *of this derivation to be the sequence of subderivations* $d_1, d_2, \dots$ *such that:*

- $d = [d_1 \Rightarrow d_2 \Rightarrow \dots]$,
- *for all* $i \in \mathbb{N}$ *the subderivation* $d_i = [(p'_1, s'_1) \Rightarrow \dots \Rightarrow (p'_r, s'_r)]$ *satisfies that* p'_r *is of the form* $(x := e; q)$ *and for all* $1 \leq j < r$ *it holds that* p'_j *is not of the form* $(x := e; q)$.

For an infinite derivation $d = (p_1, s_1) \Rightarrow (p_2, s_2) \Rightarrow \dots$ *where only finitely many* p_i *are of the form* $(x := e; q)$*, we define the* stepwise division *of this derivation to be the sequence of subderivations* $d_1, \dots, d_m$ *such that:*

- $d = [d_1 \Rightarrow \dots \Rightarrow d_m]$,

- *for the last subderivation* $d_m = [(p'_1, s'_1) \Rightarrow (p'_2, s'_2) \Rightarrow \dots]$ *none of the* p'_j *is of the form* $(x := e; q)$*; and*
- *for all* $1 \leq i < m$ *the subderivation* $d_i = [(p'_1, s'_1) \Rightarrow \dots \Rightarrow (p'_r, s'_r)]$ *satisfies that* p'_r *is of the form* $(x := e; q)$ *and for all* $1 \leq j < r$ *it holds that* p'_j *is not of the form* $(x := e; q)$*.*

The stepwise division of any infinite derivation is uniquely defined.

It is easy to verify that for any derivation with stepwise division $d_1, \dots, d_n$ we have that for any $d_i = [(p_1, s_1) \Rightarrow \dots \Rightarrow (p_r, s_r)]$ we get by the definition of the operational semantics that $s_1 = \dots = s_r$ (we will say that this state $s_1 = \dots = s_r$ is the state of the subderivation d_i).

For partial derivations $d = [(p_1, s_1) \Rightarrow \dots \Rightarrow (p_m, s_m)]$ occurring in stepwise divisions, we write $(p_i, s_i) \in d$ for all $1 \leq i \leq m$. Similarly for infinite partial derivations. Also, for any successive partial derivations d_i and d_{i+1} occurring in a stepwise division we write $d_i \Rightarrow d_{i+1}$.

To illustrate this notion of stepwise division of derivations, in Figure 3.1 we give an example of a stepwise division of a derivation for our running example.

3.1.2 The Programming Language *Goto*

3.1.2.1 Syntax

We define the syntax of the simple representative imperative programming language *Goto*. We use expressions of categories **AExp** and **BExp** as defined in Section 3.1.1. We let n range over values of $\mathbb{N}$, x over the set of variables $\mathcal{X}$, a over expressions of category **AExp**, b over expressions of category **BExp**, and p over expressions of category **Goto**. We define expressions of category **Goto** with the following context-free grammar in Backus-Naur form:

$$p \quad ::= \quad x := a \mid return \mid \textbf{if } b \textbf{ goto } n \textbf{ else } n$$

For any $l \in \mathbb{N}$, we let $\textbf{Goto}^l$ denote the set of expressions of category **Goto** for which it holds that if they are of the form **if** b **goto** n_1 **else** n_2 then $n_1 \leq l$ and $n_2 \leq l$.

We now define a *Goto* program as a total function $\kappa : \{1, \dots, l\} \rightarrow \textbf{Goto}^l$, for some size $l \in \mathbb{N}$ of κ, such that $\kappa(l) = return$. Here $\{1, \dots, l\}$ constitutes the set of labels of the *Goto* program. For an example of a *Goto* program, see Section 3.1.2.5 below.

We let the set of variables occurring in a program κ of size l, denoted $Var(\kappa)$, be the union of variables occurring in terms $\kappa(i)$, for $1 \leq i \leq l$. We let $Bool(\kappa)$ be the union of subterms $Sub(b)$ of expressions b occurring in terms $\kappa(i) =$ **if** b **goto** n_1 **else** n_2, for $1 \leq i \leq l$.

3.1.2.2 Operational Semantics

We use the definitions of states $\mathcal{S}_X$ and interpretation functions $\mathcal{A}^X$ and $\mathcal{B}^X$ for expressions of category **AExp** and **BExp**, for a subset of variables $X \subseteq \mathcal{X}$, from Section 3.1.1. For a program κ of size l and a set X such that $Var(\kappa) \subseteq X \subseteq \mathcal{X}$, we define the operational semantics as follows. We consider the transition system $(\Gamma, T, \Rightarrow_\kappa)$, where $\Gamma = \{(n, s) \mid 1 \leq n \leq l, s \in \mathcal{S}_X\} \cup T$, $T = \mathcal{S}_X$, and $\Rightarrow_\kappa \subseteq \Gamma \times \Gamma$.

For a program κ of size l, we define the relation $\Rightarrow_\kappa$ as the smallest relation such that for each $s \in \mathcal{S}_X$, for each $a \in$ **AExp**, for each $b \in$ **BExp**, and and for each $1 \leq n \leq l$:

$$
\begin{array}{ll}
d_1 & \left\{\begin{array}{rcl} [2;2;5;3] & \bullet & p \end{array}\right. \\
 & \begin{array}{rcl} & \downarrow & \end{array} \\
d_2 & \left\{\begin{array}{rcl} [2;2;5;0] & \bullet & w := 0; p' \end{array}\right. \\
 & \begin{array}{rcl} & \downarrow & \end{array} \\
d_3 & \left\{\begin{array}{rcl} [0;2;5;0] & \bullet & p' \\ & \downarrow & \\ [0;2;5;0] & \bullet & \textbf{if } (w < x \wedge w < 3) \textbf{ then } (z := z + y; w := w + 1; p') \textbf{ else } \mathit{skip} \\ & \downarrow & \\ [0;2;5;0] & \bullet & z := z + y; w := w + 1; p' \end{array}\right. \\
 & \begin{array}{rcl} & \downarrow & \end{array} \\
d_4 & \left\{\begin{array}{rcl} [0;2;5;5] & \bullet & w := w + 1; p' \end{array}\right. \\
 & \begin{array}{rcl} & \downarrow & \end{array} \\
d_5 & \left\{\begin{array}{rcl} [1;2;5;5] & \bullet & p' \\ & \downarrow & \\ [1;2;5;5] & \bullet & \textbf{if } (w < x \wedge w < 3) \textbf{ then } (z := z + y; w := w + 1; p') \textbf{ else } \mathit{skip} \\ & \downarrow & \\ [1;2;5;5] & \bullet & z := z + y; w := w + 1; p' \end{array}\right. \\
 & \begin{array}{rcl} & \downarrow & \end{array} \\
d_6 & \left\{\begin{array}{rcl} [1;2;5;10] & \bullet & w := w + 1; p' \end{array}\right. \\
 & \begin{array}{rcl} & \downarrow & \end{array} \\
d_7 & \left\{\begin{array}{rcl} [2;2;5;10] & \bullet & p' \\ & \downarrow & \\ [2;2;5;10] & \bullet & \textbf{if } (w < x \wedge w < 3) \textbf{ then } (z := z + y; w := w + 1; p') \textbf{ else } \mathit{skip} \\ & \downarrow & \\ [2;2;5;10] & \bullet & \mathit{skip} \\ & \downarrow & \\ [2;2;5;10] & \bullet & \end{array}\right.
\end{array}
$$

Figure 3.1: Example derivation d for the running example p on state $[2;2;5;3]$, together with its stepwise division $d_1, \ldots, d_7$. States $s \in \mathcal{S}_{\{w,x,y,z\}}$ are denoted $[s(w); s(x); s(y); s(z)]$.

- if $\kappa(n) = return$, we have $(n, s) \Rightarrow_\kappa s$;
- if $\kappa(n) = (x := a)$, we have $(n, s) \Rightarrow_\kappa (n + 1, s[x \mapsto \mathcal{A}^X(a, s)])$;
- if $\kappa(n) =$ (**if** b **goto** n_1 **else** n_2, s) and $\mathcal{P}^X(b, s) = \top$, we have $(n, s) \Rightarrow_\kappa (n_1, s)$; and
- if $\kappa(n) =$ (**if** b **goto** n_1 **else** n_2, s) and $\mathcal{P}^X(b, s) = \bot$, we have $(n, s) \Rightarrow_\kappa (n_2, s)$.

Note that $\Rightarrow_\kappa$ is deterministic. We say that κ from line n terminates on s with outcome t if $(n, s) \Rightarrow_\kappa^* t$ for some $t \in T$. We say that κ does not terminate from line n on s if there is no $t \in T$ such that $(n, s) \Rightarrow_\kappa^* t$. Note also that if κ does not terminate from line n on s, then there is an infinite sequence $(n, s) \Rightarrow_\kappa (n', s') \Rightarrow_\kappa \ldots$ starting from (n, s).

In the remainder of this thesis, we will assume without loss of generality for any *Goto* program κ of size l that for all $1 \leq i < l$ we have $\kappa(l) \neq return$. It is easy to transform any *Goto* program κ into a program for which this holds, by replacing all occurrences of $return$ in lines $1 \leq i < l$ by a statement of the form **if** $\top$ **goto** l **else** l, since by definition we have that $\kappa(l) = return$.

3.1.2.3 Normal Form

Similarly to the case for *While* programs (and for similar purposes), we define the notion of normal form for *Goto* programs. Consider the following transformations, preserving the operational semantics of *Goto* programs, for fresh variables x', $\rho \in \{+, -, \star\}$ and $\pi \in \{=, \leq\}$:

$$\begin{array}{rcl l l}
(x := a_1 \,\rho\, a_2) & \rightsquigarrow & x_1 := a_1 & ; \; x := x_1 \,\rho\, a_2 & \text{if } a_1 \notin \mathcal{X} \\
(x := a_1 \,\rho\, a_2) & \rightsquigarrow & x_2 := a_2 & ; \; x := a_1 \,\rho\, x_2 & \text{if } a_2 \notin \mathcal{X} \\
\textbf{if } \varphi[a_1 \,\pi\, a_2] \textbf{ goto } m_1 \textbf{ else } m_2 & \rightsquigarrow & x_1 := a_1 & ; \; \textbf{if } \varphi[x_1 \,\pi\, a_2] \textbf{ goto } m_1 \textbf{ else } m_2 & \text{if } a_1 \notin \mathcal{X} \\
\textbf{if } \varphi[a_1 \,\pi\, a_2] \textbf{ goto } m_1 \textbf{ else } m_2 & \rightsquigarrow & x_2 := a_2 & ; \; \textbf{if } \varphi[a_1 \,\pi\, x_2] \textbf{ goto } m_1 \textbf{ else } m_2 & \text{if } a_2 \notin \mathcal{X}
\end{array}$$

Such transformations are intuitively interpreted as replacing a line of the form t with lines of the form $t_1, \ldots, t_n$. Formally, let κ be a program of size l, and $t \rightsquigarrow t_1; \ldots; t_n$ (the instantiation of) a transformation rule, such that $\kappa(m) = t$, for some $1 \leq m \leq l$. The application of this transformation to κ at line m is the following program κ' of length $l + n - 1$:

$$\kappa'(i) = \begin{cases} \sigma(\kappa(i)) & \text{if } 1 \leq i < m \\ t_{i-m+1} & \text{if } m \leq i < m + n \\ \sigma(\kappa(i - n + 1)) & \text{if } m + n \leq i \leq l + n - 1 \end{cases}$$

where the substitution σ is given by

$$\sigma(x) = \begin{cases} \textbf{if } b \textbf{ goto } \sigma'(m_1) \textbf{ else } \sigma'(m_2) & \text{if } x = \textbf{if } b \textbf{ goto } m_1 \textbf{ else } m_2 \\ x & \text{otherwise} \end{cases}$$

$$\sigma'(i) = \begin{cases} i & \text{if } i \leq m \\ i + n - 1 & \text{if } i > m \end{cases}$$

Using the above transformations on *Goto* programs, it is easy (yet tedious) to verify that we can transform any program κ to a program κ' that is operationally equivalent (when considering only the variables occurring in κ) and such that the following holds:

- for each $1 \leq n \leq l$, if $\kappa(n)$ is of the form $x := a$, then a is either of the form $x \ \rho \ y$, for $x, y \in \mathcal{X}$ and $\rho \in \{+, \star, -\}$, or of the form n for $n \in \mathbb{N}$;
- for each $1 \leq n \leq l$, if $\kappa(n)$ is of the form **if** b **goto** n_1 **else** n_2, then for each subexpression of b of the form $t \ \rho \ s$, for $\rho \in \{=, \leq\}$, we have $t, s \in \mathcal{X}$.

We will say that *Goto* programs that satisfy this particular condition are in normal form. In the remainder of this thesis we will often assume that programs are in normal form, which will make it easier to specify the encoding of programs into description logic.

3.1.2.4 Manipulating *Goto* Programs

In order to make life easier, we define some notation that allows us to easily extract subprograms of *Goto* programs. Let κ be a *Goto* program of size l, and let $1 \leq n_1 \leq n_2 \leq l$. Assume that for all $n_1 \leq i \leq n_2$ we have that $\kappa(i)$ being of the form **if** b **goto** m_1 **else** m_2 implies that $n_1 \leq m_1 \leq n_2$ and $n_1 \leq m_2 \leq n_2$. Then we define the subprogram $sub(\kappa, n_1, n_2)$ as the *Goto* program κ' of size $n_2 - \delta$, where $\delta = n_1 - 1$ and where for all $1 \leq i \leq n_2 - \delta$:

$$\kappa'(i) = \begin{cases} \textbf{if } b \textbf{ goto } (m_1 - \delta) \textbf{ else } (m_2 - \delta) & \text{if } \kappa(i + \delta) = \textbf{if } b \textbf{ goto } m_1 \textbf{ else } m_2 \\ \kappa(i + \delta) & \text{otherwise} \end{cases}$$

It is easy to see that under the assumptions we made the subprogram $sub(\kappa, n_1, n_2)$ is a well-defined *Goto* program.

3.1.2.5 Running Example

In order to illustrate the programming language *Goto*, and concepts and definitions related to *Goto* programs, we will introduce a running example, similar to the running example for *While*. We will use this example in the remainder of this thesis.

Example 3.1.4. *Consider the following* Goto *program κ. Similarly to the program in Example 3.1.1, for input values given in variables x and y, it returns the value* $\min\{x, 3\} \cdot y$ *as the value of variable z.*

$$\kappa = \begin{array}{rl} 1: & z := 0 \\ 2: & w := 0 \\ 3: & \textbf{\textit{if}}\ (w < x \wedge w < 3)\ \textbf{\textit{goto}}\ 4\ \textbf{\textit{else}}\ 7 \\ 4: & z := z + y \\ 5: & w := w + 1 \\ 6: & \textbf{\textit{if}}\ \top\ \textbf{\textit{goto}}\ 3\ \textbf{\textit{else}}\ 3 \\ 7: & return \end{array}$$

An example derivation for κ for a state $s \in \mathcal{S}_{\{w,x,y,z\}}$, with $s(w) = 2$, $s(x) = 2$, $s(y) = 5$ and $s(z) = 3$, is given in Figure 3.2. As one can see, in fact, the program returns the value $\min\{s(x), 3\} \cdot s(y) = 10$ as the value of variable z.

$$\overbrace{([2,2,5,3],1)}^{d_1} \Rightarrow_\kappa \overbrace{([2,2,5,0],2)}^{d_2} \Rightarrow_\kappa \overbrace{([0,2,5,0],3) \Rightarrow_\kappa ([0,2,5,0],4)}^{d_3} \Rightarrow_\kappa$$
$$\overbrace{([0,2,5,5],5)}^{d_4} \Rightarrow_\kappa \overbrace{([1,2,5,5],6) \Rightarrow_\kappa ([1,2,5,5],3) \Rightarrow_\kappa ([1,2,5,5],4)}^{d_5} \Rightarrow_\kappa$$
$$\overbrace{([1,2,5,10],5)}^{d_6} \Rightarrow_\kappa \overbrace{([2,2,5,10],6) \Rightarrow_\kappa ([2,2,5,10],3) \Rightarrow_\kappa ([2,2,5,10],7) \Rightarrow_\kappa [2,2,5,10]}^{d_7}$$

Figure 3.2: Example derivation d for the running example κ on state $[2,2,5,3]$, together with its stepwise division $d_1, \dots, d_7$. States $s \in \mathcal{S}_{\{w,x,y,z\}}$ are denoted $[s(w), s(x), s(y), s(z)]$.

3.1.2.6 Stepwise division of derivations

Just as we did for *While*, we will define the notion of stepwise division of derivations for the programming language *Goto*. We will use this notion for the same (purely technical) purposes as for the corresponding notion for *While* (see Section 3.1.1.5).

Definition 3.1.5 (Stepwise division of finite derivations). *For any finite derivation $d = (m_1, s_1) \Rightarrow \cdots \Rightarrow (m_n, s_n) \Rightarrow s_{n+1}$, for a given* Goto *program κ, we define the* stepwise division *of this derivation to be the sequence of subderivations $d_1, \dots, d_m$ such that:*

- *$d = [d_1 \Rightarrow \cdots \Rightarrow d_m]$,*
- *for the last subderivation $d_m = [(m'_1, s'_1) \Rightarrow \cdots \Rightarrow (m'_r, s'_r) \Rightarrow s']$ none of the $\kappa(m'_j)$ for $1 \leq j \leq r$ are of the form $x := e$; and*
- *for all $1 \leq i < m$ the subderivation $d_i = [(m'_1, s'_1) \Rightarrow \cdots \Rightarrow (m'_r, s'_r)]$ it holds that $\kappa(m'_r)$ is of the form $x := e$ and for all $1 \leq j < r$ it holds that $\kappa(m'_j)$ is not of the form $x := e$.*

The stepwise division of any finite derivation is uniquely defined.

Definition 3.1.6 (Stepwise division of infinite derivations). *For an infinite derivation $d = (m_1, s_1) \Rightarrow (m_2, s_2) \Rightarrow \dots$, for a fixed* Goto *program κ, where infinitely many $\kappa(m_i)$ are of the form $x := e$, we define the* stepwise division *of this derivation to be the sequence of subderivations $d_1, d_2, \dots$ such that:*

- *$d = [d_1 \Rightarrow d_2 \Rightarrow \dots]$,*
- *for all $i \in \mathbb{N}$ the subderivation $d_i = [(m'_1, s'_1) \Rightarrow \cdots \Rightarrow (m'_r, s'_r)]$ it holds that $\kappa(m'_r)$ is of the form $x := e$ and for all $1 \leq j < r$ it holds that $\kappa(m'_j)$ is not of the form $x := e$.*

For an infinite derivation $d = (m_1, s_1) \Rightarrow (m_2, s_2) \Rightarrow \dots$ where only finitely many $\kappa(m_i)$ are of the form $x := e$, we define the stepwise division *of this derivation to be the sequence of subderivations $d_1, \dots, d_m$ such that:*

- *$d = [d_1 \Rightarrow \cdots \Rightarrow d_m]$,*
- *for the last subderivation $d_m = [(m'_1, s'_1) \Rightarrow (m'_2, s'_2) \Rightarrow \ldots]$ none of the $\kappa(m'_j)$ is of the form $x := e$; and*
- *for all $1 \leq i < m$ the subderivation $d_i = [(m'_1, s'_1) \Rightarrow \cdots \Rightarrow (m'_r, s'_r)]$ it holds that $\kappa(m'_r)$ is of the form $x := e$ and for all $1 \leq j < r$ it holds that $\kappa(m'_j)$ is not of the form $x := e$.*

The stepwise division of any infinite derivation is uniquely defined.

It is easy to verify that for any derivation with stepwise division $d_1, \ldots, d_n$ for a program κ we have that for any $d_i = [(m_1, s_1) \Rightarrow \cdots \Rightarrow (m_r, s_r)]$ we get by the definition of the operational semantics that $s_1 = \cdots = s_r$ (we will say that this state $s_1 = \cdots = s_r$ is the state of the subderivation d_i).

For partial derivations $d = [(m_1, s_1) \Rightarrow \cdots \Rightarrow (m_m, s_l)]$ occurring in stepwise divisions, we write $(m_i, s_i) \in d$ for all $1 \leq i \leq l$. Similarly for infinite partial derivations. Also, for any successive partial derivations d_i and d_{i+1} occurring in a stepwise division we write $d_i \Rightarrow d_{i+1}$.

To illustrate this notion of stepwise division of derivations, in Figure 3.2 we give an example of a stepwise division of a derivation for our running example.

3.2 Declarative Languages

We now turn to the two declarative programming languages that we will use in the remainder of the thesis.

3.2.1 The Functional Programming Language *FPN*

We consider a functional programming language operating on the domain of natural numbers. We call this language *FPN*, which can be considered an acronym of "*F*unctional *P*rogramming with *N*atural numbers".

3.2.1.1 Syntax

We fix a finite number of function symbols $F = \{f_1, \ldots, f_n\}$, each with an arity $ar(f_i) \in \mathbb{N}$. We define a condition of length $k \in \mathbb{N}$ to be a sequence of k values in $\mathbb{N} \cup \bot$. We denote the set of all conditions with $Cond$. We define a consequent of arity $k \in \mathbb{N}$ to be a term over $\mathbb{N}$, over the variables $\{x_1, \ldots, x_k\}$, over the binary operators $+$, $-$ and $\star$, and over the operators f of arity $ar(f)$, for $f \in F$. We denote the set of all consequents with $Cons$.

A program π is a mapping from F to $(Cond \times Cons)^*$, such that the following condition holds:

- for $f \in F$, if $\pi(f) = ((c_1, e_1), \ldots, (c_m, e_m))$, we have that each c_i is a condition of length $ar(f)$, and each e_i is a consequent of arity $ar(f)$.

3.2.1.2 Semantics

We define the semantics of programs by (recursively) defining countably many different relations. For each $f \in F$, and for each $i \in \mathbb{N}$, we let Sem^i_f denote the smallest relation on $\mathbb{N}^{ar(f)} \times \mathbb{N}$ such that:

- a tuple $(\overline{n}, n_{res}) = ((n_1, \ldots, n_{ar(f)}), n_{res}) \in \mathbb{N}^{ar(f)} \times \mathbb{N}$ is in Sem^i_f if there exists a (c_k, e_k) in the sequence $\pi(f) = ((c_1, e_1), \ldots, (c_m, e_m))$ such that c_j does not match $(n_1, \ldots, n_{ar(f)})$ for all $1 \leq j < k$, and $\mathcal{J}^i_{\overline{n}}(e_k) = n_{res}$;
- where a condition $(d_1, \ldots, d_l)$ matches a sequence $(n_1, \ldots, n_l)$ if $d_h \neq \perp$ implies $d_h = n_h$, for all $1 \leq h \leq l$;
- and where $\mathcal{J}^i_{\overline{n}}$ is defined as follows

$$\mathcal{J}^i_{\overline{n}}(x) = \begin{cases} n & \text{if } x = n \in \mathbb{N} \cup \{0\} \\ n_j & \text{if } x = x_j \text{ for } 1 \leq j \leq k \text{ and } \overline{n} = (n_1, \ldots, n_k) \\ \mathcal{J}^i_{\overline{n}}(t_1)\ \rho\ \mathcal{J}^i_{\overline{n}}(t_2) & \text{if } x = t_1\ \rho\ t_2 \text{ for some } \rho \in \{+, \star\} \\ & \text{and } \mathcal{J}^i_{\overline{n}}(t_1) \text{ and } \mathcal{J}^i_{\overline{n}}(t_2) \text{ are defined} \\ \mathcal{J}^i_{\overline{n}}(t_1) \ominus \mathcal{J}^i_{\overline{n}}(t_2) & \text{if } x = t_1 - t_2 \\ & \text{and } \mathcal{J}^i_{\overline{n}}(t_1) \text{ and } \mathcal{J}^i_{\overline{n}}(t_2) \text{ are defined} \\ m & \text{if } x = f(t_1, \ldots, t_l) \text{ for } f \in F \text{, all } \mathcal{J}^i_{\overline{n}}(t_i) \text{ are defined, and} \\ & (\mathcal{J}^i_{\overline{n}}(t_1), \ldots, \mathcal{J}^i_{\overline{n}}(t_l), m) \in Sem^j_f \text{ for some } j < i \\ \text{undefined} & \text{otherwise} \end{cases}$$

- and where $\ominus$ is defined as follows

$$n_1 \ominus n_2 = \begin{cases} n_1 - n_2 & \text{if } n_1 - n_2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

We now let Sem_f denote $\bigcup_{i \in \mathbb{N}} Sem^i_f$. Let $k = ar(f)$. We say that the value of f applied to the values $n_1, \ldots, n_k$ is n, denoted $\pi(f)(n_1, \ldots, n_k) = n$, if $(n_1, \ldots, n_k, n) \in Sem_f$.

3.2.1.3 Examples

Consider the simple program π_1 that computes the ith Fibonacci number for a given i. We let $F = \{fib\}$ and $ar(fib) = 1$. Then π_1 is given by:

$$\pi_1(fib) = [((0), 1), ((1), 1), ((\perp), fib(x_1 - 2) + fib(x_1 - 1)]$$

Consider also the program π_2 that computes the Ackermann-Péter function $A(x_1, x_2)$. We let $F = \{ack\}$ and $ar(ack) = 2$. Then π_2 is given by:

$$\pi_2(ack) = [((0, \perp), x_2 + 1), ((\perp, 0), ack(x_1 - 1, 1), ((\perp, \perp), ack(x_1 - 1, ack(x_1, x_2 - 1)))]$$

3.2.1.4 Boolean conditions

For the sake of simplicity, we don't consider any Boolean conditions on conditions into account in the syntax of *FPN*. The syntax and semantics of *FPN* can be extended straightforwardly to include Boolean conditions on the matching of sequences to conditions. All the results in this thesis can also be worked out for such an extension.

We can however simulate a form of Boolean reasoning by means of the current syntactic expressivity. Instead of multiple cases, distinguished by Boolean guards, we can encode conditionals using the (partial) program π_{cond} with functional symbols dedicated to Boolean reasoning.

$$\begin{aligned}
\pi_{cond}(f_{if}) = &\quad (((0,\bot,\bot),x_2),((1,\bot,\bot),x_3)) \\
\pi_{cond}(f_{and}) = &\quad (((1,1),1),((\bot,\bot),0)) \\
\pi_{cond}(f_{or}) = &\quad (((1,\bot),1),((\bot,1),1),((\bot,\bot),0)) \\
\pi_{cond}(f_{neg}) = &\quad (((1),0),((0),1)) \\
\pi_{cond}(f_{\leq}) = &\quad (((0,\bot),1),((\bot,0),0),((\bot,\bot),f_{\leq}(x_1-1,x_2-1)))
\end{aligned}$$

We illustrate how such Boolean reasoning can be done by using this method to specify the running example we gave for *While* and *Goto* for *FPN* as well.

Example 3.2.1. *The following example program π_3 on functional symbols $F = \{g, g_{mult}, g_{min}, f_{if}, f_{\leq}\}$ can be used to calculate $\min\{x,3\}\cdot y$ by computing $\pi_3(g)(x,y)$. As intermediate value, the program calculates $\pi_3(h)(x) = \min\{x,3\}$.*

$$\begin{aligned}
\pi(g) = &\quad (((\bot,\bot),g_{mult}(x_1,x_2,0))) \\
\pi(g_{mult}) = &\quad (((0,\bot),x_3),((\bot,\bot),g_{mult}(x_1-1,x_2,x_3+x_2))) \\
\pi(g_{min}) = &\quad ((\bot,f_{if}(f_{\leq}(x_1,3),x_1,3))) \\
\pi(f_{if}) = &\quad \pi_{cond}(f_{if}) \\
\pi(f_{\leq}) = &\quad \pi_{cond}(f_{\leq})
\end{aligned}$$

3.2.2 The Logic Programming Language *LPN*

Next, we consider a logic programming language, that defines relations on the natural numbers. We call this language *LPN*, which can be considered an acronym of "*L*ogic *P*rogramming with *N*atural numbers".

One prominent feature of many logic programming languages is the automatic search mechanism involving free variables. As we will see below, however, the *LPN* language does not allow free variables in the way that logic programming languages as Prolog do. Therefore, we note that the *LPN* language does not offer the automatic search mechanisms that languages like Prolog do.

3.2.2.1 Syntax

Let $R = \{r_1,\ldots,r_n\}$ be a finite set of predicate symbols with each an arity $ar(r_i) \in \mathbb{N}$, and a countably infinite set of variables $\mathcal{X}$. We define rules as statements of the form:

$$r(x_1,\ldots,x_{ar(r)}) \leftarrow r_1(t_1^1,\ldots,t_{ar(r_1)}^1),\ldots,r_k(t_1^k,\ldots,t_{ar(r_k)}^k),c_1,\ldots,c_m$$

where $r \in R$ and each $r_j \in R$, each $x_i \in \mathcal{X}$, each t^j_l is a term over $\mathbb{N}$, the variables x_i occurring in the left hand side, and the binary operators $+$, $-$ and $\star$, and where each c_i is an expression $t \; \rho \; t'$ for some terms t, t' as defined above, and some $\rho \in \{=, \neq, <, \leq\}$. We call r the head relation symbol of the rule.

We now define a program p as a finite set of rules, for a fixed set of predicate symbols R.

3.2.2.2 Semantics

We define the semantics of programs by (recursively) defining countably many different relations. We fix an input function $\iota : R \to 2^{\mathbb{N}^*}$ such that for each $r \in R$ we have $\iota(r) \subseteq \mathbb{N}^{ar(r)}$ and $\iota(r)$ is finite. Now, for each $r \in R$, and for each $i \in \mathbb{N}$, we let Sem^i_r denote the smallest relation on $\mathbb{N}^{ar(r)}$ such that:

- if $(n_1, \ldots, n_k) \in \iota(r)$, then $(n_1, \ldots, n_k) \in Sem^i_r$;
- for any rule $r(x_1, \ldots, x_l) \leftarrow r_1(\bar{t}^1), \ldots, r_k(\bar{t}^k), c_1, \ldots, c_m$ in the program p, if there exists a mapping σ from the variables $x_1, \ldots, x_l$ to $\mathbb{N}$ such that for all $1 \leq j \leq k$ we have $\sigma(\bar{t}^j) \in Sem^{i'}_{r_j}$ for some $i' < i$, and such that all inequalities $\mathcal{J}_\sigma(t) \; \rho \; \mathcal{J}_\sigma(t')$ hold for $c_z = t \, \rho \, t'$ for all $1 \leq z \leq m$, then $(\sigma(x_1), \ldots, \sigma(x_l)) \in Sem^i_r$;
- where $\mathcal{J}_\sigma$ is defined as follows:

$$\mathcal{J}_\sigma(t) = \begin{cases} n & \text{if } t = n \in \mathbb{N} \\ \sigma(x) & \text{if } t = x \in \mathcal{X} \\ \mathcal{J}_\sigma(t_1) + \mathcal{J}_\sigma(t_2) & \text{if } t = t_1 + t_2 \\ \mathcal{J}_\sigma(t_1) \star \mathcal{J}_\sigma(t_2) & \text{if } t = t_1 \star t_2 \\ \mathcal{J}_\sigma(t_1) \ominus \mathcal{J}_\sigma(t_2) & \text{if } t = t_1 - t_2 \end{cases}$$

- and where $\ominus$ is defined as follows

$$n_1 \ominus n_2 = \begin{cases} n_1 - n_2 & \text{if } n_1 - n_2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

We now define the semantics of an $r \in R$ given the program p and the input ι as the relation $Sem_r = \bigcup_{i \in \mathbb{N}} Sem^i_r$.

In the semantic definition of *LPN* programs we refer to the additional input function ι as well as the syntax of the programs. This input function ι is basically syntactic sugar. For any $r \in R$ and any $(n_1, \ldots, n_k) \in \iota(r)$ we could alternatively add the rule $r(x_1, \ldots, x_k) \leftarrow x_1 = n_1, \ldots, x_k = n_k$ to *LPN* programs, and then we could get rid of ι (or rather, set $\iota(r) = \emptyset$ for all $r \in R$). However, since including the function ι in the definition of *LPN* has a positive effect on the readability of programs, and since it hardly alters the encoding of *LPN* programs into $\mathcal{ALC}(\mathcal{D})$ later in this thesis (specifically, see Axioms (4.51) and (4.51) in Section 4.6), we chose to include the use of ι in the definition of *LPN*. Whenever $\iota(r) = \emptyset$ for all $r \in R$, we omit mentioning ι altogether.

3.2.2.3 Examples

Consider the following example program p_1 for $R = \{even, odd\}$ and $ar(even) = ar(odd) = 1$, whose semantics correspond to the even and odd numbers, respectively.

$$\iota_1(even) = \{0\}$$

$$\begin{array}{lll} p_1 = \{ & even(x_1) & \leftarrow even(x_1 - 2), \\ & odd(x_1) & \leftarrow even(x_1 - 1) \quad \} \end{array}$$

We can express the function computed by the running example of the other programming languages (which computes the value of $\min\{x, 3\} \cdot y$ for input variables x and y) also as an *LPN* program.

Example 3.2.2. *Consider the following* LPN *program p_2 on relational symbols $R = \{answer, min, multiply, multiply'\}$. The value of* $\min\{x, 3\} \cdot y$ *can be computed using this program by checking for what z it holds that $(x, y, z) \in Sem_{answer}$.*

$$\begin{array}{lll} p_2 = \{ & answer(x, y, z) & \leftarrow min(x, 3, x'), multiply(x', y, z) \\ & multiply(x_1, x_2, x_3) & \leftarrow multiply'(x_1, x_2, 0, x_3) \\ & multiply'(x_1, x_2, x_3, x_4) & \leftarrow multiply'(x_1 - 1, x_2, x_3 + x_2, x_4), x_1 > 0 \\ & multiply'(x_1, x_2, x_3, x_4) & \leftarrow x_1 = 0, x_3 = x_4 \\ & min(x_1, x_2, x_3) & \leftarrow x_1 = x_3, x_1 \leq x_2 \\ & min(x_1, x_2, x_3) & \leftarrow x_2 = x_3, x_1 > x_2 \quad \} \end{array}$$

3.3 Computational Power

The programming languages defined above most likely seem fairly primitive and simple to the reader, and anyone familiar with programming languages has most probably seen languages similar to, yet much more complicated than the languages above before. However, these languages have as much computational power as any computational model we know of. Formally, this can be formulated as the statement that any Turing-computable function (i.e. any function that can be computed by a Turing machine [22]) can be computed by a program of the programming languages above. Systems that have this property are called Turing-complete.

The Turing-completeness for the imperative programming languages already follows from the Böhm-Jacopini-Theorem [5], which states that sequential execution, conditional selection and conditional iteration of subprograms are enough to compute all computable functions.

Nevertheless, we will briefly sketch how to show the Turing-completeness of the programming languages we defined. A relatively simple way to show this is to show that any μ-recursive function can be encoded by programs of the programming languages. It is well-known that μ-recursive functions and Turing-computable functions coincide.

Definition 3.3.1 (μ-recursive functions)**.** *The set of μ-recursive functions is the smallest class of functions that includes the basic functions:*

1. ***the constant function:***
   for each $k, n \in \mathbb{N}$, the function $f(x_1, \dots, x_k) = n$;

2. ***the successor function:***
   the function $f(x) = x + 1$*;*

3. ***the projection function:***
   for each $i, k \in \mathbb{N}$ *such that* $1 \leq i \leq k$*, the function* $f(x_1, \dots, x_k) = x_i$*;*

and that is closed under the operations:

4. ***the composition operator*** $\circ$***:***
   given an m*-ary function* $h(x_1, \dots, x_m)$ *and* k*-ary functions* $g_1(x_1, \dots, x_k), \dots,$ $g_m(x_1, \dots, x_k)$*, we define* $h \circ (g_1, \dots, g_m) = h(g_1(x_1, \dots, x_k), \dots, g_m(x_1, \dots, x_k))$*;*

5. ***the primitive recursion operator*** ρ***:***
   given the k*-ary function* $g(x_1, \dots, x_k)$ *and the* $(k+2)$*-ary function* $h(y, z, x_1, \dots, x_k)$ *we define* $\rho(g, h) = f(y, x_1, \dots, x_k)$ *where* $f(0, x_1, \dots, x_k) = g(x_1, \dots, x_k)$ *and* $f(y + 1, x_1, \dots, x_k) = h(y, f(y, x_1, \dots, x_k), x_1, \dots, x_k)$*;*

6. ***the minimisation operator*** μ***:***
   given the $(k+1)$*-ary function* $f(y, x_1, \dots, x_k)$ *we define* $\mu(f)(x_1, \dots, x_k) = z$ *iff* $\exists y_0, \dots, y_z$ *such that* $y_i = f(i, x_1, \dots, x_k)$ *for* $0 \leq i \leq z$*,* $y_i > 0$ *for* $0 \leq i < z$ *and* $y_z = 0$*.*

Proposition 3.3.2. *Every* μ*-recursive function can be expressed by a* While *program.*

Proof (sketch). For each function $f(x_1, \dots, x_k)$ we can (recursively) define a *While* program that uses variables $x_1, \dots, x_k$ as input variables and that returns the value of $f(x_1, \dots, x_k)$ in a variable x_{out}. The basic functions (1)-(3) can very straightforwardly be expressed by *While* programs consisting of only assignment expressions (e.g. $x_{out} := x_i$ for the projection function $f(x_1, \dots, x_k) = x_i$).

For the operations (4)-(6), we can straightforwardly define composition functions on *While* programs that result in suitable *While* programs expressing the result of these operations. For the composition operator, we take the programs for the separate functions $h, g_1, \dots, g_m$. By variable renaming and concatenation in a suitable fashion we can straightforwardly obtain the required *While* program computing $h \circ (g_1, \dots, g_m)$.

For the primitive recursion operator ρ, we take the programs p and q expressing the function g and h, respectively, where the output variable x_{out} is renamed to x_{res} (in both cases) and the input variable y is renamed to x_y (in the case of q). The function $\rho(g, h)$ can then be expressed by the program:

```
$x_c := 0;$
while $(x_c \leq y)$ do
  $(($if $x_c = 0$ then $p$
  else $(z := x_{res}; x_y := x_c - 1; q));$
  $x_c := x_c + 1);$
$x_{out} := x_{res}$
```

For the minimisation operator μ, we take the program p for the function f, where the output variable x_{out} is renamed to z. The function $\mu(f)(x_1, \dots, x_m)$ can then be expressed by the program:

$$\begin{array}{l} y := 0; z := 1; \\ (\textbf{while } z > 0 \textbf{ do } p; y := y + 1); \\ x_{out} := y - 1 \end{array}$$

This recursive argument shows that *While* programs can express any μ-recursive function. □

Similar results can straightforwardly be shown for the languages *Goto* An alternative method to show Turing-completeness for the programming language *Goto* is to show that for each *While* program there exists an equivalent *Goto* program. In fact, we show this in Chapter 6, with Definition 6.4.9 and Theorem 6.4.10.

For the declarative languages *FPN* and *LPN*, Turing-completeness can be shown quite straightforwardly as well. Showing this claim in full detail is beyond the scope of this thesis. We will merely make plausible that this can be done, by suggesting how to express conditional execution and conditional iteration of (sub)programs. For *FPN*, we can express conditional iteration by (partial) programs π containing definitions of the following form:

$$\pi(f_{while}) = (((1, \bot), f_{while}(f_1(x_2), f_2(x_2))), ((0, \bot), x_2))$$

Intuitively, this definition of f_{while} can be understood as $f_{while}(1, x)$ being the conditional execution of the function f_2 to a variable x until the value of $f_1(x)$ becomes 0. Together with definitions similar to those of the Boolean operators from Section 3.2.1.4, this makes it possible to express any Turing-computable function using *FPN*. By a similar argument, the Turing-completeness of *LPN* can be shown.

3.4 Reasoning Problems over Programs

For the programming languages discussed above, we defined the semantics of executing programs. For the imperative languages, this concerns the operational semantics. For the declarative languages, this concerns the (relational) declarative semantics. In essence, this definition of semantics only concerns the input-output relation induced by programs. For imperative programs, the operational semantics maps every input state to an output state. For declarative programs, the declarative semantics maps every input query to an output value.

However, as mentioned in Chapter 1, we are interested in additional semantic properties of programs. Many such additional semantic properties could be defined. For the sake of clarity and concision, we will restrict ourselves to two basic kinds of semantic properties: termination of (imperative) programs and equivalence of programs (both for imperative and declarative programs). Below, we will formally define these semantic properties for programs of the different programming languages. Using the approach that we develop in Chapter 4, we are able to express many more semantic properties. We will elaborate on such additional semantic properties a bit more below (as well as in Chapters 4 and 5), but in the remainder of the thesis we restrict the main discussion and results to the semantic properties of termination and equivalence.

3.4.1 (Uniform) Termination

The semantic property of uniform termination for programs (or simply termination, for short) concerns the question whether every execution of a program, for each possible input state, terminates. Formally, for *While* programs, we say that a program p on variables $X \subseteq \mathcal{X}$ is terminating if for every input state $s \in \mathcal{S}_X$, the derivation starting at (p, s) terminates after finitely many steps, i.e. $(p, s) \Rightarrow^* s'$ for some $s' \in \mathcal{S}_X$. Similarly, for *Goto* programs, we say that a program κ on variables $X \subseteq \mathcal{X}$ is terminating if for every input state $s \in \mathcal{S}_X$, the κ derivation starting at $(1, s)$ terminates after finitely many steps, i.e. $(1, s) \Rightarrow^*_\kappa s'$ for some $s' \in \mathcal{S}_X$.

Termination is an important semantic property for programs to have. For uniformly terminating programs, you can count on the program not running infinitely, which is extremely important in many practical settings. The importance of this termination property is illustrated by the fact that a similar property has been essential in the abstract study of computation (i.e. the uniform halting property of Turing machines), and a similar property is one of the most studied properties for computational mechanisms such as term rewriting systems.

In principle, we could also define termination for the declarative languages *FPN* and *LPN*. However, the notion of termination presupposes a fixed notion of operational execution of programs. Since we did not commit ourselves at all to any operational semantics of the declarative programming languages, we will not go further into the topic of termination for programs of the languages *FPN* and *LPN* in this thesis.

3.4.1.1 Termination on Individual Input States

Besides the uniform termination property of programs mentioned above, one can also distinguish the termination property of programs restricted to particular input states, i.e., the question whether a *While* or *Goto* program with variables $X \subseteq \mathcal{X}$ terminates on a given input state $s \in \mathcal{S}_X$. This property, though theoretically not less interesting, is not as important from a practical point of view, since it does not directly imply program reliability that is similar to the form of reliability that uniform termination guarantees.

Clearly, checking whether a program terminates on a given input state can be reduced to the uniform termination property. We briefly sketch the reduction for *While*. The case of *Goto* is completely similar. Take a *While* program p over variables $\{x_1, \dots, x_k\} = X \subseteq \mathcal{X}$, and take an input state $s \in \mathcal{S}_X$. We have that p terminates on s if and only if the program $(x_1 := s(x_1); \dots; x_k := s(x_k); p)$ is uniformly terminating. Furthermore, in the general case, the problem whether a given program terminates on a given input state is undecidable (cf. Theorem 6.3.21), just like the problem whether a given program is uniformly terminating.

In the remainder of the thesis, we will focus on the uniform termination property of *While* and *Goto* programs. We will refer to uniform termination simply as termination of programs.

3.4.2 Equivalence

Another semantic property of programs (imperative and declarative) that is very relevant for practical purposes, is the property of equivalence of two programs. Intuitively, two programs that are equivalent give exactly the same result for any input state or query, and could thus be used to perform the same computation. This notion of equivalence is essential, for instance, for the (definition of) optimization of programs. We formally define equivalence of programs for the different programming languages we consider.

Two (terminating) *While* programs p_1 and p_2 on variables $X \subseteq \mathcal{X}$ are equivalent if and only if for each input state $s \in \mathcal{S}_X$ and each output state $s' \in \mathcal{S}_X$, we have that $(p_1, s) \Rightarrow^* s'$ iff $(p_2, s) \Rightarrow^* s'$. Similarly, two *Goto* programs κ_1 and κ_2 on variables $X \subseteq \mathcal{X}$ are equivalent if and only if for each input state $s \in \mathcal{S}_X$ and each output state $s' \in \mathcal{S}_X$, we have that $(1, s) \Rightarrow^*_{\kappa_1} s'$ iff $(1, s) \Rightarrow^*_{\kappa_2} s'$. Note that whenever the one program uses variables $X_1 \subseteq \mathcal{X}$ and the other uses variables $X_2 \subseteq \mathcal{X}$, and $X_1 \neq X_2$, we can simply let $X = X_1 \cup X_2$.

Analogously, since *While* and *Goto* programs have the same operational setting (they both define possibly partial mappings $\mathcal{S}_X \to \mathcal{S}_X$) we can define equivalence of a *While* program and a *Goto* program as follows. A *While* program p and a *Goto* program κ on variables $X \subseteq \mathcal{X}$ are equivalent if and only if for each input state $s \in \mathcal{S}_X$ and each output state $s' \in \mathcal{S}_X$, we have $(p, s) \Rightarrow^* s'$ iff $(1, s) \Rightarrow^*_{\kappa} s'$.

In the remainder of the thesis, for the sake of keeping things as little complicated as possible, we will usually only consider equivalence of terminating programs. The definition of equivalence of (imperative) programs above works also in the case of programs that are not necessarily terminating, however.

For *FPN* programs, we can define the notion of equivalence of equivalence on the basis of the declarative (relational) semantics. Let π_1 and π_2 be two *FPN* programs, and assume without loss of generality that they use the same set F of functional symbols. We denote the semantic relation induced by π_1 with Sem^{π_1} and the semantic relation induced by π_2 with Sem^{π_2}. We say that π_1 and π_2 are equivalent on a symbol $f \in F$ of arity k if and only if for each $(n_1, \dots, n_k, n) \in \mathbb{N}^{k+1}$ we have that $(n_1, \dots, n_k, n) \in Sem^{\pi_1}_f$ iff $(n_1, \dots, n_k, n) \in Sem^{\pi_2}_f$.

Similarly, for *LPN* programs, we define equivalence on the basis of the declarative (relational) semantics. Let p_1 and p_2 be two *LPN* programs, and assume without loss of generality that they use the same set R of relational symbols. For the sake of simplicity, we fix an input function ι such that $\iota(r) = \emptyset$ for each $r \in R$. We denote the semantic relation induced by p_1 with Sem^{p_1}, and the semantic relation induced by p_2 with Sem^{p_2}. We say that p_1 and p_2 are equivalent on a symbol $r \in R$ of arity k if and only if for each $(n_1, \dots, n_k) \in \mathbb{N}^k$ we have that $(n_1, \dots, n_k) \in Sem^{p_1}_r$ iff $(n_1, \dots, n_k) \in Sem^{p_2}_r$.

3.4.3 Additional semantic properties

Besides the notions of termination and equivalence of programs defined above, there are many more interesting semantic properties of programs that we could consider. We will briefly mention a few of them here, but we will not further investigate these additional properties in much detail in this thesis.

First of all, we could consider equivalence of imperative and declarative programs. We illustrate this by defining (one possible notion of) equivalence of a *While* program and a *FPN* program. Let p be a *While* program using variables $X \subseteq \mathcal{X}$ and let π be a *FPN* program using functional symbols F. We say that p is equivalent for inputs $\{x_1, \dots, x_k\} = X' \subseteq X$ and output $x \in X$ to π for $f \in F$ if and only if (i) $ar(f) = k$, and (ii) for each $(n_1, \dots, n_k, n) \in \mathbb{N}^{k+1}$ we have that $(n_1, \dots, n_k, n) \in Sem^{\pi}_f$ holds if and only if it is the case that $s(x_1) = n_1, \dots, s(x_k) =$

n_k and $(p, s) \Rightarrow^* s'$ implies $s'(x) = n$ for all $s, s' \in \mathcal{S}_X$. We will consider this property a bit more in Section 4.7. Similar equivalence properties can be defined straightforwardly for the different combinations of imperative and declarative programming languages.

Another class of semantic properties that we could define for programs are conditional variants on termination or equivalence properties. Think for instance of the relaxation of the uniform termination property from termination on all possible input states to termination on a restricted subset of all possible input states. Such conditional semantic properties might be practically relevant if it is known that the input states satisfy a certain condition, for example.

Furthermore, we could also define notions like inverse relations of programs. We say a *While* program p is the inverse of another *While* program p', both containing variables $X \subseteq \mathcal{X}$, iff for each $s \in \mathcal{S}_X$ we have that $(p, s) \Rightarrow^* s'$ and $(p', s') \Rightarrow^* s$, for some $s' \in \mathcal{S}_X$. In fact, the semantic properties that could be defined for programs and that might be relevant to consider are numerous.

CHAPTER 4

Encoding into Description Logic

4.1 General Approach

As explained before, in Chapter 1, we will show how to encode arbitrary programs from the different programming languages into $\mathcal{ALC}(\mathcal{D})$ TBoxes, in such a way that the semantics of the program corresponds to the model theoretic semantics of the TBox resulting from the encoding. In order to do so, we will formally specify these encodings for each of the programming languages introduced in Chapter 3. Along with the specification of these encodings, for each programming language, we formally prove the correspondence between the semantics of programs (as defined in Chapter 3) and the model theoretic semantics of the encoding. Guiding both the encoding and the semantic correspondence proofs are intuitions of how to model the behavior of programs of the different programming languages in the (model-theoretic) semantics of the description logic $\mathcal{ALC}(\mathcal{D})$.

4.2 Modelling the Behavior of Programming Languages

Before we give the formal specifications of the encodings together with the semantic correspondence proofs, we explain these underlying intuitions for each of the different programming (sub)paradigms.

4.2.1 Imperative Programming Languages

In order to describe the intuition behind modelling programs of the imperative programming languages in the semantics of the description logic $\mathcal{ALC}(\mathcal{D})$, we identify the core elements of the semantics of the imperative languages. Essential in the definition of the operational semantics of imperative programs are states (i.e. assignments of values to variables), (sub)programs of the programming language, and derivations involving states and programs. We represent these elements, and the relations between them, in the model-theoretic semantics as follows.

We let objects in our domain represent occurrences of states in derivations of the operational semantics of the imperative language. Note that identical states can occur multiple times (in a derivation of the operational semantics), and therefore we allow multiple occurrences of one particular state. For each state, the assignment of values to (the fixed set of) variables is represented by concrete roles, one for each variable. We represent the notion of successor state in the derivations in the operational semantics of the imperative language as an abstract role in the description logic, which is interpreted as a binary relation on the occurrences of states.

We represent programs defined by the imperative programming language with concepts, which are interpreted as sets of occurrences of states. These concepts allow us to encode restrictions on the possible derivations in the operational semantics, i.e. restrictions on the chains of the binary relation that is the interpretation of the successor state role. Such restrictions, that ultimately ensure the correspondence between the operational semantics of the imperative language and the model theoretic interpretation of the encoding, are encoded using terminological (TBox) axioms. Furthermore, in order to enable us to talk about terminating derivations and resulting states of derivations, we introduce a unique concept referring to terminated programs.

4.2.2 Functional Programming Languages

Similarly to the case of the imperative programming languages, in order to describe the intuition behind modelling programs of the functional programming language *FPN* into the semantics of the description logic $\mathcal{ALC}(\mathcal{D})$, we identify the core elements in the semantics of the this functional programming language. The semantics of this language consists of k-ary relations (one for each function symbol) over numerical values, where intuitively, for each instance, the program determines the last value based on the first $k-1$ values and on other instances of the relations defined in the semantics of the program.

Instances of these semantic relations are represented by objects in the abstract domain of the $\mathcal{ALC}(\mathcal{D})$ model. The values of these instances are encoded by concrete roles, relating the objects corresponding to instances to (numerical) values in the concrete domain. The function symbols present in the program are represented by concepts, that are interpreted as sets of instances occurring in the semantic relation corresponding to the function symbols.

The dependencies between the values of the instances of the semantic relations defined by the program are then encoded using terminological (TBox) axioms. Whenever the values of an instance depend on other instances, objects corresponding to these other instances are selected by using abstract roles.

4.2.3 Logic Programming Languages

Again, similarly to the case of the other programming languages, in order to describe the intuition behind modelling programs of the logic programming language *LPN* into the semantics of the description logic $\mathcal{ALC}(\mathcal{D})$, we identify the core elements in the semantics of the logic programming language. The semantics of this language consists of relations (one for each predicate symbol) over numerical values, where each instance must have a justification based on instances in the input function and on other instances of the relations defined in the semantics of the program. The exact nature of this justification depends on the program.

Similarly to the case for the functional programming language, instances of the semantic relations are represented by objects in the abstract domain of the first-order model. The values of these instances are encoded by concrete roles, relating the objects corresponding to instances to (numerical) values in the concrete domain. The predicate symbols present in the program are represented by concepts, that are interpreted as sets of instances occurring in the semantic relation corresponding to the predicate symbols.

The justification constraints on the instances of the semantic relations defined by the program are then encoded using terminological (TBox) axioms. Whenever this justification depends on other instances, objects corresponding to these other instances are selected by using abstract roles.

4.3 Encoding *While* Programs in $\mathcal{ALC}(\mathcal{D})$

We now show how to model the behavior of programs of the language *While* using description logic. The concrete values in the programming language correspond to concrete values in the description logic. Remember, states will be represented by objects, and programs are represented by concepts. We represent the successor state relation, induced by the execution of programs on states, by a (functional) role $\mathsf{nextState}$.

In particular, for a given program p with $Var(p) = X = \{x_1, \dots, x_n\}$, we denote states $s \in \mathcal{S}_X$ with objects that are related to numerical values with concrete features $\mathsf{valueOf}_{x_i}$ for each $1 \leq i \leq n$.

4.3.1 Constructing a TBox

Take an arbitrary *While* program p, i.e., an expression of category **While**. W.l.o.g., we assume p is in normal form. We define an $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^p$ as follows. We use concept names C_q for each $q \in cl(p)$, and concept names D_b for each $b \in Bool(p)$.

For each variable $x \in Var(p)$, we create a concrete feature $\mathsf{valueOf}_x$, and we require for each $q \in cl(p)$:

$$C_q \sqsubseteq \neg \mathsf{valueOf}_x \uparrow \tag{4.1}$$

We let $\mathsf{nextState}$ be an abstract feature and for C_{skip} we require:

$$C_{skip} \sqsubseteq \neg \exists \mathsf{nextState}.\top \tag{4.2}$$

For each $b \in Bool(p)$, we require the following, where x_1, x_2 range over X, and b_1, b_2 range over $Bool(p)$:

$$D_{x_1 = x_2} \equiv \exists (\mathsf{valueOf}_{x_1})(\mathsf{valueOf}_{x_2}).= \tag{4.3}$$

$$D_{x_1 \leq x_2} \equiv \exists (\mathsf{valueOf}_{x_1})(\mathsf{valueOf}_{x_2}).\leq \tag{4.4}$$

$$D_{\neg b_1} \equiv \neg D_{b_1} \tag{4.5}$$

$$D_{b_1 \wedge b_2} \equiv D_{b_1} \sqcap D_{b_2} \tag{4.6}$$

Furthermore, we let $D_\top$ denote $\top$ and $D_\bot$ denote $\bot$. Then, for each $q \in cl(p)$ of the form $p_1; p_2$ we require the following for C_q, where p_1, p_2, q_1, q_2 range over $cl(p)$, x, y_1, y_2 range over X, a

ranges over $Arith(p)$,

$$C_{(x:=a);p_2} \sqsubseteq \exists \mathsf{nextState}.C_{p_2} \tag{4.7}$$

$$C_{skip;p_2} \sqsubseteq C_{p_2} \tag{4.8}$$

$$C_{(x:=n);p_2} \sqsubseteq \exists(\mathsf{nextState}\ \mathsf{valueOf}_x).{=_n} \tag{4.9}$$

$$C_{(x:=y);p_2} \sqsubseteq \exists(\mathsf{nextState}\ \mathsf{valueOf}_x)(\mathsf{valueOf}_y).{=} \tag{4.10}$$

$$C_{(x:=y_1+y_2);p_2} \sqsubseteq \exists(\mathsf{nextState}\ \mathsf{valueOf}_x)(\mathsf{valueOf}_{y_1})(\mathsf{valueOf}_{y_2}).{+} \tag{4.11}$$

$$\begin{aligned}C_{(x:=y_1-y_2);p_2} \sqsubseteq\ & (\neg\exists(\mathsf{valueOf}_{y_2})(\mathsf{valueOf}_{y_1}).{\leq} \sqcup \\ & \exists(\mathsf{valueOf}_{y_1})(\mathsf{nextState}\ \mathsf{valueOf}_x)(\mathsf{valueOf}_{y_2}).{+}) \sqcap \\ & (\neg\exists(\mathsf{valueOf}_{y_2})(\mathsf{valueOf}_{y_1}).{>} \sqcup \\ & \exists(\mathsf{nextState}\ \mathsf{valueOf}_x).{=0})\end{aligned} \tag{4.12}$$

$$C_{(x:=a);p_2} \sqsubseteq \exists(\mathsf{valueOf}_y)(\mathsf{nextState}\ \mathsf{valueOf}_y).{=} \quad \text{for } y \neq x \tag{4.13}$$

$$C_{(\textbf{if } b \textbf{ then } q_1 \textbf{ else } q_2);p_2} \sqsubseteq (\neg D_b \sqcup C_{q_1;p_2}) \sqcap (D_b \sqcup C_{q_2;p_2}) \tag{4.14}$$

$$C_{(\textbf{while } b \textbf{ do } q);p_2} \sqsubseteq (\neg D_b \sqcup C_{q;(\textbf{while } b \textbf{ do } q);p_2}) \sqcap (D_b \sqcup C_{p_2}) \tag{4.15}$$

Notice that, in general, the TBox $\mathcal{T}^p$ is not acyclic, since Axiom (4.15) can induce a cycle in combination with Axioms (4.7), (4.8) and (4.14).

Intuitively, these axioms serve the following purpose. Axioms (4.1) and (4.2) ensure some basic properties of the modelling of derivations in the model. Axioms (4.3)-(4.6) capture the behavior of Boolean expressions. The remaining axioms enforce the modelling of the behavior of programs in the model. Axiom (4.8) handles programs starting with a skip statement. Axiom (4.7) and (4.9)-(4.13) handle programs starting with a variable assignment. Of these, Axiom (4.13) can be considered as a frame axiom, making sure that unmentioned variables are unchanged. Axiom (4.14) handles programs starting with if-then-statements, and Axiom (4.15) handles programs starting with while-statements.

4.3.2 Semantic Correspondence

In order to use the above encoding of a program p into an $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^p$, we show the following correspondence between the operational semantics of p and the model theoretic semantics of $\mathcal{T}^p$.

Lemma 4.3.1 (Boolean correspondence)**.** *For any program p, any X such that $Var(p) \subseteq X \subseteq \mathcal{X}$, any state $s \in \mathcal{S}_X$, any $b \in Bool(p)$, and for any model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^p$, we have that $d \in \Delta^{\mathcal{I}}$ and $(d, s(x_i)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$ for all $1 \leq i \leq n$ implies that $d \in D^{\mathcal{I}}_b$ iff $\mathcal{B}^X(b, s) = \top$.*

Proof. By induction on the structure of b. We know $\mathcal{I}$ is a model of $\mathcal{T}^b$. The base cases $b = \top$ and $b = \bot$ follow directly, since $D_\top = \top$ and $D_\bot = \bot$. The base cases $b = (a_1 = a_2)$ and $b = (a_1 \leq a_2)$ follow directly from the fact that Axioms (4.3) and (4.4) hold, respectively. The inductive cases $b = \neg b_1$ and $b = b_1 \wedge b_2$ follow directly from the fact that Axioms (4.5) and (4.6) hold, respectively. □

Theorem 4.3.2 (Semantic enforcement)**.** *For any program p, any X such that $Var(p) \subseteq X \subseteq \mathcal{X}$, any state $s \in \mathcal{S}_X$ such that p terminates on s with outcome t, and for any model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^p$ we have that $d \in C_p^{\mathcal{I}}$ and $(d, s(x_i)) \in \mathsf{valueOf}_{x_i}^{\mathcal{I}}$ for all $1 \leq i \leq n$ implies that $e \in C_{skip}^{\mathcal{I}}$ and $(e, t(x_i)) \in \mathsf{valueOf}_{x_i}^{\mathcal{I}}$ for all $1 \leq i \leq n$, for some $e \in \Delta^{\mathcal{I}}$.*

Proof. By induction on the length of the $\Rightarrow$-derivation $(p, s) \Rightarrow^k t$. Assume $d \in C_p^{\mathcal{I}}$ and $(d, s(x_i)) \in \mathsf{valueOf}_{x_i}^{\mathcal{I}}$ for all $1 \leq i \leq n$, for some $d \in \Delta^{\mathcal{I}}$. The base case $k = 0$ holds vacuously. In the case for $k = 1$, we know $p = skip$, since p is in normal form. Therefore, we know $s = t$, and thus $e = d$ witnesses the implication.

In the inductive case, we distinguish several cases. Case $p = skip; q$. We know $(p, s) \Rightarrow (q, s) \Rightarrow^{k-1} t$. Since $\mathcal{I}$ satisfies $\mathcal{T}^p$, by Axiom (4.8), we know $d \in C_q^{\mathcal{I}}$. The result now follows directly by the induction hypothesis.

Case $p = (x := a); q$. We know $(p, s) \Rightarrow (q, s') \Rightarrow^{k-1} t$, and $s' = s[x \mapsto \mathcal{A}^X(a, s)]$. Since $\mathcal{I}$ satisfies $\mathcal{T}^p$, by Axioms (4.1), (4.7), (4.9)-(4.12) and (4.13), we know there must exist a $d' \in C_q^{\mathcal{I}}$ such that $(d', s'(x_i)) \in \mathsf{valueOf}_{x_i}^{\mathcal{I}}$ for all $1 \leq i \leq n$. Then by the induction hypothesis, the result follows directly.

Case $p = (\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2); q$. Assume $\mathcal{B}^X(b, s) = \top$. Then $(p, s) \Rightarrow (p_1; q, s) \Rightarrow^{k-1} t$. By Lemma 4.3.1, we know $d \in D_b^{\mathcal{I}}$. Then, by the fact that Axiom (4.14) holds, we know $d \in C_{p_1;q}^{\mathcal{I}}$. The result now follows directly by the induction hypothesis. The case for $\mathcal{B}^X(b, s) = \bot$ can be shown by an analogous argument.

Case $p = (\textbf{while}\ b\ \textbf{do}\ p_1); q$. Assume $\mathcal{B}^X(b, s) = \top$. Then $(p, s) \Rightarrow^2 (p_1; p, s) \Rightarrow^{k-2} t$. By Lemma 4.3.1, we know $d \in D_b^{\mathcal{I}}$. Then, by the fact that Axiom (4.15) holds, we know $d \in C_{p_1;p}$. The result now follows directly by the induction hypothesis.

If, however, in the same case holds $\mathcal{B}^X(b, s) = \bot$, then $(p, s) \Rightarrow^3 (q, s) \Rightarrow^{k-3} t$. By Lemma 4.3.1, we know $d \notin D_b^{\mathcal{I}}$. By the fact that Axiom (4.15) holds, we know $d \in C_q^{\mathcal{I}}$. The result now follows directly by the induction hypothesis. □

In the following two theorems canonical models are constructed on the basis of (terminating and nonterminating) sequences.

Theorem 4.3.3 (Canonical model for nonterminating sequences)**.** *For any program p, any X such that $Var(p) \subseteq X \subseteq \mathcal{X}$, and any state $\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\} = s \in \mathcal{S}_X$ such that p does not terminate on s, there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^p$ such that for some $d \in \Delta^{\mathcal{I}}$ we have $d \in C_p^{\mathcal{I}}$, $(d, c_i) \in \mathsf{valueOf}_{x_i}^{\mathcal{I}}$, for all $1 \leq i \leq n$, and $C_{skip}^{\mathcal{I}} = \emptyset$.*

Proof. Since p does not terminate on s, we know there exists an infinite $\Rightarrow$-sequence d such that $(p_i, s_i) \Rightarrow (p_{i+1}, s_{i+1})$, for $i \in \mathbb{N}$, where $(p_1, s_1) = (p, s)$. We construct the canonical model of $\mathcal{T}^p$ for this infinite $\Rightarrow$-sequence: $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. Let D be the stepwise division of the derivation d (which contains either finitely many or infinitely many partial derivations $d_i \in D$). We let $\Delta^{\mathcal{I}} = D$. For $q \in cl(p)$, we let $C_q^{\mathcal{I}} = \{d \in D \mid (q, s) \in d\}$. For $b \in Bool(p)$, we let $D_b^{\mathcal{I}} = \{d \in D \mid (p, s) \in d, \mathcal{P}^X(b, s) = \top\}$. For each $x \in X$, we let $\mathsf{valueOf}_x^{\mathcal{I}} = \{(d', s'(x)) \mid d' \in D, s'$ the state corresponding to $d'\}$. We let $\mathsf{nextState}^{\mathcal{I}} = \{(d_i, d_{i+1}) \mid d_i, d_{i+1} \in D, d_i \Rightarrow d_{i+1}\}$.

The definition of $\mathcal{I}$ implies that $C^{\mathcal{I}}_{skip} = \emptyset$. Assume $d_i \in C^{\mathcal{I}}_{skip}$. Then for some $(p_k, s_k) \in d_i$ we would have $p_k = skip$, and thus $(p_k, s_k) \Rightarrow s_k$, which contradicts our assumption of nontermination.

Clearly, $\mathcal{I}$ satisfies Axiom (4.1). Since $C^{\mathcal{I}}_{skip} = \emptyset$, $\mathcal{I}$ also satisfies Axiom (4.2). It is easy to verify, that by the definition of $D^{\mathcal{I}}_b$ we get that $\mathcal{I}$ satisfies Axioms (4.3)-(4.6).

To see that $\mathcal{I}$ satisfies Axioms (4.7)-(4.15), we take an arbitrary class $C^{\mathcal{I}}_{p_j}$, take arbitrary $d_k \in C^{\mathcal{I}}_{p_j}$ with $(p_j, s_j) \in d_k$, and we distinguish several cases.

Consider $p_j = skip; q$. Then by the constraints on $\Rightarrow$, we know $(p_{j+1}, s_{j+1}) \in d_k$ where $p_{j+1} = q$ and $s_{j+1} = s_j$. By definition then also $d_k \in C^{\mathcal{I}}_q$. This witnesses that the subsumption in Axiom (4.8) holds.

Consider $p_j = (x := a); q$. Then by the constraints on $\Rightarrow$, we know $(p_{j+1}, s_{j+1}) \in d_{k+1}$, for the $d_{k+1} \in D$ such that $d_k \Rightarrow d_{k+1}$, where $p_{j+1} = q$ and $s_{j+1} = s_j[x \mapsto \mathcal{A}^X(a, s_j)]$. By definition of $\mathcal{I}$, we know $(d_k, d_{k+1}) \in \mathsf{nextState}^{\mathcal{I}}$. It is now easy to verify that the subsumptions in Axioms (4.7), (4.9)-(4.12) and (4.13) are satisfied.

Consider $p_j = (\textbf{if } b \textbf{ then } p'_1 \textbf{ else } p'_2); q$. Assume $\mathcal{B}^X(b, s_j) = \top$. Then $d_k \in D^{\mathcal{I}}_b$. Also, by the constraints on $\Rightarrow$, we know $(p_{j+1}, s_{j+1}) \in d_k$, where $p_{j+1} = p'_1; q$ and $s_{j+1} = s_j$. It is easy to verify that, in this case, the subsumption in Axiom (4.14) holds. The case for $\mathcal{B}^X(b, s_j) = \bot$ is completely analogous.

Consider $p_j = (\textbf{while } b \textbf{ do } p'); q$. If $\mathcal{B}^X(b, s_j) = \top$, then $d_k \in D^{\mathcal{I}}_b$ and $(p_{j+1}, s_{j+1}) \in d_k$, where $p_{j+1} = p'; p_j$ and $s_{j+1} = s_j$. If $\mathcal{B}^X(b, s_j) = \bot$, then $d_k \notin D^{\mathcal{I}}_b$ and $(p_{j+1}, s_{j+1}) \in d_k$, where $p_{j+1} = skip; q$ and $s_{j+1} = s_j$. It is easy to verify that, in either case, the subsumption in Axiom (4.15) holds. □

Note that the canonical model constructed in the proof of Theorem 4.3.3 is infinite, and might not be effectively constructable.

Theorem 4.3.4 (Canonical model for terminating sequences)**.** *For any program p, any X such that $Var(p) \subseteq X \subseteq \mathcal{X}$ any state $\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\} = s \in \mathcal{S}_X$ such that p terminates on s, there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^p$ such that for some $d \in \Delta^{\mathcal{I}}$ we have $d \in C^{\mathcal{I}}_p$, $(d, c_i) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$, for all $1 \leq i \leq n$.*

Proof (sketch). We know there exists a sequence $(p, s) \Rightarrow^k (p', s') \Rightarrow t$. Analogously to the proof of Theorem 4.3.3, we can construct the canonical model $\mathcal{I}$ of $\mathcal{T}^p$ for the sequence $(p, s) \Rightarrow^k (p', s') \Rightarrow t$ (by using the stepwise division). By similar arguments to those in the proof of Theorem 4.3.3 it follows that $\mathcal{I} \models \mathcal{T}^p$. Then, $d_1 \in C^{\mathcal{I}}_p$, where d_1 is the first partial derivation in the stepwise division of the derivation, for which we furthermore know $(p, s) \in d_1$, witnesses the further constraints on $\mathcal{I}$. □

The above results can intuitively be explained as follows. Lemma 4.3.1 shows us that the behavior of Boolean expressions is correctly captured in models of the encoding. This is then used in Theorem 4.3.2 to show that all models of the encoding (that satisfy some requirements) reflect the behavior of programs. Furthermore, Theorems 4.3.3 and 4.3.4 tell us that this is not a vacuous statement, and that there are in fact models witnessing the behavior of both terminating and nonterminating derivations.

Note that the syntax and operational semantics of *While* can be adapted to various concrete domains with varying operators. The encoding into $\mathcal{ALC}(\mathcal{D})$ can be adapted correspondingly, and a corresponding correlation between the operational semantics and the model theoretic semantics can be proven.

4.3.3 Encoding Reasoning Problems

Theorems 4.3.2, 4.3.3 and 4.3.4 allow us to use the encoding of *While* programs into $\mathcal{ALC}(\mathcal{D})$ to reduce various reasoning problems over *While* programs to reasoning problems over $\mathcal{ALC}(\mathcal{D})$. For instance, termination of a program p directly reduces to unsatisfiability of the concept C_p with respect to the TBox $\mathcal{T}^p \cup \{C_{skip} \sqsubseteq \bot\}$. Any nonterminating execution of p (and only nonterminating executions of p) will namely induce a model witnessing the satisfiability of C_p with respect to this TBox.

Another example is checking whether two (terminating) programs p_1 and p_2 are equivalent. This can be expressed in the description logic $\mathcal{ALCO}(\mathcal{D})$ (the extension of $\mathcal{ALC}(\mathcal{D})$ with nominal concepts). Without loss of generality, we can assume $Var(p_1) = Var(p_2)$. This equivalence problem can directly be reduced to the problem of $\mathcal{ALCO}(\mathcal{D})$ unsatisfiability of the ABox

$$\mathcal{A}^{p_1,p_2} = \{o : C_{p_1}, o : C_{p_2}, s : C_{test}\}$$

with respect to the TBox $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2} \cup \mathcal{T}^{eq}$ where $\mathcal{S}^{p_i}$ is the modification of $\mathcal{T}^{p_i}$ where C_{skip} is replaced by C^i_{skip} and $\mathsf{nextState}$ is replaced by $\mathsf{nextState}_i$, and

$$\begin{array}{lll}\mathcal{T}^{eq} = \{ & C_{test} \equiv & \exists \mathsf{res}_1.C^1_{skip} \sqcap \exists \mathsf{res}_2.C^2_{skip} \sqcap \\ & & (\exists (\mathsf{res}_1\ \mathsf{valueOf}_{x_1})(\mathsf{res}_2\ \mathsf{valueOf}_{x_1}).\neq \\ & & \sqcup \cdots \sqcup \\ & & \exists (\mathsf{res}_1\ \mathsf{valueOf}_{x_n})(\mathsf{res}_2\ \mathsf{valueOf}_{x_n}).\neq)\ \}\end{array}$$

for res_1, res_2 abstract features, and C^1_{skip}, C^2_{skip} and C_{test} nominal concepts. Models for this ABox and TBox correspond to derivations of p_1 and p_2 starting with the same input state, and resulting in at least one different output value.

In addition to the semantic properties of termination and equivalence, on which we focus, we are able to encode abduction problems over *While* problems in the description logic $\mathcal{ALCO}(\mathcal{D})$. The question what input states for a program p could have led to the (partial) output state s, for $dom(s) \subseteq Var(p)$, reduces to finding models for the ABox $\mathcal{A}^p = \{o : C_{skip}\} \cup \{(o, s(x)) : \mathsf{valueOf}_x \mid x \in dom(s)\}$ and the TBox $\mathcal{T}^p$, where C_{skip} is required to be a nominal concept.

Also, this approach allows us to encode more intricate reasoning problems over *While* programs into description logic reasoning problems. For instance, we can test whether the state after any execution of program p always assigns a value c to variable x. Description logic offers us a very flexible formalism to express a variety of reasoning problems over *While* programs.

4.3.4 Example

In order to illustrate how encoding *While* programs into $\mathcal{ALC}(\mathcal{D})$ TBoxes works, and to illustrate the statements of Theorems 4.3.2, 4.3.3 and 4.3.4, we consider an example. In Figure 4.1 a derivation for the example program $p = (x := 0; (\textbf{while } (x < 2) \textbf{ do } x := x + 1); skip)$ is given for an example state $s \in \mathcal{S}_{\{x\}}$ with $s(x) = 5$. Also, in this same figure, the encoding $\mathcal{T}^p$ of p into $\mathcal{ALC}(\mathcal{D})$ is given, as well as a model of $\mathcal{T}^p$ corresponding to the example derivation.

4.4 Encoding *Goto* Programs in $\mathcal{ALC}(\mathcal{D})$

We now show how to model the behavior of programs of the language *Goto* using description logic. The concrete values in the programming language correspond to concrete values in the description logic. States will be represented by objects, and (lines of) programs are represented by concepts. We represent the execution of programs on states by an abstract feature nextState.

In particular, for a given program κ with $Var(\kappa) = X = \{x_1, \dots, x_n\}$, we denote states $s \in \mathcal{S}_X$ with objects that have concrete features $\mathsf{valueOf}_{x_i}$ for each $1 \leq i \leq n$.

4.4.1 Constructing a TBox

Take an arbitrary *Goto* program κ of length l. W.l.o.g., we assume κ is in normal form. We define an $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^\kappa$ as follows. We use concept names $C_{\kappa,n}$ for each $1 \leq n \leq l$, and concept names D_b for each $b \in Bool(\kappa)$.

For each variable $x \in Var(\kappa)$, we create a concrete feature $\mathsf{valueOf}_x$, and we require

$$\top \sqsubseteq \neg \mathsf{valueOf}_x \uparrow \tag{4.16}$$

We let $\mathsf{nextState}$ be an abstract feature. For each $1 \leq n \leq l$ such that $\kappa(n) = return$, we require:

$$C_{\kappa,n} \sqsubseteq C_{\kappa,return} \tag{4.17}$$

Furthermore, we require:

$$C_{\kappa,return} \sqsubseteq \neg \exists \mathsf{nextState}.\top \tag{4.18}$$

For each $b \in Bool(\kappa)$, we require the following, where x_1, x_2 range over X, and b_1, b_2 range over $Bool(\kappa)$:

$$\begin{aligned} D_{x_1 = x_2} &\equiv \exists (\mathsf{valueOf}_{x_1})(\mathsf{valueOf}_{x_2}).= && (4.19) \\ D_{x_1 \leq x_2} &\equiv \exists (\mathsf{valueOf}_{x_1})(\mathsf{valueOf}_{x_2}).\leq && (4.20) \\ D_{\neg b_1} &\equiv \neg D_{b_1} && (4.21) \\ D_{b_1 \wedge b_2} &\equiv D_{b_1} \sqcap D_{b_2} && (4.22) \end{aligned}$$

Furthermore, we let $D_\top$ denote $\top$ and $D_\bot$ denote $\bot$. For each $1 \leq n \leq l$ such that $\kappa(n) =$

Example *While* program p (with subprogram p').

$$p = \quad (x := 0; \overbrace{(\textbf{while}\ (x < 2)\ \textbf{do}\ x := x + 1); \mathit{skip}}^{p'})$$

(Compact) derivation of p for the state $s \in \mathcal{S}_{\{x\}}$ with $s(x) = 5$.
States s' with $s'(x) = n$ are represented as $[n]$.

$$\underset{p}{\overset{[5]}{\cdot}} \longrightarrow \underset{p'}{\overset{[0]}{\cdot}} \xrightarrow{*} \underset{x := x + 1; p'}{\overset{[0]}{\cdot}} \longrightarrow \underset{p'}{\overset{[1]}{\cdot}} \xrightarrow{*} \underset{x := x + 1; p'}{\overset{[1]}{\cdot}} \longrightarrow \underset{p'}{\overset{[2]}{\cdot}} \xrightarrow{*} \underset{\mathit{skip}}{\overset{[2]}{\cdot}}$$

$\mathcal{ALC}(\mathcal{D})$ encoding $\mathcal{T}^p$ of the program p.
We do a bit of handwaving: we do not transform the program into normal form, but instead use predicates $\leq_2 = \{0, 1, 2\}$ and $+=_1 = \{(n, n + 1) \mid n \in \mathbb{N}\}$.

$$\begin{array}{llcl}
\mathcal{T}^p = \{ & C_p & \sqsubseteq & \neg\mathsf{valueOf}_x\uparrow, \\
& C_{p'} & \sqsubseteq & \neg\mathsf{valueOf}_x\uparrow, \\
& C_{x:=x+1;p'} & \sqsubseteq & \neg\mathsf{valueOf}_x\uparrow, \\
& C_{skip} & \sqsubseteq & \neg\mathsf{nextState}.\top, \\
& D_{x<2} & \equiv & \exists(\mathsf{valueOf}_x).\leq_2, \\
& C_{x:=0;p'} & \sqsubseteq & \exists\mathsf{nextState}.C_{p'}, \\
& C_{x:=0;p'} & \sqsubseteq & \exists(\mathsf{nextState}\ \mathsf{valueOf}_x).=_0, \\
& C_{p'} & \sqsubseteq & (\neg D_{x<2} \sqcup C_{x:=x+1;p'}) \sqcap (D_{x<2} \sqcup C_{skip}), \\
& C_{x:=x+1;p'} & \sqsubseteq & \exists\mathsf{nextState}.C_{p'}, \\
& C_{x:=x+1;p'} & \sqsubseteq & \exists(\mathsf{nextState}\ \mathsf{valueOf}_x)(\mathsf{valueOf}_x).+=_1\ \}
\end{array}$$

$\mathcal{ALC}(\mathcal{D})$ model of the encoding $\mathcal{T}^p$ corresponding to the above derivation:

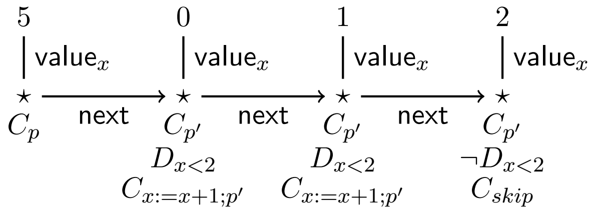

Figure 4.1: Example *While* program p, together with an example derivation of the operational semantics, its encoding into $\mathcal{ALC}(\mathcal{D})$, and a model corresponding to the given derivation.

$(x := a)$, we require:

$$
\begin{aligned}
&C_{\kappa,n} \sqsubseteq \exists(\mathsf{nextState}\ \mathsf{valueOf}_x).{=_n} && \text{if } \kappa(n) = (x := n) && (4.23)\\
&C_{\kappa,n} \sqsubseteq \exists(\mathsf{nextState}\ \mathsf{valueOf}_x)(\mathsf{valueOf}_y).{=} && \text{if } \kappa(n) = (x := y) && (4.24)\\
&C_{\kappa,n} \sqsubseteq \exists(\mathsf{nextState}\ \mathsf{valueOf}_x)(\mathsf{valueOf}_{y_1})(\mathsf{valueOf}_{y_2}).{+} && \text{if } \kappa(n) = (x := y_1 + y_2) && (4.25)\\
&C_{\kappa,n} \sqsubseteq (\neg\exists(\mathsf{valueOf}_{y_2})(\mathsf{valueOf}_{y_1}).{\leq} \sqcup \\
&\qquad \exists(\mathsf{valueOf}_{y_1})(\mathsf{nextState}\ \mathsf{valueOf}_x)(\mathsf{valueOf}_{y_2}).{+}) \sqcap \\
&\qquad (\neg\exists(\mathsf{valueOf}_{y_2})(\mathsf{valueOf}_{y_1}).{>} \sqcup \\
&\qquad \exists(\mathsf{nextState}\ \mathsf{valueOf}_x).{=}\, 0) && \text{if } \kappa(n) = (x := y_1 - y_2) && (4.26)\\
&C_{\kappa,n} \sqsubseteq \exists(\mathsf{valueOf}_y)(\mathsf{nextState}\ \mathsf{valueOf}_y).{=} && \text{if } \kappa(n) = (x := a) && \\
& && \text{and } y \neq x && (4.27)\\
&C_{\kappa,n} \sqsubseteq \exists\mathsf{nextState}.C_{\kappa,n+1} && && (4.28)
\end{aligned}
$$

For each $1 \leq n \leq l$ such that $\kappa(n) =$ (**if** b **goto** n_1 **else** n_2), we require:

$$C_{\kappa,n} \sqsubseteq (\neg D_b \sqcup C_{\kappa,n_1}) \sqcap (D_b \sqcup C_{\kappa,n_2}) \tag{4.29}$$

Notice that, in general, the TBox $\mathcal{T}^\kappa$ is not acyclic, since combinations of Axioms (4.28) and (4.29) can induce a cycle.

Intuitively, these axioms serve the following purpose. Axioms (4.16)-(4.18) ensure some basic properties of the modelling of derivations in the model. Axioms (4.19)-(4.22) capture the behavior of Boolean expressions. The remaining axioms enforce the modelling of the behavior of programs in the model. Axiom (4.23)-(4.28) handle programs starting with a variable assignment. Of these, Axiom (4.27) can be considered as a frame axiom, making sure that unmentioned variables are unchanged. Axiom (4.29) handles programs starting with conditional goto statements.

4.4.2 Semantic Correspondence

In order to use the above encoding of a program κ into an $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^\kappa$, we show the following correspondence between the operational semantics of κ and the model theoretic semantics of $\mathcal{T}^\kappa$.

Lemma 4.4.1 (Boolean correspondence)**.** *For any program κ, any X such that $Var(\kappa) \subseteq X \subseteq \mathcal{X}$, any state $s \in \mathcal{S}_X$, any $b \in Bool(\kappa)$, and for any model $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ of $\mathcal{T}^\kappa$, we have that $d \in \Delta^\mathcal{I}$ and $(d, s(x_i)) \in \mathsf{valueOf}^\mathcal{I}_{x_i}$ for all $1 \leq i \leq n$ implies that $d \in D^\mathcal{I}_b$ iff $\mathcal{B}^X(b, s) = \top$.*

Proof. By induction on the structure of b. We know $\mathcal{I}$ is a model of $\mathcal{T}^\kappa$. The base cases $b = \top$ and $b = \bot$ follow directly, since $D_\top = \top$ and $D_\bot = \bot$. The base cases $b = (a_1 = a_2)$ and $b = (a_1 \leq a_2)$ follow directly from the fact that Axioms (4.19) and (4.20) hold, respectively. The inductive cases $b = \neg b_1$ and $b = b_1 \wedge b_2$ follow directly from the fact that Axioms (4.21) and (4.22) hold, respectively. □

Theorem 4.4.2 (Semantic enforcement)**.** *For any program κ, any X such that $Var(\kappa) \subseteq X \subseteq \mathcal{X}$, any state $s \in \mathcal{S}_X$ such that κ terminates on s with outcome t, and for any model $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$*

of $\mathcal{T}^\kappa$ we have that $d \in C^{\mathcal{I}}_{\kappa,1}$ and $(d, s(x_i)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$ for all $1 \leq i \leq n$ implies that $e \in C^{\mathcal{I}}_{\kappa,return}$ and $(e, t(x_i)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$ for all $1 \leq i \leq n$, for some $e \in \Delta^{\mathcal{I}}$.

Proof. By induction on the length of the $\Rightarrow$-derivation $(m, s) \Rightarrow^k t$. Assume $d \in C^{\mathcal{I}}_p$ and $(d, s(x_i)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$ for all $1 \leq i \leq n$, for some $d \in \Delta^{\mathcal{I}}$. The base case for $k = 0$ holds vacuously. In the base case for $k = 1$, we know $\kappa(m) = return$, and thus $s = t$, and $e = d$ witnesses the result, since $\mathcal{I}$ satisfies Axiom (4.17).

In the inductive case, we distinguish several cases. Case $\kappa(m) = (x := a)$. We know $(m, s) \Rightarrow (m + 1, s') \Rightarrow^{k-1} t$, and $s' = s[x \mapsto \mathcal{A}^X(a, s)]$. Since $\mathcal{I}$ satisfies $\mathcal{T}^\kappa$, by Axioms (4.23)-(4.26), (4.27) and (4.28), we know there must exist a $d' \in C^{\mathcal{I}}_{\kappa,m+1}$ such that $(d', s'(x_i)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$ for all $1 \leq i \leq n$. Then by the induction hypothesis, the result follows directly.

Case $\kappa(m) = (\textbf{if}\, b\, \textbf{goto}\, m_1\, \textbf{else}\, m_2)$. Assume $\mathcal{B}^X(b, s) = \top$. Then $(m, s) \Rightarrow (m_1, s) \Rightarrow^{k-1} t$. By Lemma 4.4.1, we know $d \in D^{\mathcal{I}}_b$. Then, by the fact that Axiom (4.29) holds, we know $d \in C_{\kappa,m_1}$. The result now follows directly by the induction hypothesis.

If, however, in the same case holds $\mathcal{B}^X(b, s) = \bot$, then $(m, s) \Rightarrow (m_2, s) \Rightarrow^{k-1} t$. By Lemma 4.4.1, we know $d \notin D^{\mathcal{I}}_b$. By the fact that Axiom (4.29) holds, we know $d \in C^{\mathcal{I}}_{\kappa,m_2}$. The result now follows directly by the induction hypothesis. □

In the following two theorems canonical models are constructed on the basis of (terminating and nonterminating) sequences.

Theorem 4.4.3 (Canonical model for nonterminating sequences)**.** *For any program κ, any X such that $Var(\kappa) \subseteq X \subseteq \mathcal{X}$ any state $\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\} = s \in \mathcal{S}_X$ such that κ does not terminate on s, there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^\kappa$ such that for some $d \in \Delta^{\mathcal{I}}$ we have $d \in C^{\mathcal{I}}_{\kappa,1}$, $(d, c_i) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$, for all $1 \leq i \leq n$, and $C^{\mathcal{I}}_{\kappa,return} = \emptyset$.*

Proof. Since κ does not terminate on s, we know there exists an infinite $\Rightarrow$-sequence d (m_1, s_i) $\Rightarrow_\kappa$ (m_{i+1}, s_{i+1}), for $i \in \mathbb{N}$, where $(m_1, s_1) = (1, s)$. We construct the canonical model of $\mathcal{T}^\kappa$ for this infinite $\Rightarrow$-sequence: $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. Let D be the stepwise division of the derivation d (which contains either finitely many or infinitely many partial derivations $d_i \in D$). We let $\Delta^{\mathcal{I}} = D$. For $1 \leq m \leq l$, we let $C^{\mathcal{I}}_{\kappa,m} = \{d \in D \mid ((m, s) \in d\}$. For $b \in Bool(p)$, we let $D^{\mathcal{I}}_b = \{d \in D \mid (m, s) \in d, \mathcal{P}^X(b, s) = \top\}$. For each $x \in X$, we let $\mathsf{valueOf}^{\mathcal{I}}_x = \{(d', s'(x)) \mid d' \in D, s'$ the state corresponding to $d'\}$. We let $\mathsf{nextState}^{\mathcal{I}} = \{(d_i, d_{i+1}) \mid d_i, d_{i+1} \in D, d_i \Rightarrow d_{i+1}\}$.

The definition of $\mathcal{I}$ implies that $C^{\mathcal{I}}_{\kappa,return} = \emptyset$. Assume $d_i \in C^{\mathcal{I}}_{\kappa,return}$. Then for some $(m_k, s_k) \in d_i$ we would have $\kappa(m_k) = return$, and thus $(m_k, s_k) \Rightarrow s_k$, which contradicts our assumption of nontermination.

Clearly, $\mathcal{I}$ satisfies Axiom (4.16). It is also easy to verify, since $C_{\kappa,return} = \emptyset$, that $\mathcal{I}$ satisfies Axioms (4.17) and (4.18). Also, by the definition of $D^{\mathcal{I}}_b$ we get that $\mathcal{I}$ satisfies Axioms (4.19)-(4.22).

To see that $\mathcal{I}$ satisfies Axioms (4.23)-(4.29), we take an arbitrary class $C^{\mathcal{I}}_{\kappa,m_j}$, an arbitrary object d_k in the interpretation with $(m_j, s_j) \in d_k$, and we distinguish several cases. Consider $\kappa(m_j) = (x := a)$. Then by the constraints on $\Rightarrow$, we know that $(m_{j+1}, s_{j+1}) \in d_{k+1}$ for the $d_{k+1} \in D$ such that $d_k \Rightarrow d_{k+1}$ where $m_{j+1} = m_j + 1$ and $s_{j+1} = s_j[x \mapsto \mathcal{A}^X(a, s_j)]$. By definition of $\mathcal{I}$, we know $(d_k, d_{k+1}) \in \mathsf{nextState}^{\mathcal{I}}$. It is now easy to verify that the subsumptions in Axioms (4.23)-(4.28) are satisfied.

Consider $\kappa(m_j) =$ (**if** b **goto** m'_1 **else** m'_2). Assume $\mathcal{B}^X(b, s_j) = \top$. Then $d_k \in D_b^{\mathcal{I}}$. Also, by the constraints on $\Rightarrow$, we know that $(m_{j+1}, s_{j+1}) \in d_{k+1}$ for the $d_{k+1} \in D$ such that $d_k \Rightarrow d_{k+1}$ where $m_{j+1} = m'_1$ and $s_{j+1} = s_j$. It is easy to verify that, in this case, the subsumption in Axiom (4.29) holds. The case for $\mathcal{B}^X(b, s_j) = \bot$ is completely analogous. □

Theorem 4.4.4 (Canonical model for terminating sequences)**.** *For any program κ, any X such that $Var(\kappa) \subseteq X \subseteq \mathcal{X}$ any state $\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\} = s \in \mathcal{S}_X$ such that κ terminates on s, there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{\kappa}$ such that for some $d \in \Delta^{\mathcal{I}}$ we have $d \in C_{\kappa,1}^{\mathcal{I}}$, $(d, c_i) \in \mathsf{valueOf}_{x_i}^{\mathcal{I}}$, for all $1 \leq i \leq n$.*

Proof (sketch). We know there exists a sequence $(1, s) \Rightarrow^k (m', s') \Rightarrow t$. Analogously to the proof of Theorem 4.4.3, we can construct the canonical model $\mathcal{I}$ of $\mathcal{T}^{\kappa}$ for the sequence $(1, s) \Rightarrow^k (m', s')$ (by using the stepwise division). By similar arguments to those in the proof of Theorem 4.4.3 it follows that $\mathcal{I} \models \mathcal{T}^{\kappa}$. Then, $d_1 \in C_{\kappa,1}^{\mathcal{I}}$, where d_1 is the first partial derivation in the stepwise division of the derivation, for which we furthermore know $(1, s) \in d_1$, witnesses the further constraints on $\mathcal{I}$. □

The above results can intuitively be explained as follows. Lemma 4.4.1 shows us that the behavior of Boolean expressions is correctly captured in models of the encoding. This is then used in Theorem 4.4.2 to show that all models of the encoding (that satisfy some requirements) reflect the behavior of programs. Furthermore, Theorems 4.4.3 and 4.4.4 tell us that this is not a vacuous statement, and that there are in fact models witnessing the behavior of both terminating and nonterminating derivations.

4.4.3 Encoding Reasoning Problems

Since Theorems 4.4.2, 4.4.3 and 4.4.4 are completely analogous to Theorems 4.3.2, 4.3.3 and 4.3.4, they allow us to reduce reasoning problems over *Goto* programs to reasoning problems over $\mathcal{ALC}(\mathcal{D})$, similarly to the reduction of reasoning problems over *While* programs in Section 4.3.3. By changing concept names from C_p to $C_{\kappa,1}$ for *Goto* programs κ, and by changing concept names from C_{skip} to C_{return}, we can directly convert the encodings of reasoning problems over *While* programs into encodings of reasoning problems over *Goto* programs.

4.5 Encoding *FPN* Programs into $\mathcal{ALC}(\mathcal{D})$

We now show how to model the behavior of programs of the language *FPN* using description logic. We let definitions of functional symbols f in programs be represented by concepts E^f. Tuples of natural numbers in the relational semantics of programs will be represented by objects in the interpretation of such concepts E^f. Furthermore, for such definitions of functional symbols, we use (indexed) concepts C^f to represent conditions and (indexed) concepts S^f to represent consequents. In order to represent the values of (subterms of) consequents, we use concepts T_s, (indexed) abstract features hasArg, and a number of concrete features.

4.5.1 Constructing a TBox

Given a program π over the function symbols F, we define a TBox $\mathcal{T}^{\pi}$ as follows. For each $f \in F$, we introduce a concept E^f. For each condition c occurring in the sequence $\pi(f)$ for any $f \in F$, we introduce a concept C^f_c. Furthermore, for each such condition $c = (d_1, \ldots, d_m)$, we let the (concept) expression D_{d_i} be $\exists(\mathsf{hasInput}_{x_i}).{=_{d_i}}$ if $d_i \in \mathbb{N}$ and $\top$ if $d_i = \bot$. Now, for the condition $c = (d_1, \ldots, d_m)$ occurring in $\pi(f)$ we require

$$C^f_c \equiv D_{d_1} \sqcap \cdots \sqcap D_{d_m} \tag{4.30}$$

For each consequent e occurring in the sequence $\pi(f)$ for any $f \in F$, we introduce a concept S^f_e. Furthermore, for each $f \in F$ and $\pi(f) = ((c_1, e_1), \ldots, (c_n, e_n))$ we require

$$\begin{aligned} E^f \sqsubseteq\ & (\neg C^f_{c_1} \sqcup S^f_{e_1}) \sqcap \\ & (C^f_{c_1} \sqcup \neg C^f_{c_2} \sqcup S^f_{e_2}) \sqcap \\ & \ldots \\ & (C^f_{c_1} \sqcup \cdots \sqcup C^f_{c_{n-1}} \sqcup \neg C^f_{c_n} \sqcup S^f_{e_n}) \sqcap \\ & (C^f_{c_1} \sqcup \cdots \sqcup C^f_{c_n} \sqcup \mathsf{hasOutput}{\uparrow}) \end{aligned} \tag{4.31}$$

For each consequent e occurring in the sequence $\pi(f)$ for any $f \in F$, and each subterm s of e, we introduce a concept T_s, and we require the following, where ρ ranges over $\{+, -\}$.

$$T_s \sqsubseteq \neg\exists\mathsf{hasArg}_j \sqcup \exists(\mathsf{hasInput}_i)(\mathsf{hasArg}_j\ \mathsf{hasInput}_i).{=} \tag{4.32}$$

$$T_n \sqsubseteq \exists\mathsf{hasValue}.{=_n} \tag{4.33}$$

$$T_{x_i} \sqsubseteq \exists(\mathsf{inputValue}_i)(\mathsf{hasValue}).{=} \tag{4.34}$$

$$T_{t_1\ \rho\ t_2} \sqsubseteq \exists\mathsf{hasArg}_1.T_{t_1} \sqcap \exists\mathsf{hasArg}_2.T_{t_2} \tag{4.35}$$

$$\begin{aligned} T_{t_1 - t_2} \sqsubseteq\ & (\neg\exists(\mathsf{hasArg}_1\ \mathsf{hasValue})(\mathsf{hasArg}_2\ \mathsf{hasValue}).{\geq} \\ & \sqcup \exists(\mathsf{hasValue}).{=_0}) \sqcap \\ & (\neg\exists(\mathsf{hasArg}_1\ \mathsf{hasValue})(\mathsf{hasArg}_2\ \mathsf{hasValue}).{<} \\ & \sqcup \exists(\mathsf{hasValue})(\mathsf{hasArg}_1\ \mathsf{hasValue})(\mathsf{hasArg}_2\ \mathsf{hasValue}).{-}) \end{aligned} \tag{4.36}$$

$$T_{t_1 + t_2} \sqsubseteq \exists(\mathsf{hasValue})(\mathsf{hasArg}_1\ \mathsf{hasValue})(\mathsf{hasArg}_2\ \mathsf{hasValue}).{+} \tag{4.37}$$

$$T_{f(t_1, \ldots, t_n)} \sqsubseteq \exists\mathsf{hasArg}_1.T_{t_1} \sqcap \cdots \sqcap \exists\mathsf{hasArg}_n.T_{t_n} \tag{4.38}$$

$$T_{f(t_1, \ldots, t_n)} \sqsubseteq \exists\mathsf{hasArg}.E_f \tag{4.39}$$

$$T_{f(t_1, \ldots, t_n)} \sqsubseteq \exists(\mathsf{hasArg}\ \mathsf{inputValue}_i)(\mathsf{hasArg}_i\ \mathsf{hasValue}).{=} \tag{4.40}$$

$$\begin{aligned} T_{f(t_1, \ldots, t_n)} \sqsubseteq\ & (\exists(\mathsf{hasValue})(\mathsf{hasArg}\ \mathsf{outputValue}).{=}) \sqcup \\ & (\mathsf{hasValue}{\uparrow} \sqcap (\mathsf{hasArg}\ \mathsf{outputValue}){\uparrow}) \end{aligned} \tag{4.41}$$

Finally, we require for each consequent e occurring in the sequence $\pi(f)$ for any $f \in F$

$$S^f_e \sqsubseteq \exists\mathsf{hasArg}.T_e \tag{4.42}$$

$$S^f_e \sqsubseteq \exists(\mathsf{outputValue})(\mathsf{hasArg}\ \mathsf{hasValue}).{=} \tag{4.43}$$

$$S^f_e \sqsubseteq \exists(\mathsf{inputValue}_i)(\mathsf{hasArg}\ \mathsf{hasInput}_i).{=} \tag{4.44}$$

Here all roles are functional. Note that the TBox $\mathcal{T}^\pi$ is not acyclic, in general, since (combinations of) Axioms (4.31), (4.42) and (4.38)-(4.39) can induce cycles.

Intuitively, these axioms serve the following purpose. Axiom (4.30) enforces the correct behavior of conditions in the modelling. Axiom (4.31) ensures that for each definition in the program the right case is selected in the modelling, i.e. the first case that matches the conditions. Axiom (4.32)-(4.41) make sure the value of expressions occurring in the program are computed (inductively). Finally, Axioms (4.42)-(4.44) connect the definitions in the program to expressions occuring in them.

4.5.2 Semantic Correspondence

In order to use the above encoding of a program π into an $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^\pi$, we show the following correspondence between the semantics of π and the model theoretic semantics of $\mathcal{T}^\pi$.

Lemma 4.5.1 (Correspondence of conditions)**.** *For any program π, any $f \in F$ of arity k and any model $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ of $\mathcal{T}^\pi$, it holds that for any $(n_1, \dots, n_k) \in \mathbb{N}^k$, any condition $(c_1, \dots, c_k)$ of length k occurring in $\pi(f)$, and any $d \in \Delta^\mathcal{I}$ such that $(d, n_i) \in \mathsf{hasInput}_{x_i}^\mathcal{I}$, for all $1 \leq i \leq k$, we have $d \in (C_c^f)^\mathcal{I}$ iff condition $(c_1, \dots, c_k)$ matches $(n_1, \dots, n_k)$.*

Proof. This follows immediately by the definition of matching of conditions and the fact that $\mathcal{I}$ satisfies $\mathcal{T}^\pi$ and thus Axiom (4.30). □

Lemma 4.5.2 (Selection of conditions)**.** *For any program π, any $f \in F$ with arity k and any model $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ of $\mathcal{T}^\pi$, it holds that for any $\overline{n} = (n_1, \dots, n_k) \in \mathbb{N}^k$, and any $d \in \Delta^\mathcal{I}$ such that $(d, n_i) \in \mathsf{hasInput}_{x_i}^\mathcal{I}$, for all $1 \leq i \leq k$, the existence of a (c_i, e_i) in the sequence $\pi(f) = ((c_1, e_1), \dots, (c_m, e_m))$ such that c_i matches $\overline{n}$ and c_j does not match $\overline{n}$ for all $1 \leq j < i$, implies that we have that $d \in (E^f)^\mathcal{I}$ implies $d \in (S_{e_i}^f)^\mathcal{I}$; and the nonexistence of such a (c_i, e_i) implies $(d, n) \notin (\mathsf{hasOutput})^\mathcal{I}$ for all $n \in \mathbb{N}$.*

Proof. Assume there exists a suitable $d \in \Delta^\mathcal{I}$ such that $d \in (E^f)^\mathcal{I}$. Also, assume there exists a suitable (c_i, e_i) in $\pi(f)$. By Lemma 4.5.1, we know $d \notin (C_{c_j}^f)^\mathcal{I}$ for all $1 \leq j < i$. Then, by the fact that $\mathcal{I}$ satisfies Axiom (4.31), we know $d \in (S_{e_i}^f)^\mathcal{I}$.

Now, assume that no (c_i, e_i) exists such that c_i matches $\overline{n}$. By Lemma 4.5.1, we know $d \notin (C_{c_i}^f)^\mathcal{I}$ for all c_i. Then, by the fact that $\mathcal{I}$ satisfies Axiom (4.31), we know $d \in (\mathsf{hasOutput}\uparrow)^\mathcal{I}$, and the result follows. □

Theorem 4.5.3 (Semantic enforcement)**.** *For any program π, any $f \in F$ with arity k, any model $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ of $\mathcal{T}^\pi$, any $(n_1, \dots, n_k, n) \in Sem_f$, and any object $d \in \Delta^\mathcal{I}$ such that $(d, n_i) \in \mathsf{inputValue}_i^\mathcal{I}$ for all $1 \leq i \leq k$, we have that $d \in (E^f)^\mathcal{I}$ implies $(d, n) \in \mathsf{outputValue}^\mathcal{I}$.*

Proof. Take an arbitrary $(n_1, \dots, n_k, n) \in Sem_f$. By definition, it must be in Sem_f^j, for some $j \in \mathbb{N}$. We prove the result for all Sem_f^j, by induction on j. Take an arbitrary object $d \in (E^f)^\mathcal{I}$ such that $(d, n_i) \in \mathsf{inputValue}_i^\mathcal{I}$ for all $1 \leq i \leq k$. By the fact that $(n_1, \dots, n_k, n) \in Sem_f^j$, we know there exists a (c_z, e_z) in the sequence $\pi(f) = ((c_1, e_1), \dots, (c_m, e_m))$ such that c_i does not match $(n_1, \dots, n_k)$ for all $1 \leq i < z$. Then, by Lemma 4.5.2, we know that $d \in (S_{e_z}^f)^\mathcal{I}$.

Now, by the fact that $\mathcal{I}$ satisfies Axioms (4.42)-(4.44), we know that there exists a $d' \in T^{\mathcal{I}}_{e_z}$ with $(d, d') \in \mathsf{hasArg}^{\mathcal{I}}$ and $(d', n_i) \in \mathsf{hasInput}^{\mathcal{I}}_i$, and that it suffices to show that $(d', n') \in \mathsf{hasValue}^{\mathcal{I}}$, for $n' = \mathcal{J}^j_{\overline{n}}(e_z)$. Note that in order to show this, we can make use of the induction hypothesis for Sem^j_f, for $j' < j$.

We prove that the property holds for suitable d' by induction on the structure of e_z. The base cases for $e_z \in \mathbb{N}$ or $e_z = x_i$ for some x_i are immediate, by the fact that $\mathcal{I}$ satisfies Axioms (4.33) and (4.34). The inductive cases for $e_z = e_1 \; \rho \; e_2$, for $\rho \in \{+, -\}$, follow immediately from the induction hypothesis and the fact that $\mathcal{I}$ satisfies Axioms (4.35)-(4.37).

Consider the inductive case for $e_z = f(t_1, \ldots, t_n)$. By the induction hypothesis, and the fact that $\mathcal{I}$ satisfies Axioms (4.32) and (4.38), we know there exist $d'_i \in T^{\mathcal{I}}_{t_i}$ with $(d', d'_i) \in \mathsf{hasArg}^{\mathcal{I}}_i$ and $(d'_i, n'_i) \in \mathsf{hasInput}^{\mathcal{I}}_i$ for $n'_i = \mathcal{J}^j_{\overline{n}}(t_i)$. Then, by the fact that $\mathcal{I}$ satisfies Axioms (4.39) and (4.40), we know there exists a $e \in (E^f)^{\mathcal{I}}$ with $(e, n'_i) \in \mathsf{inputValue}^{\mathcal{I}}_i$. Now, by the induction hypothesis (of the outermost induction) we know that $(e, n') \in \mathsf{outputValue}^{\mathcal{I}}$, where $n' = \mathcal{J}^j_{\overline{n}}(e_z)$, since $(\mathcal{J}^i_{\overline{n}}(t_1), \ldots, \mathcal{J}^i_{\overline{n}}(t_n), n') \in Sem^{j'}_f$ for some $j' < j$. Now, by the fact that $\mathcal{I}$ satisfies Axiom (4.41), we know $(d', n') \in \mathsf{hasValue}^{\mathcal{I}}$. □

In the following theorem, a canonical model is constructed on the basis of the relational semantics of programs.

Theorem 4.5.4 (Canonical model)**.** *For any program π, any $f \in F$ with arity k, any $(n_1, \ldots, n_k, n) \in Sem_f$, there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{\pi}$, such that there exists $d \in \Delta^{\mathcal{I}}$ with $d \in (E^f)^{\mathcal{I}}$, $(d, n_i) \in \mathsf{inputValue}^{\mathcal{I}}_i$ for all $1 \leq i \leq k$, and $(d, n) \in \mathsf{outputValue}^{\mathcal{I}}$.*

Proof. We define the canonical model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{\pi}$, which is a suitable model. We let Y denote the set of all subterms e' of terms e occurring as a consequent in $\pi(f)$ for some $f \in F$. Let $\Delta^{\mathcal{I}} = \{(x, f) \mid f \in F, x \in Sem_f\} \cup \{(n_1, \ldots, n_{ar(f)}, \dagger, f) \mid f \in F, \forall n' \in \mathbb{N}.(n_1, \ldots, n_{ar(f)}, n') \notin Sem_f\} \cup \mathbb{N}^* \times (\mathbb{N}^* \cup \{\dagger\}) \times Y$. For each $f \in F$ and each $(\overline{n}, n) \in Sem_f$, we let $(\overline{n}, n, f) \in (E^f)^{\mathcal{I}}$. Also, for each $f \in F$ and each $(n_1, \ldots, n_k, \dagger, f) \in \Delta^{\mathcal{I}}$, we let $(n_1, \ldots, n_k, \dagger, f) \in (E^f)^{\mathcal{I}}$. Also, we define the interpretation $(C^f_c)^{\mathcal{I}}$ of a concept C^f_c to be the set of those $(\overline{n}, n, f)$ for $(\overline{n}, n) \in Sem_f$ and those $(\overline{n}, \dagger, f) \in \Delta^{\mathcal{I}}$ such that $\overline{n}$ matches c. Furthermore, we define the interpretation $(S^f_e)^{\mathcal{I}}$ of concepts S^f_e to be the set of those $(\overline{n}, n, f)$ for $(\overline{n}, n) \in Sem_f$ and those $(\overline{n}, \dagger, f) \in \Delta^{\mathcal{I}}$ such that $\pi(f) = ((c_1, e_1), \ldots, (c_n, e_n), (c, e), (c'_1, e'_1), \ldots, (c'_m, e'_m))$ for which $c_1, \ldots, c_n$ do not match $\overline{n}$ and c does match $\overline{n}$. Then, for each $\overline{n} \in \mathbb{N}^*$, for each $f \in F$ and for each e' occurring as a subterm of some consequent e in $\pi(f)$, we let $(\overline{n}, n', e') \in T^{\mathcal{I}}_{e'}$ if $n' = \mathcal{J}_{\overline{n}}(e')$. Also, for each $\overline{n} \in \mathbb{N}^*$, for each $f \in F$ and for each e' occurring as a subterm of some consequent e in $\pi(f)$, if $\mathcal{J}_{\overline{n}}(e')$ is undefined, we let $(\overline{n}, \dagger, e') \in T^{\mathcal{I}}_{e'}$.

Now, for each $(n_1, \ldots, n_m, n) \in Sem_f$ for some $f \in F$, we let $((n_1, \ldots, n_m, n, f), n_k) \in \mathsf{inputValue}^{\mathcal{I}}_k$ for all $1 \leq k \leq m$, and we let $((n_1, \ldots, n_m, n, f), n) \in \mathsf{outputValue}^{\mathcal{I}}$. For each $(n_1, \ldots, n_m, \dagger, f) \in \Delta^{\mathcal{I}}$, we let $((n_1, \ldots, n_m, \dagger, f), n_k) \in \mathsf{inputValue}^{\mathcal{I}}_k$ for all $1 \leq k \leq m$. Then, for each $(n_1, \ldots, n_m, n', e') \in T^{\mathcal{I}}_{e'}$ we let $((n_1, \ldots, n_m, n', e'), n_k) \in \mathsf{hasInput}^{\mathcal{I}}_k$ for all $1 \leq k \leq m$, and we let $((n_1, \ldots, n_m, n', e'), n') \in \mathsf{hasValue}^{\mathcal{I}}$. Similarly, for each $(n_1, \ldots, n_m, \dagger, e') \in T^{\mathcal{I}}_{e'}$, we let $((n_1, \ldots, n_m, \dagger, e'), n_k) \in \mathsf{hasInput}^{\mathcal{I}}_k$ for all $1 \leq k \leq m$. Also, for each $(\overline{n}, x, e') \in T^{\mathcal{I}}_{e'}$, for $x \in \mathbb{N} \cup \{\dagger\}$, and where e' is of the form $g(t_1, \ldots, t_n)$, we let $((\overline{n}, x, e'), (\overline{n}, y, t_k)) \in \mathsf{hasArg}_k^{\mathcal{I}}$ for all $1 \leq k \leq n$, where $y = \mathcal{J}_{\overline{n}}$ if $\mathcal{J}_{\overline{n}}$ is defined and $y = \dagger$ otherwise; this includes g being $+$ or $-$. For each $(\overline{n}, n, f) \in (S_e^f)^{\mathcal{I}}$, we let $((\overline{n}, n, f), (\overline{n}, n', e)) \in \mathsf{hasArg}^{\mathcal{I}}$, where $n' = \mathcal{J}_{\overline{n}}(e)$. Finally, for each $(\overline{n}, \dagger, f) \in (S_e^f)^{\mathcal{I}}$, we let $((\overline{n}, \dagger, f), (\overline{n}, \dagger, e)) \in \mathsf{hasArg}^{\mathcal{I}}$.

With the definition of $\mathcal{I}$ in place, it is now straightforward to verify that $\mathcal{I}$ satisfies Axioms (4.30)-(4.44). The witness for the required object d with suitable properties is then $(n_1, \ldots, n_k, n, f) \in \Delta^{\mathcal{I}}$. □

Corollary 4.5.5 (Canonical countermodel)**.** *For any program π, any $f \in F$ with arity k, any $(n_1, \ldots, n_k, n) \in \mathbb{N}^{k+1}$ such that $(n_1, \ldots, n_k, n) \notin Sem_f$, there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{\pi}$, such that there exists a $d \in \Delta^{\mathcal{I}}$ with $d \in (E^f)^{\mathcal{I}}$, $(d, n_i) \in \mathsf{inputValue}_i^{\mathcal{I}}$ for all $1 \leq i \leq k$, and $(d, n) \notin \mathsf{outputValue}^{\mathcal{I}}$.*

Proof. We consider the canonical model $\mathcal{I}$ for $\mathcal{T}^{\pi}$ as constructed in the proof of Theorem 4.5.4. If $(n_1, \ldots, n_k, n') \in Sem_f$ for some $n' \neq n$, we know $(n_1, \ldots, n_k, n', f) \in \Delta^{\mathcal{I}}$ is a suitable witness. If $(n_1, \ldots, n_k, n') \notin Sem_f$ for all $n' \in \mathbb{N}$, we know $(n_1, \ldots, n_k, \dagger, f) \in \Delta^{\mathcal{I}}$ is a suitable witness. □

The above results can intuitively be explained as follows. Lemma 4.5.1 ensures that conditions are correctly modelled, and Lemma 4.5.2 ensures that the selection of the correct case in definitions according to the conditions is modelled correctly. Theorem 4.5.3 then tells us that all models of the encoding (that satisfy some requirements) reflect the behavior of programs. Furthermore, Theorem 4.5.4 and Corollary 4.5.5 tell us that this is not a vacuous statement, and that there are in fact models witnessing the behavior of programs, and countermodels witnessing non-behavior of programs.

4.5.3 Encoding Reasoning Problems

Theorems 4.5.3 and 4.5.4 and Corollary 4.5.5 allow us to use the encoding of *FPN* programs into $\mathcal{ALC}(\mathcal{D})$ to reduce various reasoning problems over *FPN* programs to reasoning problems over $\mathcal{ALC}(\mathcal{D})$. We show how the equivalence problem of *FPN* programs can be encoded as an $\mathcal{ALC}(\mathcal{D})$ (un)satisfiability problem. Let π_1 and π_2 be two *FPN* programs with disjoint function symbols, and assume that they have function symbol f_1 and f_2, respectively, of the same arity k. The equivalence problem of f_1 and f_2 can be encoded as the unsatisfiability problem of the concept $C_{witness}$ with respect to the TBox

$$\begin{aligned}
\mathcal{T}_{eq,f_1,f_2}^{\pi_1,\pi_2} = \quad & \mathcal{T}^{\pi_1} \cup \mathcal{T}^{\pi_2} \cup \{C_{witness} \sqsubseteq \\
& \exists \mathsf{ref}_1.E_{f_1} \sqcap \exists \mathsf{ref}_2.E_{f_2} \sqcap \\
& \exists (\mathsf{ref}_1\ \mathsf{inputValue}_1)(\mathsf{ref}_2\ \mathsf{inputValue}_1).= \sqcap \\
& \ldots \sqcap \\
& \exists (\mathsf{ref}_1\ \mathsf{inputValue}_k)(\mathsf{ref}_2\ \mathsf{inputValue}_k).= \sqcap \\
& \exists (\mathsf{ref}_1\ \mathsf{outputValue})(\mathsf{ref}_2\ \mathsf{outputValue}).\neq \}
\end{aligned}$$

where ref_1 and ref_2 are abstract features. Any model of $\mathcal{T}_{eq,f_1,f_2}^{\pi_1,\pi_2}$ that satisfies $C_{witness}$ corresponds to a witness of the non-equivalence of the programs p_1 and p_2 by pointing out an example of a tuple that distinguishes the semantics of the two programs.

An example of another reasoning problem on *FPN* programs is the problem of finding a tuple in the semantics of a function symbol given certain constraints on the values in this tuple. We illustrate encoding such reasoning problems in $\mathcal{ALC}(\mathcal{D})$ by giving an encoding for one variant of this general problem. Given a *FPN* program π, a function symbol f occurring in π, and a partial specification P of a candidate instance of f (in the form of a set P_{constr} of constraints $P_i \doteq n_i$, denoting that the ith value of the instance has value n_i), we can reduce the problem of deciding whether there exists an instance satisfying P in the semantic relation for f induced by π to the satisfiability of the concept P with respect to the TBox

$$\mathcal{T}_\pi^P = \mathcal{T}^\pi \cup \{P \sqsubseteq E_f\} \cup \{P \sqsubseteq \exists \mathsf{value}_i.{=_{n_i}} \mid P_i \doteq n_i \in P_{constr}\}$$

where value_i denotes $\mathsf{inputValue}_i$ for $1 \leq i \leq ar(f)$ and denotes $\mathsf{outputValue}$ for $i = ar(f)+1$.

4.6 Encoding *LPN* Programs into $\mathcal{ALC}(\mathcal{D})$

We now show how to model the behavior of programs of the language *LPN* using description logic. We let definitions of functional symbols r in programs be represented by concepts T^r and F^r. Tuples of natural numbers in the relational semantics of programs will be represented by objects in the interpretation of concepts T^r, and tuples that are not in the relational semantics will be represented by objects in the interpretation of concepts F^r. We use (indexed) concepts I^r (resp. J^r) to represent tuples that are (resp. are not) in the input function ι. We use concepts C_s^r (resp. D_s^r) to represent rules s witnessing that tuples are (resp. are not) in the relational semantics for the symbol r. We use (indexed) concepts Q and R to represent constraints occurring in rules. We use concepts N_t to represent complex terms t and their resulting values. Finally, we use (indexed) abstract features hasArg to refer to other tuples that are (or are not) in the relational semantics. Also, we use a number of concrete features to express the required relations between different concrete values.

4.6.1 Constructing a TBox

Given a program p over the relational symbols R and an input function ι for R, we define a TBox $\mathcal{T}^{p,\iota}$ as follows. For each $r \in R$, and each rule s with head relation symbol r, we define concepts C_s^r and D_s^r. Furthermore, for each $r \in R$ and each $\overline{n} \in \iota(r)$, we define concepts $I_{\overline{n}}^r$ and $J_{\overline{n}}^r$. Also, for each $r \in R$, we define two concepts T^r and F^r. For each subterm t' of a term t occurring in any rule in p, we introduce a concept $N_{t'}$. Now, we require:

$$T^r \sqsubseteq \neg \mathsf{hasValue}_1\uparrow \sqcap \cdots \sqcap \neg \mathsf{hasValue}_{ar(r)}\uparrow \tag{4.45}$$

$$T^r \sqsubseteq \neg \mathsf{hasIndex}\uparrow \tag{4.46}$$

$$T^r \sqsubseteq \bigsqcup_s C_s^r \sqcup \bigsqcup_{\overline{n} \in \iota(r)} I_{\overline{n}}^r \tag{4.47}$$

$$F^r \sqsubseteq \neg \mathsf{hasValue}_1\uparrow \sqcap \cdots \sqcap \neg \mathsf{hasValue}_{ar(r)}\uparrow \tag{4.48}$$

$$F^r \sqsubseteq \neg \left(\bigsqcup_s \neg D_s^r \sqcup \bigsqcup_{\overline{n} \in \iota(r)} \neg J_{\overline{n}}^r \right) \tag{4.49}$$

Also, for each $r \in R$, each $\overline{n} = (n_1, \dots, n_k) \in \iota(r)$, we require:

$$I^r_{\overline{n}} \sqsubseteq \exists(\mathsf{hasValue}_1).{=_{n_1}} \sqcap \cdots \sqcap \exists(\mathsf{hasValue}_k).{=_{n_k}} \tag{4.50}$$
$$J^r_{\overline{n}} \sqsubseteq \exists(\mathsf{hasValue}_1).{\neq_{n_1}} \sqcup \cdots \sqcup \exists(\mathsf{hasValue}_k).{\neq_{n_k}} \tag{4.51}$$

Now, for each rule $s = r(x_1, \dots, x_l) \leftarrow r_1(\overline{t}^1), \dots, r_k(\overline{t}^k), c_1, \dots, c_b$ in p, we require, for each $1 \leq j \leq k$, each $1 \leq m \leq ar(r_j)$, for each $1 \leq i \leq l$, and for each $1 \leq v \leq b$, for $c_v = t^1_v\ \rho_v\ t^2_v$, for each $e \in \{1, 2\}$, and for E ranging over $\{C, D\}$:

$$\begin{aligned} C^r_s \sqsubseteq\ & \exists\mathsf{hasArg}^s_j.T^{r_j} \sqcap \\ & \exists(\mathsf{hasArg}^s_j\ \mathsf{hasIndex})(\mathsf{hasIndex}).{<} \end{aligned} \tag{4.52}$$
$$E^r_s \sqsubseteq \exists\mathsf{hasNumber}^j_m.N_{t^j_m} \tag{4.53}$$
$$E^r_s \sqsubseteq \exists(\mathsf{hasNumber}^j_m\ \mathsf{hasInput}_i)(\mathsf{hasValue}_i).{=} \tag{4.54}$$
$$E^r_s \sqsubseteq \exists(\mathsf{hasArg}^s_j\ \mathsf{hasValue}_m)(\mathsf{hasNumber}^j_m\ \mathsf{hasOutput}).{=} \tag{4.55}$$
$$C^r_s \sqsubseteq \exists\mathsf{hasConstr}^s_v.Q^{\rho_v} \tag{4.56}$$
$$E^r_s \sqsubseteq \exists\mathsf{hasConstrNumber}^{s,e}_v.N_{t^e_v} \tag{4.57}$$
$$E^r_s \sqsubseteq \exists(\mathsf{hasConstrNumber}^{s,e}_v\ \mathsf{hasInput}_i)(\mathsf{hasValue}_i).{=} \tag{4.58}$$
$$E^r_s \sqsubseteq \exists(\mathsf{hasConstr}^s_v\ \mathsf{hasValue}_e)(\mathsf{hasConstrNumber}^{s,e}_v\ \mathsf{hasOutput}).{=} \tag{4.59}$$
$$\begin{aligned} D^r_s \sqsubseteq\ & \exists\mathsf{hasArg}^s_1.F^{r_1} \sqcup \dots\ \sqcup \exists\mathsf{hasArg}^s_k.F^{r_k} \sqcup \\ & \exists\mathsf{hasConstr}^s_1.R^{\rho_1} \sqcup \cdots \sqcup \exists\mathsf{hasConstr}^s_b.R^{\rho_b} \end{aligned} \tag{4.60}$$

Then, for each constraint $c_v = t^1_v\ \rho_v\ t^2_v$ for $1 \leq v \leq b$, we require

$$Q^{\rho_v} \sqsubseteq \exists(\mathsf{hasValue}_1)(\mathsf{hasValue}_2).\rho_v \tag{4.61}$$
$$R^{\rho_v} \sqsubseteq \exists(\mathsf{hasValue}_1)(\mathsf{hasValue}_2).\overline{\rho_v} \tag{4.62}$$

Then, for each concept N_t we require the following, where we distinguish different cases for different forms of t, where ρ ranges over $\{+, -\}$, $j \in \{1, 2\}$, and $1 \leq i \leq \max\{i \mid x_i \text{ occurs in } t\}$:

$$N_n \sqsubseteq \exists(\mathsf{hasOutput}).{=_n} \tag{4.63}$$
$$N_{x_i} \sqsubseteq \exists(\mathsf{hasOutput})(\mathsf{hasInput}_i).{=} \tag{4.64}$$
$$N_{t_1\ \rho\ t_2} \sqsubseteq \exists\mathsf{hasNumber}_1.N_{t_1} \sqcap \exists\mathsf{hasNumber}_2.N_{t_2} \tag{4.65}$$
$$N_{t_1\ \rho\ t_2} \sqsubseteq \exists(\mathsf{hasNumber}_j\ \mathsf{hasInput}_i)(\mathsf{hasInput}_i).{=} \tag{4.66}$$
$$\begin{aligned} N_{t_1 - t_2} \sqsubseteq\ & (\neg\exists(\mathsf{hasNumber}_1\ \mathsf{hasOutput})(\mathsf{hasNumber}_2\ \mathsf{hasOutput}).{\geq} \\ & \sqcup \exists(\mathsf{hasOutput}).{=_0}) \sqcap \\ & (\neg\exists(\mathsf{hasNumber}_1\ \mathsf{hasOutput})(\mathsf{hasNumber}_2\ \mathsf{hasOutput}).{<} \\ & \sqcup \exists(\mathsf{hasOutput})(\mathsf{hasNumber}_1\ \mathsf{hasOutput})(\mathsf{hasNumber}_2\ \mathsf{hasOutput}).{-}) \end{aligned} \tag{4.67}$$
$$N_{t_1 + t_2} \sqsubseteq \exists(\mathsf{hasOutput})(\mathsf{hasNumber}_1\ \mathsf{hasOutput})(\mathsf{hasNumber}_2\ \mathsf{hasOutput}).{+} \tag{4.68}$$

Note that, in general, the TBox $\mathcal{T}^{p,\iota}$ is not acyclic. There are two possible combinations of axioms that can induce cycles: on the one hand, combinations of Axioms (4.47) and (4.52), and on the other hand, combinations of Axioms (4.49) and (4.60) can induce cycles.

Intuitively, these axioms serve the following purpose. We simultaneously model the behavior and non-behavior of programs. Axioms (4.45)-(4.49) encode some general properties of the modelling. Axioms (4.50) and (4.51) encode the behavior of the input function. Axioms (4.52)-(4.60) encode the behavior of the right hand side of rules, where Axioms (4.61) and (4.62) encode the behavior of constraints occuring in the right hand side of rules. Finally, Axioms (4.63)-(4.68) compute the value of expressions occurring in rules.

4.6.2 Semantic Correspondence

In order to use the above encoding of a program p with input ι into an $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^{p,\iota}$, we show the following correspondence between the semantics of p w.r.t. ι and the model theoretic semantics of $\mathcal{T}^{p,\iota}$.

Lemma 4.6.1 (Value computation)**.** *For any program p, any input function ι, any model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{p,\iota}$, any (sub)term t occurring in any rule of p (with l variables in its head), and for any $(n_1, \dots, n_l) \in \mathbb{N}^l$, if $d \in N_t^{\mathcal{I}}$ and $(d, n_i) \in \mathsf{hasInput}_i^{\mathcal{I}}$ for all $1 \leq i \leq l$, then $(d, o) \in \mathsf{hasOutput}^{\mathcal{I}}$ for $o = \mathcal{J}_\sigma(t)$ where $\sigma(x_i) = n_i$ for all $1 \leq i \leq l$.*

Proof. We prove this by induction on the structure of t. In the base case for $t \in \mathbb{N}$, this follows directly from the definition of $\mathcal{J}_\sigma(t)$ and from the fact that Axiom (4.63) holds. Similarly, in the base case for $t = x_i$, the result follows directly from the definition and the fact that Axiom (4.64) holds.

In the inductive case, for $t = t_1 \ \rho \ t_2$ where $\rho \in \{+, -\}$, we know by Axioms (4.65) and (4.66) that there exist $d_1 \in N_{t_1}^{\mathcal{I}}$ and $d_2 \in N_{t_2}^{\mathcal{I}}$ with $(d_j, n_i) \in \mathsf{hasInput}_i^{\mathcal{I}}$ for both $j \in \{1, 2\}$ and all $1 \leq i \leq l$. By the induction hypothesis, then, we know that $(d_j, o_j) \in \mathsf{hasOutput}^{\mathcal{I}}$ for $j \in \{1, 2\}$ and $o_j = \mathcal{J}_\sigma(t_j)$. The result now follows directly from the definition of $\mathcal{J}_\sigma(t)$ and from the fact that Axioms (4.67) and (4.68) hold. □

Theorem 4.6.2 (Positive semantic enforcement)**.** *For any program p, any input function ι, any model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{p,\iota}$, any $r \in R$ of arity k, and for any $n_1, \dots, n_k \in \mathbb{N}$, if there is a $d \in (T^r)^{\mathcal{I}}$ with $(d, n_i) \in \mathsf{hasValue}_i^{\mathcal{I}}$ for each $1 \leq i \leq k$, then $(n_1, \dots, n_k) \in Sem_r$.*

Proof. Take an arbitrary suitable model $\mathcal{I}$. We show for all suitable elements $d \in (T^r)^{\mathcal{I}}$ with $(d, j) \in \mathsf{hasIndex}^{\mathcal{I}}$ that $\overline{n} \in Sem_r^j$, by induction on j. This suffices, since we know that each suitable d can be associated with such a j, by Axiom (4.46), and that $Sem_r^j \subseteq Sem_r$.

In the base case, by Axioms (4.47) and (4.52), we know that $d \notin (C_s^r)^{\mathcal{I}}$ for any rule s with at least one predicate symbol in the body, and thus by Axiom (4.47), either (i) $d \in (C_s^r)^{\mathcal{I}}$ for a rule s with 0 predicate symbols in the body, or (ii) $d \in (I_{\overline{m}}^r)^{\mathcal{I}}$ for some $\overline{m} \in \iota(r)$.

In case (i), by Axioms (4.56)-(4.59), and by Lemma 4.6.1, we know that for each constraint $t_v^1 \ \rho_v \ t_v^2$ in the body of s, there exist an element $d_v \in (Q^{\rho_v})^{\mathcal{I}}$ such that $(d, d_v) \in (\mathsf{hasConstr}_v^s)^{\mathcal{I}}$, $(d_v, \mathcal{J}_\sigma(t_v^1)) \in \mathsf{hasValue}_1^{\mathcal{I}}$ and $(d_v, \mathcal{J}_\sigma(t_v^2)) \in \mathsf{hasValue}_2^{\mathcal{I}}$ with $\sigma(x_i) = n_i$ for all $1 \leq i \leq k$. Then, by Axiom (4.61), we know that the instantiation σ satisfies all constraints $t_v^1 \ \rho_v \ t_v^2$, and thus by definition of Sem_r^j, we have that $(n_1, \dots, n_k) \in Sem_r^j$.

In case (ii), by Axiom (4.50) we know that $(n_1, \dots, n_k) = \overline{m} \in \iota(r)$ and thus by definition of Sem_r^j, we have that $(n_1, \dots, n_k) \in Sem_r^j$.

In the inductive case, by Axiom (4.47), we know that either (iii) $d \in (C^r_s)^{\mathcal{I}}$ for a rule s, or (iv) $d \in (I^r_{\overline{m}})^{\mathcal{I}}$ for some $\overline{m} \in \iota(r)$.

In case (iii), similarly to the base case, by Axioms (4.56)-(4.59) and (4.61), and by Lemma 4.6.1, we know that σ satisfies all constraints in the body of s, where $\sigma(x_i) = n_i$ for all $1 \leq i \leq k$. It thus suffices to show that $(\mathcal{J}_\sigma(t^m_1), \dots, \mathcal{J}_\sigma(t^m_{ar(r_m)}) \in Sem^{j'}_r$ for $j' < j$, for all atoms $r_m(t^m_1, \dots, t^m_{ar(r_m)})$ in the body of rule s. Take an arbitrary such atom $r_m(\overline{t}^m)$. By Axioms (4.52)-(4.55) and by Lemma 4.6.1, we know that there exists a $d' \in (T^{r_m})^{\mathcal{I}}$ such that $(d, d') \in (\mathsf{hasArg}^s_m)^{\mathcal{I}}$, $(d', \mathcal{J}_\sigma(t^m_u)) \in \mathsf{hasValue}^{\mathcal{I}}_u$ for all $1 \leq u \leq ar(r_m)$, and $(d', j') \in \mathsf{hasIndex}^{\mathcal{I}}$ for some $j' < j$. The result now follows directly from the induction hypothesis.

The argumentation in case (iv) is exactly the same as in case (ii). □

Theorem 4.6.3 (Negative semantic enforcement). *For any program p, any input function ι, any model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{p,\iota}$, any $r \in R$ of arity k, and for any $n_1, \dots, n_k \in \mathbb{N}$, if there is a $d \in (F^r)^{\mathcal{I}}$ with $(d, n_i) \in \mathsf{hasValue}^{\mathcal{I}}_i$ for each $1 \leq i \leq k$, then $(n_1, \dots, n_k) \notin Sem_r$.*

Proof. We prove the contrapositive, for all $(n_1, \dots, n_k) \in Sem^j_r$, by induction on j. Without loss of generality, we consider only the inductive case. Since $(n_1, \dots, n_k) \in Sem^j_r$, we know that either (i) $(n_1, \dots, n_k) \in \iota(r)$ or (ii) some rule $r(\overline{x}) \leftarrow r_1(\overline{t}^1), \dots, r_m(\overline{t}^m), c_1, \dots, c_b$ witnesses that $(n_1, \dots, n_k) \in Sem^j_r$: for σ where for all $1 \leq i \leq k$ we have $\sigma(x_i) = n_i$, for each $1 \leq l \leq m$ we get $(\mathcal{J}_\sigma(t^l_1), \dots, \mathcal{J}_\sigma(t^l_{ar(r_l)})) \in Sem^{j'}_{r_l}$ for some $j' < j$, and all constraints $\sigma(c_v)$ for $1 \leq v \leq b$ hold.

Assume there exists a suitable $d \in (F^r)^{\mathcal{I}}$ for some model $\mathcal{I}$ of $\mathcal{T}^{p,\iota}$. By Axiom (4.49), we know then that $d \in (J^r_{\overline{n}})^{\mathcal{I}}$ for all $\overline{n} \in \iota(r)$ and $d \in (D^r_s)^{\mathcal{I}}$ for all rules s with predicate symbol r in the head.

In case (i), we know $(n_1, \dots, n_k) \in \iota(r)$. We also know that $(d, n_i) \in \mathsf{hasValue}_i$ for $1 \leq i \leq k$. However, this, with the fact that $d \in (J^r_{(n_1,\dots,n_k)})^{\mathcal{I}}$, and the fact that Axiom (4.51) holds, leads to a direct contradiction. Thus we can conclude $d \notin (F^r)^{\mathcal{I}}$.

In case (ii), we know that some rule $s = r(\overline{x}) \leftarrow r_1(\overline{t}^1), \dots, r_m(\overline{t}^m), c_1, \dots, c_b$ witnesses $(n_1, \dots, n_k) \in Sem^j_r$. Remember that, by Axiom (4.49), we know $d \in (D^r_s)^{\mathcal{I}}$. Also, by Axiom (4.60), we know that either (ii.a) for some $1 \leq i \leq k$ there exists a d_i such that $(d, d_i) \in (\mathsf{hasArg}^s_i)^{\mathcal{I}}$, and $d_i \in (F^{r_i})^{\mathcal{I}}$, or (ii.b) for some $1 \leq v \leq b$ there exists a e_v such that $(d, e_v) \in (\mathsf{hasConstr}^s_v)^{\mathcal{I}}$ and $e_v \in (R^{\rho_v})^{\mathcal{I}}$.

In case (ii.a), by Axioms (4.52)-(4.55), and by Lemma 4.6.1, we know that $(d_i, \mathcal{J}_\sigma(t^i_c)) \in \mathsf{hasValue}^{\mathcal{I}}_c$ for all $1 \leq c \leq ar(r_i)$. Since $(\mathcal{J}_\sigma(t^i_1), \dots, \mathcal{J}_\sigma(t^i_{ar(r_i)})) \in Sem^{j'}_r$ for $j' < j$, we can use the induction hypothesis, and thus we know $d_i \notin (F^{r_i})^{\mathcal{I}}$. This is a direct contradiction with our previous conclusion that $d_i \in (F^{r_i})^{\mathcal{I}}$, and thus we can conclude that $d \notin (F^r)^{\mathcal{I}}$.

In case (ii.b), by Axioms (4.57)-(4.59), and by Lemma 4.6.1, we know that for $c_v = t^1_v \; \rho_v \; t^2_v$ we have $(e_v, \mathcal{J}_\sigma(t^1_v)) \in \mathsf{hasNumber}^{\mathcal{I}}_1$ and $(e_v, \mathcal{J}_\sigma(t^2_v)) \in \mathsf{hasNumber}^{\mathcal{I}}_2$. By the fact that $e_v \in (R^{\rho_v})^{\mathcal{I}}$ and by Axiom (4.62), then, we know that the constraint $\sigma(c_v)$ is not satisfied, which is a direct contradiction with our previous assessment, that $\sigma(c_v)$ is satisfied. Thus we can conclude that $d \notin (F^r)^{\mathcal{I}}$. □

In the following theorem a canonical model is constructed on the basis of the relational semantics of programs.

Theorem 4.6.4 (Canonical model). *For any program p, any input function ι, any $r \in R$ of arity k, and for any $n_1, \dots, n_k \in \mathbb{N}$, if $(n_1, \dots, n_k) \in Sem_r$ then there exists a model $\mathcal{I}$ of $\mathcal{T}^{p,\iota}$ with some $d \in \Delta^{\mathcal{I}}$ such that $d \in (T^r)^{\mathcal{I}}$ and $(d, n_i) \in \mathsf{hasValue}_i^{\mathcal{I}}$ for each $1 \leq i \leq k$, and if $(n_1, \dots, n_k) \notin Sem_r$ then there exists a model $\mathcal{I}$ of $\mathcal{T}^{p,\iota}$ with some $d \in \Delta^{\mathcal{I}}$ such that $d \in (F^r)^{\mathcal{I}}$ and $(d, n_i) \in \mathsf{hasValue}_i^{\mathcal{I}}$ for each $1 \leq i \leq k$.*

Proof. We construct a canonical model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^{p,\iota}$, which is a suitable model. We let $\Delta^{\mathcal{I}} = \bigcup_{r \in R}(\mathbb{N}^{ar(r)} \cup \mathbb{N}^{ar(r)+1} \cup (\mathbb{N}^{ar(r)} \times (\{c \mid \text{constraint } c \text{ in a rule for } r\})))$, and we define $\cdot^{\mathcal{I}}$ as follows. For each r, we let $(T^r)^{\mathcal{I}} = Sem_r$ and we let $(F^r)^{\mathcal{I}} = \mathbb{N}^{ar(r)} \backslash Sem_r$. Also, for each $\overline{n} = (n_1, \dots, n_k) \in \Delta^{\mathcal{I}}$ we let $(\overline{n}, n_i) \in \mathsf{hasValue}_i^{\mathcal{I}}$ for $1 \leq i \leq k$. For each r and each $\overline{n} \in (T^r)^{\mathcal{I}}$, we let $(\overline{n}, j) \in \mathsf{hasIndex}^{\mathcal{I}}$ for the smallest j such that $\overline{n} \in Sem_r^j$. For each r and each $\overline{n} \in Sem_r$, we know there exists a justification for $\overline{n} \in Sem_r$, in the form of either (i) $\overline{n} \in iota(r)$, or (ii) a rule $s = r(\overline{x}) \leftarrow r_1(\overline{t}^1), \dots, r_m(\overline{t}^m), c_1, \dots, c_b$ with suitable $\overline{n}^i \in Sem_{r_i}$ for $1 \leq i \leq m$. Without loss of generality, we can assume that there is exactly one such justification. In case (i), we let $\overline{n} \in (I^r_{\overline{n}})^{\mathcal{I}}$.

In case (ii), we let $\overline{n} \in (C^r_s)^{\mathcal{I}}$ and $(\overline{n}, \overline{n}^i) \in (\mathsf{hasArg}_i^s)^{\mathcal{I}}$ for $1 \leq i \leq m$. For each term t_i^l occurring as ith term in predicate r^l in rule s, we let $(\overline{n}, (\overline{n}, \mathcal{J}_\sigma(t_i^l))) \in (\mathsf{hasNumber}_i^l)^{\mathcal{I}}$, for σ defined by $\sigma(x_i) = n_i$ for $1 \leq i \leq k$. Also, for each constraint $c_v = t_v^1 \ \rho_v \ t_v^2$ in s, we let $(\overline{n}, c_v) \in (Q^{\rho_v})^{\mathcal{I}}$, $(\overline{n}, (\overline{n}, c_v)) \in (\mathsf{hasConstr}_v^s)^{\mathcal{I}}$, and $((\overline{n}, c_v), \mathcal{J}_\sigma(t_v^j)) \in \mathsf{hasValue}_j^{\mathcal{I}}$ for $j \in \{1, 2\}$ and σ defined by $\sigma(x_i) = n_i$ for $1 \leq i \leq k$. For each term t_v^j occurring as jth term in constraint c_v, we let $(\overline{n}, (\overline{n}, \mathcal{J}_\sigma(t_v^j))) \in (\mathsf{hasConstrNumber}_v^{s,j})^{\mathcal{I}}$.

For each (sub)term t occurring in a rule with head predicate r, and each $\overline{n} = (n_1, \dots, n_k) \in \mathbb{N}^{ar(r)}$, we let $(\overline{n}, \mathcal{J}_\sigma(t)) \in (N_t)^{\mathcal{I}}$, $((\overline{n}, \mathcal{J}_\sigma(t)), n_i) \in \mathsf{hasInput}_i^{\mathcal{I}}$ for $1 \leq i \leq k$, and $((\overline{n}, \mathcal{J}_\sigma(t)), \mathcal{J}_\sigma(t)) \in \mathsf{hasOutput}^{\mathcal{I}}$, for σ defined by $\sigma(x_i) = n_i$ for $1 \leq i \leq k$. Also, for such t of the form $t_1 \cdot t_2$ for some operator $\cdot$, for each $\overline{n} = (n_1, \dots, n_k) \in \mathbb{N}^{ar(r)}$, we let $((\overline{n}, \mathcal{J}_\sigma(t)), (\overline{n}, \mathcal{J}_\sigma(t_j))) \in \mathsf{hasNumber}_j^{\mathcal{I}}$ for $j \in \{1, 2\}$.

For each r and each $\overline{n} = (n_1, \dots, n_k) \in (F^r)^{\mathcal{I}}$, we let $\overline{n} \in (J^r_{\overline{m}})^{\mathcal{I}}$ for all $\overline{m} \in \iota(r)$, and $\overline{n} \in (D^r_s)^{\mathcal{I}}$ for all rules s for r. Further, for each rule s and each atom $r_l(t_1^l, \dots, t_{ar(r_l)}^l)$ occurring in s, we let $(\overline{n}, (\mathcal{J}_\sigma(t_1^l), \dots, \mathcal{J}_\sigma(t_{ar(r_l)}^l))) \in (\mathsf{hasArg}_l^s)^{\mathcal{I}}$, and for each constraint $c_v = t_v^1 \ \rho_v \ t_v^2$ in s, we let $(\overline{n}, c_v) \in (Q^{\rho_v})^{\mathcal{I}}$, $(\overline{n}, (\overline{n}, c_v)) \in (\mathsf{hasConstr}_v^s)^{\mathcal{I}}$, and $((\overline{n}, c_v), \mathcal{J}_\sigma(t_v^j)) \in \mathsf{hasValue}_j^{\mathcal{I}}$ for $j \in \{1, 2\}$ and σ defined by $\sigma(x_i) = n_i$ for $1 \leq i \leq k$. For each term t_v^j occurring as jth term in constraint c_v, we let $(\overline{n}, (\overline{n}, \mathcal{J}_\sigma(t_v^j))) \in (\mathsf{hasConstrNumber}_v^{s,j})^{\mathcal{I}}$.

Finally, for each $(\overline{n}, c) \in \Delta^{\mathcal{I}}$ for a constraint $c = t^1 \ \rho \ t^2$, such that $\sigma(c)$ is not satisfied, where $\overline{n} = (n_1, \dots, n_k)$ σ is defined by $\sigma(x_i) = n_i$ for $1 \leq i \leq k$, we let $(\overline{n}, c) \in (R^\rho)^{\mathcal{I}}$.

It is now straightforward to verify that $\mathcal{I}$ satisfies all axioms. Also, it is easy to see that $\mathcal{I}$ is a suitable model witnessing the result. □

The above results can intuitively be explained as follows. Lemma 4.6.1 shows us that the values of expressions are correctly computed in the modelling. Theorem 4.6.2 then shows us that all models of the encoding (that satisfy some requirements) reflect the behavior of programs. Similarly, Theorem 4.6.3 shows us that all models of the encoding (that satisfy some requirements) reflect the non-behavior of programs. Finally, Theorem 4.6.4 shows us that these are not vacuous statements, and that there are in fact models witnessing the behavior and non-behavior of programs.

4.6.3 Encoding Reasoning Problems

Theorems 4.6.2, 4.6.3 and 4.6.4 allow us to use the encoding of *LPN* programs into $\mathcal{ALC}(\mathcal{D})$ to reduce various reasoning problems over *LPN* programs to reasoning problems over $\mathcal{ALC}(\mathcal{D})$. We show how the equivalence problem of *LPN* programs can be encoded as an $\mathcal{ALC}(\mathcal{D})$ (un)-satisfiability problem.

Let p_1 and p_2 be two *LPN* programs with disjoint predicate symbols, with input functions ι_1 and ι_2, respectively, and assume that they have predicate symbol r_1 and r_2, respectively, of the same arity k. The equivalence problem of r_1 and r_2 can be encoded as the unsatisfiability problem of the concept $C_{witness}$ with respect to the TBox

$$\begin{aligned}\mathcal{T}^{p_1,\iota_1,p_2,\iota_2}_{eq,r_1,r_2} = \quad & \mathcal{T}^{p_1,\iota_1} \cup \mathcal{T}^{p_2,\iota_2} \cup \{C_{witness} \sqsubseteq \\ & ((\exists \mathsf{ref}_1.F^{r_1} \sqcap \exists \mathsf{ref}_2.T^{r_2}) \sqcup (\exists \mathsf{ref}_1.T^{r_1} \sqcap \exists \mathsf{ref}_2.F^{r_2})) \sqcap \\ & \exists(\mathsf{ref}_1\ \mathsf{hasValue}_1)(\mathsf{ref}_2\ \mathsf{hasValue}_1).= \sqcap \\ & \ldots \sqcap \\ & \exists(\mathsf{ref}_1\ \mathsf{hasValue}_k)(\mathsf{ref}_2\ \mathsf{hasValue}_k).= \}\end{aligned}$$

where ref_1 and ref_2 are abstract features. Any model of $\mathcal{T}^{p_1,\iota_1,p_2,\iota_2}_{eq,r_1,r_2}$ that satisfies $C_{witness}$ corresponds to a witness of the non-equivalence of the programs p_1 (with ι_1) and p_2 (with ι_2) by pointing out an example of a tuple that distinguishes the semantics of the two programs.

An example of another reasoning problem on *LPN* programs is the problem of finding a tuple in the semantics of a function symbol given certain constraints on the values in this tuple. We illustrate encoding such reasoning problems in $\mathcal{ALC}(\mathcal{D})$ by giving an encoding for one variant of this general problem. Given a *LPN* program p with input function ι, a predicate symbol r occurring in p, and a partial specification P of a candidate instance of r (in the form of a set P_{constr} of constraints $P_i \doteq n_i$, denoting that the ith value of the instance has value n_i), we can reduce the problem of deciding whether there exists an instance satisfying P in the semantic relation for r induced by p and ι to the satisfiability of the concept P with respect to the TBox

$$\mathcal{T}^P_{p,\iota} = \mathcal{T}^{p,\iota} \cup \{P \sqsubseteq T^r\} \cup \{P \sqsubseteq \exists \mathsf{hasValue}_i.=_{n_i} \mid P_i \doteq n_i \in P_{constr}\}$$

Note that executing *LPN* programs can be seen as a particular instance of the above problem of finding instances satisfying a given partial specification.

4.7 Encoding Reasoning Problems on Different Languages

In Sections 4.3.3, 4.4.3, 4.5.3 and 4.6.3, we described how reasoning over programs of any single one of the programming languages *While*, *Goto*, *FPN* and *LPN* can be encoded as reasoning problems of the description logic $\mathcal{ALC}(\mathcal{D})$. However, it is also possible to reason simultaneously over programs of different programming languages. We illustrate how this can be done by means of a number of examples.

The first example we consider illustrates how to reason on programs of different imperative programming languages simultaneously. In Section 3.4.2, we defined what it means for a *While* program p and a *Goto* program κ to be equivalent. We show how to encode this property as an $\mathcal{ALCO}(\mathcal{D})$ (un)satisfiability problem. Let $\mathcal{T}^p$ be the TBox encoding of p, as defined in Section 4.3, and let $\mathcal{T}^\kappa$ be the TBox encoding of κ, as defined in Section 4.4. Consider the ABox

$$\mathcal{A}^{p,\kappa} = \{o : C_p, o : C_{\kappa,1}, s : C_{test}\}$$

with respect to the TBox $\mathcal{S}^p \cup \mathcal{S}^\kappa \cup \mathcal{T}^{eq}$ where $\mathcal{S}^p$ consists of $\mathcal{T}^p$ with $\mathsf{nextState}$ replaced by a new abstract feature $\mathsf{nextState}_1$, where $\mathcal{S}^\kappa$ consists of $\mathcal{T}^\kappa$ with $\mathsf{nextState}$ replaced by a new abstract feature $\mathsf{nextState}_2$, and where

$$\begin{aligned}\mathcal{T}^{eq} = \{ \quad C_{test} \equiv \;& \exists \mathsf{res}_1.C_{skip} \sqcap \exists \mathsf{res}_2.C_{return} \sqcap \\ & (\exists(\mathsf{res}_1\ \mathsf{valueOf}_{x_1})(\mathsf{res}_2\ \mathsf{valueOf}_{x_1}).\neq \\ & \sqcup \cdots \sqcup \\ & \exists(\mathsf{res}_1\ \mathsf{valueOf}_{x_n})(\mathsf{res}_2\ \mathsf{valueOf}_{x_n}).\neq) \;\}\end{aligned}$$

for res_1, res_2 abstract features, and C_{skip}, C_{return} and C_{test} nominal concepts. $\mathcal{ALCO}(\mathcal{D})$ models for this ABox and TBox correspond to derivations of p and κ starting with the same input state (reflected in the ABox assertions $o : C_p$ and $o : C_{\kappa,1}$), and resulting in at least one different output value (reflected in the only TBox axiom).

Similarly, we could encode the equivalence of programs of the different declarative languages as an $\mathcal{ALC}(\mathcal{D})$ (un)satisfiability problem. By taking the encoding of an *FPN* program and the encoding of an *LPN* program, in combination with suitable $\mathcal{ALC}$ statements expressing the required relation between concepts of the two encodings, we can express the constraint that there be values $(n_1, \ldots, n_k) \in \mathbb{N}^k$ such that for some $r \in R$ with arity k and some $f \in F$ with arity $k-1$ we have that not both $(n_1, \ldots, n_k) \in Sem^p_r$ and $(n_1, \ldots, n_k) \in Sem^\pi_f$.

The final example we consider relates an imperative and a declarative program. We encode the problem whether a *While* program p is equivalent for inputs $X' = \{x_1, \ldots, x_n\}$ and output x to a *FPN* program π for a functional symbol $f \in F$, as defined in Section 3.4.3. We show how to encode this property as an $\mathcal{ALCO}(\mathcal{D})$ unsatisfiability problem. Let $\mathcal{T}^p$ be the TBox encoding of p, as defined in Section 4.3, and let $\mathcal{T}^\pi$ be the TBox encoding of π, as defined in Section 4.5. Consider the ABox

$$\mathcal{A}^{p,\kappa} = \{o : C_p, o' : C_\pi, s : C_{test}, (s, o') : \mathsf{ref}_1, (s, o) : \mathsf{ref}_2\}$$

with respect to the TBox $\mathcal{T}^p \cup \mathcal{T}^\pi \cup \mathcal{T}^{eq}$ where

$$\begin{aligned}\mathcal{T}^{eq} = \{ \quad C_{test} \equiv \;& \exists \mathsf{ref}_3.C_{skip} \sqcap \\ & (\exists(\mathsf{ref}_1\ \mathsf{inputValue}_1)(\mathsf{ref}_2\ \mathsf{valueOf}_{x_1}).= \\ & \sqcap \cdots \sqcap \\ & \exists(\mathsf{ref}_1\ \mathsf{inputValue}_n)(\mathsf{ref}_2\ \mathsf{valueOf}_{x_n}).= \sqcap \\ & \exists(\mathsf{ref}_1\ \mathsf{outputValue})(\mathsf{ref}_3\ \mathsf{valueOf}_x.\neq)) \;\}\end{aligned}$$

for res_1, res_2, res_3 abstract features, and C_{skip} a nominal concept. $\mathcal{ALCO}(\mathcal{D})$ models for this ABox and TBox correspond to derivations of p starting with some values $(n_1, \ldots, n_k) \in \mathbb{N}^k$ for the variables X' leading to the output value n for x, such that π does not map $(n_1, \ldots, n_k)$ to n for the functional symbol f.

4.8 Computational Power and Decidability of Finding $\mathcal{ALC}(\mathcal{D})$ Models

The relations between executing programs of the different programming languages and the model-theoretic semantics of $\mathcal{ALC}(\mathcal{D})$ knowledge bases that we identified above lead to a number of interesting conclusions with respect to the computational power and decidability of the problem of finding models for $\mathcal{ALC}(\mathcal{D})$ knowledge bases, in the general case.

4.8.1 Turing-Completeness

In Section 4.3, we formally established a connection between executing *While* programs and the model-theoretic semantics of particular $\mathcal{ALC}(\mathcal{N}_{lin})$ knowledge bases. Together with the well-known Turing-completeness results we discussed in Section 3.3, these results lead to the following insight about the computational power of finding models for $\mathcal{ALC}(\mathcal{N}_{lin})$ knowledge bases.

Theorem 4.8.1. *Any mechanism that, given a general $\mathcal{ALC}(\mathcal{N}_{lin})$ TBox $\mathcal{T}$ and given an $\mathcal{ALC}$-($\mathcal{N}_{lin}$) concept C returns a model $\mathcal{I}$ for $\mathcal{T}$ with $C^{\mathcal{I}} \neq \emptyset$, if such a model exists, has Turing-complete computational power.*

Proof (sketch). Let M be such a mechanism. We show that M can be used to compute arbitrary Turing-computable functions $f(x_1, \ldots, x_m)$. Let f be such a function. By Proposition 3.3.2 we know that this function f can be expressed by a *While* program p, i.e. when p is executed on an input state s_{in} that provides values $c_1, \ldots, c_m$ for the variables $x_1, \ldots, x_m$ (respectively) it returns the value of $f(c_1, \ldots, c_m)$ in an output variable z iff p terminates on this input state. By Theorems 4.3.2, 4.3.3 and 4.3.4, we know that we can use M to compute $f(c_1, \ldots, c_m)$. We do so as follows.

We provide M with the TBox $\mathcal{T}^p$ and with the concept $C = C_p \sqcap \exists \mathsf{valueOf}_{x_1}.{=_{c_1}} \sqcap \cdots \sqcap \exists \mathsf{valueOf}_{x_m}.{=_{c_m}}$. We know either p terminates on the input state s_{in}, or it does not terminate on s_{in}. By Theorems 4.3.3 and 4.3.4, we know that either way, a model $\mathcal{I}$ of $\mathcal{T}^p$ with $C^{\mathcal{I}} \neq \emptyset$ exists. Thus, M returns such a model $\mathcal{I}$. If $C^{\mathcal{I}}_{skip} = \emptyset$, we know that p does not terminate on s_{in}, by Theorem 4.3.2, and thus that $f(c_1, \ldots, c_m)$ is undefined. Otherwise (i.e. if $C^{\mathcal{I}}_{skip} \neq \emptyset$), we can use the model $\mathcal{I}$ and any object d witnessing that $C^{\mathcal{I}} \neq \emptyset$, together with the final element e in the $\mathsf{nextState}$ chain in $\mathcal{I}$ starting at d and the unique value $k \in \mathbb{N}$ such that $(e, k) \in \mathsf{valueOf}^{\mathcal{I}}_z$ to obtain $f(c_1, \ldots, c_m)$. □

4.8.2 Decidability

We have seen in Section 4.7 that the encodings described in this chapter offer a flexible framework to express a myriad of different reasoning problems in $\mathcal{ALC}(\mathcal{D})$. The downside of this result is that the resulting reasoning problems, in the general case, are undecidable. Several straightforward formal undecidability proofs witnessing this will be given in Chapter 6. In this section, we will informally describe the possible causes of the undecidability of the reasoning problems resulting from the encoding into description logic. Investigating these possible causes helps to give us a better understanding of the nature of the problems we are trying to solve, as well as guiding our search for additional fragments of the programming language that allow for decidable reasoning problems.

It must be noted that, despite having the practical disadvantage of being undecidable in general, these problems do have a clearly defined formal semantics. In other words, it is well decidable whether a particular candidate solution is in fact a solution for such a problem. This opens up the possibility of using incomplete methods for trying to find solutions to these problems. For instance, it is well possible to use local search algorithms to try to find counterexample models for the $\mathcal{ALC}(\mathcal{D})$ reasoning problems created using the encoding framework.

4.8.2.1 General TBoxes

In [17], it is shown that $\mathcal{ALC}(\mathcal{D})$ concept satisfiability (and thereby related reasoning problems such as concept subsumption and equivalence) with respect to general TBoxes is undecidable for arithmetic concrete domains. A concrete domain is arithmetic if it includes the natural numbers and has predicates for (in)equality, (in)equality to zero, addition and multiplication. The concrete domains on which the considered programming languages operate, without further restrictions, thus are arithmetic concrete domains.

Also, the programming languages we consider in their general form allow a certain looping behavior that leads to a cyclicity in the TBoxes of the encoding of programs. For the imperative programming languages, this cyclicity is induced by the unbounded number of loops that can appear in the execution of programs. For the language *While*, this looping behavior is caused by **while** statements, and the resulting need for general TBoxes can be seen in the fact that the combination of Axiom (4.15) with Axioms (4.7), (4.8) and (4.14) can induce cycles. Similarly, for the language *Goto*, this looping behavior is caused by **goto** statements, and the corresponding need for general TBoxes is mirrored in the fact that Axioms (4.28) and (4.29) can induce cycles.

For the declarative programming languages we considered, an analogous looping behavior is present in the semantics of the languages that is caused by the possibility of programs to be recursive. For the language *FPN*, the need for general TBoxes due to this recursive behavior can be seen in the fact that Axioms (4.31), (4.42) and (4.38)-(4.39) can induce cycles. Similarly, for the language *LPN*, this need for general TBoxes is mirrored both in the fact that Axioms (4.47) and (4.52), and in Axioms (4.49) and (4.60) can induce cycles.

Because of the use of arithmetic domains and the need for general TBoxes, the $\mathcal{ALC}(\mathcal{D})$ reasoning problems that result from the encoding of programs into logic are particular instances of a generally undecidable class of problems. This by itself does not imply that the resulting reasoning problems are in fact undecidable. In principle, it could be the case that we encode decidable problems into a class of problems that are undecidable in general. However, as we will see in Chapter 6, we are in fact dealing with generally undecidable reasoning problems, exactly because of the combination of numerical domains with (unbounded) cyclic behavior of programs.

In short, this first cause of undecidability is the combination of arithmetic (yet decidable) concrete domains and cyclicity in the terminological axioms of the description logic.

4.8.2.2 Inadmissible Concrete Domains

Another reason why the resulting $\mathcal{ALC}(\mathcal{D})$ reasoning problems could be undecidable, is the need for inadmissible concrete domains. For instance, non-linear integer arithmetic (i.e. logical theories capable of expressing more than linear expressions, such as polynomial expressions in general) is undecidable, and therefore the corresponding concrete domain is inadmissible. If we allow the use of arithmetical operations like addition and multiplication without further restriction, in the programming languages, we end up needing such inadmissible concrete domains when encoding the programs and their behavior into $\mathcal{ALC}(\mathcal{D})$. This, therefore, leads to undecidability in the reasoning problems, even for non-looping programs.

In short, this second cause of undecidability is rooted in the fact that satisfiability problems for concrete domains that are expressive enough are undecidable.

CHAPTER 5

Benefits of Using Formal Logic

In Chapter 4, we developed a method of assigning a model-theoretic semantics to programs of different programming languages by encoding programs of different programming languages into description logics. This allows us to reduce the decision problem of deciding semantic properties of these programs to description logic reasoning problems. This method of encoding programs into formal logic, with the goal of deciding semantic properties of programs using reasoning algorithms based on the formal logic, has a number of advantages over developing special-purpose algorithms to decide such properties. We discuss the advantages that our framework offers over the direct approach of developing ad-hoc algorithms.

5.1 Conceptually Simplifying the Problem

The first class of advantages are related to conceptual advantages in investigating and classifying reasoning problems over programs. In short, these advantages boil down to the fact that our framework allows us to reduce various reasoning problems on a variety of structures to one well-studied logical setting.

For instance, the framework allows us to reduce the problem of deciding semantic properties of programs of the various programming languages to deciding semantic properties of description logic knowledge bases. In effect, we move from the setting of various different mathematical objects (the different programming languages and their semantics) to the well-known setting of formal logic. We are considering a number of different semantic properties of programs of a number of very diverse programming languages. In other words, we are considering quite some different decision problems on different structures (the syntax of the different languages is structurally diverse) with differently defined semantics (the programming languages have different semantics). In addition, we are considering a number of conceptually different problems. By encoding all these reasoning problems of deciding various semantic properties of various types of programs into reasoning problems of knowledge bases of one particular knowledge base, we reduce the semantic underpinnings of all these reasoning problems to one well-understood structure (namely description logic interpretations). In addition, since essentially all reasoning problems in description logics can be reduced to one single reasoning problem (namely ABox satisfiability), by means of our encoding we reduce the diverse set of decision problems involved in reasoning into one well-understood reasoning problem. Clearly, this is a major conceptual simplification.

Secondly, the reduction of the various reasoning problems to the problem of ABox satisfiability not only reduces the number of problems we are considering. The description logic satisfiability problem is conceptually very straightforward to solve. By virtue of the direct relation between the description logic syntax and their model-theoretic semantics, the design of algorithms to decide description logic satisfiability problems is directly guided by the definition of the semantics of the language. This point is also illustrated above, in the discussion of tableau algorithms for description logics, in Section 2.2. The fact that the problem can be conceived quite intuitively makes it much easier to develop highly sophisticated algorithms, dealing with complex variants of the problem. In fact, extremely sophisticated variants of the tableau algorithm have been developed for description logics with all sorts of complex extensions. The setting of finding a witness model of satisfiability for description logic knowledge bases, with the clear guidance of the semantics of the logic, is conceptually much simpler than the plethora of heterogeneously defined structures and semantics of the direct setting of designing ad hoc reasoning algorithms. This point will also be illustrated in Chapter 6, in Sections 6.3 and 6.4, where we will make heavy use of the semantical properties of description logic to identify additional fragments of the programming languages *While* and *Goto* for which solving certain reasoning problems is decidable.

Incidentally, another (more practical) advantage of transferring the reasoning problems we consider to the description logic setting, is that there has been more research on optimization of algorithms in this description logic setting. This is mainly due to the fact that description logics have seen more interest in the last few decades, because of their applicability in a general setting. However, the fact that this description logic setting is conceptually easier than many domain-specific settings, such as the setting we consider involving reasoning over various programming languages, is also very beneficial for the optimization possibilities of reasoning algorithms.

The conceptual simplification of the problem also has some advantages for the decidability and complexity analysis of programs from different fragments of the programming languages. Because the satisfiability problem of description logic knowledge bases underlies many important applications, a lot of research has been done on the decidability and complexity of this problem for different description logics. Partially due to the reason that the problem has a clear and intuitive model-theoretic semantics, extensive decidability and complexity results have been found. Even for the limited class of description logics with concrete domains, there is a magnitude of (non-trivial) decidability and complexity results that have been found (cf. [15, 16]). By connecting the reasoning problems of deciding semantic properties of programs with the description logic reasoning problems, we can use the results, ideas and techniques from the decidability and complexity analysis of description logics for analyzing the decidability and complexity of the programming language reasoning problems. The topic of decidability and computational complexity of reasoning over semantic properties of the programming languages is discussed in more detail in Chapter 6.

5.2 Identifying Abstract Reasoning Patterns

Connecting the setting of reasoning in our particular domain (i.e. deciding semantic properties of programs) to reasoning in the setting of formal logic also helps us to understand the abstract structure of reasoning in the domain of programming languages better.

Formal logics often reveal what reasoning structures can be used to reason in a particular domain. For examples of this, one can think of encoding planning problems or many other practical problems into the Boolean satisfiability problem (SAT), revealing the type and complexity of reasoning that is sufficient to solve such problems. In a similar way, our reduction of deciding semantic properties of programs of the different programming languages to $\mathcal{ALC}(\mathcal{D})$ satisfiability problems gives us a better idea of the type and complexity of reasoning needed to reason on programs. In particular, the encoding directly shows us what kind of reasoning structures are sufficient to solve the original problems.

Furthermore, conversely, the connection we established between the specification of computation in the form of programs and the semantics of description logics gives us insight into the power of description logic reasoning. A prime example of such novel insight is Theorem 4.8.1, which tells us that a particular type of description logic reasoning suffices to compute any Turing-computable function.

5.3 Obtaining a Flexible, Declarative and Unified Reasoning Framework

Another advantage of encoding the behavior of programs of the various programming languages into description logic knowledge bases is that by doing so we obtain a flexible, declarative and unified framework to solve the reasoning problems over programs.

Since we encode all programming languages into the same description logic, and since we are able to express the semantic properties of programs in this very same description logic, we get a unified reasoning framework. The same reasoning algorithms can be used to decide semantic properties of programs that possibly are of quite different nature. Also, the same algorithms can be used to decide various different semantic properties of programs.

The framework we end up with also has a declarative nature. As described in Chapter 4, the various different reasoning problems can be expressed using only the description logic language, describing the requirements on the behavior of programs in a declarative fashion. In order to express the semantic properties of programs for which automated reasoning is to be performed, there is no need to express any of the internal behavior of programs. In other words, the (non-declarative) behavior of the programming languages is hidden in the encoding, allowing us to specify reasoning problems purely declaratively.

The flexibility of the framework is a result of encoding the behavior of the different programming languages in the same model-theoretic semantics. The model-theoretic properties of multiple encodings can straightforwardly be related to each other using description logic statements. In this way, it is possible to add further programming languages to the framework and relate their encoding to the encoding of other programming languages. Furthermore, using the expressivity of description logic, it is straightforward to express additional semantic properties of programs for which automated reasoning is to be performed.

5.4 Integration with External Knowledge Bases

Another interesting possibility that the method of encoding reasoning over imperative programs into description logic offers us, is to combine reasoning over programs with ontological reasoning in the domain in which the imperative programs are applied. For instance, if a particular type of imperative programs is used in a medical domain, we can combine reasoning with the behavior of such programs with other types of medical knowledge that might be present.

CHAPTER 6

Decidability and Complexity Results

In the general case, checking termination of (*While* or *Goto*) programs is undecidable, as is checking equivalence of programs for any of the languages. This is proven for the languages *While* and *Goto* in Sections 6.3 and 6.4, respectively. In this chapter, we discuss a few ways of defining restricted fragments of the programming languages such that reasoning (i.e., deciding termination and equivalence of programs) becomes decidable. We will slightly abuse the terminology, and often call such fragments decidable fragments of the programming languages. We will focus the formal results in this chapter mainly on the imperative programming languages, and in particular on the programming language *While*. Similar results can be obtained for the other programming languages, and we will sketch how to obtain these results without going into full detail.

6.1 Finite Concrete Domains

A first way to define such a decidable fragment is to restrict the concrete domains to finite domains. An example of such a finite concrete domain that is used often in implementations of computational models would be based on the set of natural numbers representable by at most 16 binary digits, i.e. the set $\{0, \dots, 65535\}$. Such a restriction would correspond to the limitation of memory in practical computation, and therefore is plausible for practical settings. Concretely, an example of a finite concrete domain would be the restriction of the concrete domain $\mathcal{N}_{lin}$, defined in Chapter 2, to the subset of the domain $\{0, \dots, 65535\} \subseteq \mathbb{N}$.

In this section, we will give a proof of decidability for reasoning over programs, with the restriction that the concrete domain is finite. The idea behind this proof is as follows. If there are only finitely many concrete values, we can compile the working of the concrete domain concept constructs into regular description logic concepts. This way, we will end up with an encoding in regular $\mathcal{ALC}$ without concrete domains, for which concept satisfiability w.r.t. general TBoxes is decidable.

Theorem 6.1.1. *If we restrict the programming language* While *to a finite concrete domain, checking termination and equivalence of* While *programs is decidable.*

Proof (sketch). With finite concrete domains, the $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^p$ can be simulated by a regular $\mathcal{ALC}$ TBox $(\mathcal{T}^p)'$, by introducing concepts that simulate the behavior of the concrete values. Let the concrete domain have values $\{1, \dots, k\}$. We know we only have to consider a finite number of variables $\{x_1, \dots, x_m\} = X \subseteq \mathcal{X}$. Hence, we only have finitely many concrete features $\mathsf{valueOf}_{x_1}, \dots, \mathsf{valueOf}_{x_m}$. For each value $1 \leq i \leq k$ and each variable x_j, we introduce a fresh concept $E_{x_j,i}$, representing those objects who are related to value i with the concrete feature $\mathsf{valueOf}_{x_j}$. Since there are only finitely many variables, each predicate in the concrete domain is a relation consisting of finitely many tuples. We can now encode concepts like $\mathsf{valueOf}_x\!\uparrow$ as $\neg E_{x,1} \sqcap \cdots \sqcap \neg E_{x,k}$. Also, we can then encode concept constructions like $\exists(\mathsf{valueOf}_x)(\mathsf{nextState}\ \mathsf{valueOf}_y).\psi$, for some predicate ψ with extension $\{(v_1, v_1'), \dots, (v_n, v_n')\}$, as the concept $\neg\mathsf{valueOf}_x \uparrow \sqcap ((E_{x,v_1} \sqcap \exists\mathsf{nextState}.E_{y,v_1'}) \sqcup \cdots \sqcup (E_{x,v_n} \sqcap \exists\mathsf{nextState}.E_{y,v_n'}))$. A similar procedure works for predicates of any arity. This way, we end up with an encoding of any program p that consists of a general $\mathcal{ALC}$ TBox. Since we know that concept satisfiability with respect to general TBoxes is decidable for $\mathcal{ALCO}$, we know by the result from Section 4.3.3 that the problems of checking termination and equivalence of *While* programs are decidable in this setting. □

Similar results can be obtained for the other programming languages. We will not go into detail here, but we state the results that can be gotten.

Theorem 6.1.2. *If we restrict the programming language* Goto *to a finite concrete domain, checking termination and equivalence of* Goto *programs is decidable.*

Proof (sketch). Completely analogously to the proof of Theorem 6.1.1. □

Theorem 6.1.3. *If we restrict the programming language* FPN *to a finite concrete domain, checking equivalence of funclang programs is decidable.*

Proof (sketch). Completely analogously to the proof of Theorem 6.1.1. □

Theorem 6.1.4. *If we restrict the programming language* LPN *to a finite concrete domain, checking equivalence of* LPN *programs is decidable.*

Proof (sketch). Completely analogously to the proof of Theorem 6.1.1. □

The above decidability results (at least for *While* and *Goto*) can also be obtained in a different way, namely by using the notion of control states (see Definition 6.3.4 for *While* and Definition 6.4.4 for *Goto* below). Whenever the concrete domain is finite, there trivially exists a finite control state partition for any program, since for finite concrete domains the state space $\mathcal{S}_X$ for finite X is finite as well. The above decidability results then follow directly from Theorems 6.3.9 and 6.3.14 for *While*, and from Theorems 6.4.12 and 6.4.13 for *Goto*. For more details on these notions and results, see Sections 6.3 and 6.4.

6.1.1 Computational Complexity

In addition to the decidability results given above, we give an indication of the computational complexity of deciding termination and equivalence of programs that are restricted to finite concrete domains. In particular, we prove an EXPTIME upper bound (in the size of programs) on the complexity of deciding these properties for *While* programs. This is not a novel result (any naive approach that explores the search space in a brute force fashion would suffice to show this result[1]), but merely intended to indicate some complexity properties of algorithms to decide reasoning problems on programs by means of the encoding method proposed in this thesis.

Theorem 6.1.5. *If we restrict the programming language* While *to a finite concrete domain, checking termination and equivalence of* While *programs can be done in* $\mathcal{O}(2^{|p| \cdot 2^{|\mathcal{D}|}})$ *time.*

Proof (sketch). It is straightforward to verify that for any program p we have $|\mathcal{T}^p| = \mathcal{O}(|p|)$, and that for the concept C used for checking termination or equivalence we have $|C| = \mathcal{O}(1)$. By using the method in the proof of Theorem 6.1.1, we obtain an $\mathcal{ALC}$ TBox $(\mathcal{T}^p)'$ that simulates the $\mathcal{ALC}(\mathcal{D})$ TBox $\mathcal{T}^p$. By the construction of $(\mathcal{T}^p)'$ we know that $|(\mathcal{T}^p)'| = \mathcal{O}(|\mathcal{T}^p| \cdot 2^{|\mathcal{D}|})$ for $|\mathcal{D}|$ the size of the concrete domain. By the results in Section 4.3.3, we know that we can solve the problems of termination and equivalence for *While* programs by means of checking satisfiability with respect to general TBoxes. We know satisfiability for $\mathcal{ALC}$ and $\mathcal{ALCO}$ concepts C with respect to general TBoxes $\mathcal{T}$ is in EXPTIME [8, 14], i.e. can be done in time $\mathcal{O}(2^{|C| \cdot |\mathcal{T}|})$. Concludingly, we get a decision procedure for termination and equivalence of *While* programs that runs in time $\mathcal{O}(2^{|p| \cdot 2^{|\mathcal{D}|}})$. □

For the other programming languages *Goto*, *FPN* and *LPN*, in the setting of finitely bounded concrete domains, EXPTIME upper bounds on deciding termination (in case of *Goto*) and equivalence can be found in a similar fashion.

6.2 Non-Looping Programs

Another way to restrict programs of the different programming languages in such a way that the problems of checking termination and equivalence of programs become decidable, is to forbid any kind of looping behavior. This results in the control flow for any execution of the program to be fixed, which leads to decidability of reasoning over programs. Even though this is a very severe restriction on the computational power of the programming languages (i.e. without (conditional) loops, the languages are not Turing-complete anymore) this restriction still leads to cases that are interesting, at least from a practical point of view. For more details on an application concerning a real-world, industrial instance of the general problem in which no loops occur, see Chapter 7.

In order to define the proposed fragments of the programming languages *While*, *Goto*, *FPN* and *LPN* more precisely, we will define the notion of loop-free programs for each of these languages.

[1] In fact, such naive approaches can even be used to show better upper bounds.

Definition 6.2.1 (Loop-free *While* programs)**.** *We say that a* While *program p is* loop-free *if and only if it contains no subprograms of the form **while** b **do** q.*

Example 6.2.2. *Consider the following loop-free* While *program p, that swaps the values of variables x and y (by using a third variable z) iff the value of x exceeds the value of y:*

$$p = (\textbf{\textit{if}}\ (x > y)\ \textbf{\textit{then}}\ (z := x; x := y; y := z)\ \textbf{\textit{else}}\ skip)$$

Definition 6.2.3 (Loop-free *Goto* programs)**.** *We say that a* Goto *program κ of length l is* loop-free *if and only if for each $1 \leq i \leq l$ we have that $\kappa(i) =$ **if** b **goto** m_1 **else** m_2 implies that $m_1 > i$ and $m_2 > i$.*

Example 6.2.4. *Consider the following loop-free* Goto *program κ, that swaps the values of variables x and y (by using a third variable z) iff the value of x exceeds the value of y:*

$$\kappa = \begin{array}{rl} 1: & \textbf{\textit{if}}\ (x > y)\ \textbf{\textit{goto}}\ 2\ \textbf{\textit{else}}\ 5 \\ 2: & z := x \\ 3: & x := y \\ 4: & y := z \\ 5: & return \end{array}$$

Definition 6.2.5 (Loop-free *FPN* programs)**.** *We say that a* FPN *program π on a set of functional symbols F is* loop-free *if and only if the graph $G = (F, E)$ contains no cycles, where:*

$$E = \{(f, g) \mid f, g \in F, \pi(f) = ((c_1, e_1), \ldots, (c_m, e_m)), \textit{some } e_i \textit{ contains the symbol } g\}$$

Definition 6.2.6 (Loop-free *LPN* programs)**.** *We say that a* LPN *program p on a set of relational symbols R is* loop-free *if and only if the graph $G = (R, E)$ contains no cycles, where:*

$$E = \{(r, s) \mid r, s \in R, p \textit{ contains a rule for } r \textit{ in which } s \textit{ occurs in the body}\}$$

We will show that for each of the four programming languages we get that reasoning on loop-free programs is decidable, by showing that in each case the encoding of programs can be considered as an acyclic $\mathcal{ALC}(\mathcal{D})$ TBox, for which satisfiability is decidable.

Theorem 6.2.7. *Termination and equivalence of loop-free* While *programs are decidable.*

Proof. We firstly show that for any loop-free *While* program p the encoding $\mathcal{T}^p$ into $\mathcal{ALC}(\mathcal{D})$ consists of an acyclic TBox. Let p be an arbitrary loop-free *While* program, and let $\mathcal{T}^p$ be its encoding into $\mathcal{ALC}(\mathcal{D})$. By the fact that p is loop-free, we know that p contains no statements of the form **while** b **do** q. This means that no instance of Axiom (4.15) occurs in $\mathcal{T}^p$. The following linear order $<$ on the concepts C_q for $q \in cl(p)$ and D_b for $b \in Bool(p)$ witnesses that $\mathcal{T}^p$ can be considered as an acyclic TBox, i.e. each axiom in $\mathcal{T}^p$ is of the form $C \sqsubseteq D$ for a concept name C and a (possibly) complex concept D containing only concept names C' such that $C > C'$, and is to be replaced by the axiom $C \equiv D$. Let $<$ be induced by:

$$\begin{array}{rl} C_q > D_b & \text{for any } C_q \text{ and any } D_b \\ D_b > D_{b'} & \text{for any } b \in Bool(p) \text{ and any subexpression } b' \text{ of } b \\ C_{q;q'} > C_{q'} & \text{for any } q; q' \in cl(p) \\ C_{\textbf{if}\ b\ \textbf{then}\ q_1\ \textbf{else}\ q_2} > C_q & \text{for any } \textbf{if}\ b\ \textbf{then}\ q_1\ \textbf{else}\ q_2 \in cl(p) \text{ and } q \in \{q_1, q_2\} \end{array}$$

The constructions and arguments from Chapter 4 (i.e., Lemma 4.3.1, Theorem 4.3.2, Theorem 4.3.3 and Theorem 4.3.4) also work for the acyclic version of the TBox $\mathcal{T}^p$. This is straightforward to verify. Furthermore, when checking satisfiability of a concept with respect to an acyclic TBox, we can unfold the concept definitions from the TBox in the concept. Thus we can reduce the problem of satisfiability with respect to the TBox to the satisfiability problem of a single concept.

Now, by the results in Section 4.3.3, we know that termination and equivalence of *While* programs can be reduced to (pure) concept satisfiability of $\mathcal{ALCO}(\mathcal{D})$, which we know to be decidable [14]. □

The very same proof idea works for the other programming languages as well. We will not spell out the details, but merely state the results.

Theorem 6.2.8. *Termination and equivalence of loop-free* Goto *programs are decidable.*

Proof (sketch). Analogous to the proof of Theorem 6.2.7. The linear order that witnesses the acyclicity of the encoding corresponds to the linear order $<$ on line numbers. □

Theorem 6.2.9. *Termination and equivalence of loop-free* FPN *programs are decidable.*

Proof (sketch). Analogous to the proof of Theorem 6.2.7. The linear order that witnesses the acyclicity of the encoding is induced by the graph G in the definition of loop-free *FPN* programs. □

Theorem 6.2.10. *Termination and equivalence of loop-free* LPN *programs are decidable.*

Proof (sketch). Analogous to the proof of Theorem 6.2.7. The linear order that witnesses the acyclicity of the encoding is induced by the graph G in the definition of loop-free *LPN* programs. □

6.2.1 Computational Complexity

Similar to the case of programs restricted to finite concrete domains, we give an indication of the computational complexity of deciding equivalence of loop-free programs. Note that the termination problem of loop-free programs is trivial, since loop-free programs always terminate. Similarly to the complexity results in Section 6.1.1, this result is merely intended to indicate some complexity properties of algorithms to decide reasoning problems on programs by means of the encoding method proposed in this thesis. Note that in the proof of the following theorem, we refer to notions and results from Section 6.3 below.

Theorem 6.2.11. *Checking the equivalence of two loop-free* While *programs p_1 and p_2 can be done in* NEXPTIME *in* $|p_1| + |p_2|$.

Proof (sketch). We use a decision procedure similar to the one used in Theorem 6.3.14. We make use of the fact that for loop-free *While* programs every derivation has one of finitely many derivation structures (see Definition 6.3.10). Similarly to the procedure of Theorem 6.3.14 (by using Lemma 6.3.12), we can then check equivalence of p_1 and p_2 by checking for only a bounded number of structures $\mathcal{L}$ whether they can be extended to a suitable model. In fact, for loop-free *While* programs there are only $\mathcal{O}(|p|)$ such structures that need to be considered. The NEXPTIME result can then straightforwardly be obtained by an argument similar to the one in the proof of Theorem 6.3.16. □

6.3 Structuring Variable States for *While*

In order to characterize classes of programs for which the semantic properties of termination and equivalence of programs are decidable, we will identify a particular structure in the set of input states on which programs can be executed. In particular, we will define a notion of control states. Intuitively, control states for a program p are sets of states for which the control flow of the execution of p is identical, regardless of what particular state s from this set of states is used as input state (for relevant subprograms of p). We will use partitions of the set of states into control states in identifying decidable classes of programs.

This approach to structuring the infinite space of states into a finite number of equivalence classes is similar to approaches to model checking infinite state systems. In fact, our setting is in essence also an infinite state system. For infinite state systems such finite partitions of the state space, that satisfy certain properties allowing them to be used for model checking, are often called finite bisimulations. Bisimulation is a notion of behavioral equivalence that is well-known in the field theoretical computer science and logic. An example of infinite state systems for which finite bisimulations are used for formal verification are hybrid automata (cf. [12, 1]).

6.3.1 Control State Partitions

In order to define the notion of control state partitions for programs p, we must firstly define the following two notions of maximal condition free subprograms and maximal Boolean conditions. Maximal condition free subprograms, intuitively, are the largest subprograms that don't contain any statements involving any conditional statements. In other words, these are the maximal subprograms for which the control flow is entirely linear. Maximal Boolean conditions are, as the name suggests, the maximal Boolean subexpressions occurring in a program.

We will use these notions to confine the control flow of programs. The behavior of these two elements of programs (i.e. maximal Boolean conditions and maximal condition free subprograms) completely determines the control flow of programs, as we will see below. The reason for this, intuitively, is that all conditional elements of a program depend on these maximal Boolean conditions, and that besides conditional elements programs consist only of the maximal condition free subprograms.

Definition 6.3.1 (Maximal condition free subprograms). *For a given* While *program p, we define* *the set $\mathcal{G}^p$ of maximal condition free subprograms as*

$$\mathcal{G}^p = \begin{cases} \emptyset & \text{if } p = skip \\ \{q_1; \dots ; q_n\} \cup \mathcal{G}^r & \text{if } p = q_1; \dots ; q_n; r \text{ for some } n \geq 1, \\ & \text{for all } i,\ q_i \text{ is either of the form } (x_i := e_i) \text{ or of the form } skip, \\ & \text{and } r \text{ is not of the form } (r'; r'') \text{ nor of the form } r', \\ & \text{for any } r' \text{ of the form } (x := e) \text{ or } skip \\ \mathcal{G}^{q_1} \cup \mathcal{G}^{q_2} & \text{if } p = \textbf{\textit{if }} b \textbf{\textit{ then }} q_1 \textbf{\textit{ else }} q_2 \\ \mathcal{G}^q & \text{if } p = \textbf{\textit{while }} b \textbf{\textit{ do }} q \end{cases}$$

Definition 6.3.2 (Maximal Boolean conditions)**.** *For any given* While *program p, we define the set $\mathcal{H}^p$ of all maximal Boolean conditions occurring in p as*

$$\mathcal{H}^p = \begin{cases} \mathcal{H}^{q_1} \cup \mathcal{H}^{q_2} & \text{if } p = q_1; q_2 \\ \emptyset & \text{if } p = skip \\ \{b\} \cup \mathcal{H}^{q_1} \cup \mathcal{H}^{q_2} & \text{if } p = \textbf{\textit{if }} b \textbf{\textit{ then }} q_1 \textbf{\textit{ else }} q_2 \\ \{b\} \cup \mathcal{H}^q & \text{if } p = \textbf{\textit{while }} b \textbf{\textit{ do }} q \end{cases}$$

Note that for any *While* program p, both sets $\mathcal{G}^p$ and $\mathcal{H}^p$ can be determined purely syntactically.

Clearly, for any *While* program p we have that all $g \in \mathcal{G}^p$ are terminating (on all input states). In order to fix notation, given any *While* program q containing variables $X \subseteq \mathcal{X}$ and any input state $s \in \mathcal{S}_X$, we denote the unique state $t \in \mathcal{S}_X$ such that $(q, s) \Rightarrow^* t$ by $q(s)$.

In order to illustrate these notions, we will give the sets $\mathcal{G}^p$ and $\mathcal{H}^p$ for the *While* program p in our running example.

Example 6.3.3. *Remember from Example 3.1.1 that*

$$p = (z := 0; w := 0; \overbrace{\textbf{\textit{while}}\ (w < x \wedge w < 3)\ \textbf{\textit{do}}\ (z := z + y; w := w + 1); skip}^{p'})$$

We then have that

$$\mathcal{G}^p = \{(z := 0; w := 0), (z := z + y; w := w + 1), skip\}$$

$$\mathcal{H}^p = \{(w < x \wedge w < 3)\}$$

With these preliminary notions in place, we are now ready to define the notions of control states and control state partitions.

Definition 6.3.4 (Control state partitions)**.** *For any* While *program p containing variables $X \subseteq \mathcal{X}$, we define a* control state partition *for p to be a family $(P_i)_{i \in I}$ of subsets of states (called* control states*) $P_i \subseteq \mathcal{S}_X$ such that:*

1. *$\cup_{i \in I} P_i = \mathcal{S}_X$ and $P_i \cap P_j = \emptyset$ for all $i, j \in I$ such that $i \neq j$;*

2. *for all $i \in I$, all $s, t \in P_i$ and all $h \in \mathcal{H}^p$ we have that $\mathcal{B}^X(s,h) = \mathcal{B}^X(t,h)$; and*

3. *for all $i, j \in I$, all $s, t \in P_i$ and all $g \in \mathcal{G}^p$ we have that $g(s) \in P_j$ iff $g(t) \in P_j$; and*

4. *for all $i \in I$, and for each subset $S \subseteq \mathcal{S}_X$, if $P_i \cap S \neq \emptyset$ then some state $s \in P_i \cap S$ must be effectively constructible.*

The conditions on control state partitions $(P_i)_{i \in I}$ can be explained intuitively as follows. Condition (1) requires that $(P_i)_{i \in I}$ in fact partitions the set of states $\mathcal{S}_X$. Condition (2) ensures that with the information in what partition P_i a given state $s \in \mathcal{S}_X$ is, one can determine the value of any Boolean expression $b \in \mathcal{H}^p$ in state s. Similarly, condition (3) ensures that the information in what partition P_i a given state $s \in \mathcal{S}_X$ is uniquely determines in what partition P_j the resulting state $g(s)$ is, for any given maximal condition free subprogram $g \in \mathcal{G}^p$. Finally, condition (4) ensures that it is possible to find a representative state for each control state. Furthermore, condition (4) is designed in a way that makes it possible to combine multiple control state partitions into one (see Theorem 6.3.15).

For a control state partition $(P_i)_{i \in I}$ for a program p containing variables X, and for a state $s \in \mathcal{S}_X$, we let $P(s)$ denote the unique P_i such that $s \in P_i$.

Example 6.3.5. *Consider the following control state partition $(P_i)_{i \in I}$ for our running example program p.*

$$I = \{(n,m) \mid 0 \leq n, m \leq 3\}$$

$$f(n) = \begin{cases} 3 & \textit{if } n \geq 3 \\ n & \textit{otherwise} \end{cases}$$

$$s \in P_{(f(s(x)), f(s(w)))} \textit{ for any } s \in \mathcal{S}_{\{w,x,y,z\}}$$

Remember from Example 6.3.3 that

$$\mathcal{G}^p = \{(z := 0; w := 0), (z := z + y; w := w + 1), skip\}$$

$$\mathcal{H}^p = \{(w < x \wedge w < 3)\}$$

It is easy to see that $(P_i)_{i \in I}$ partitions the state space $\mathcal{S}_{\{w,x,y,z\}}$. It is also straightforward to verify that for any state $s \in \mathcal{S}_{\{w,x,y,z\}}$ the control state $P(s)$ of s determines the evaluation of $(w < x \wedge w < 3)$. Intuitively, this holds because the Boolean statement cannot distinguish values of x that are at least 3, and similarly for w. To see that the third condition of control state partitions holds, note that for any $P_{(i,j)}$ and any state $s \in P_{(i,j)}$, the (sub)program $(z := 0; w := 0)$ executed on s results in a state in $P_{(i,0)}$. Also, for any $P_{(i,j)}$ and any state $s \in P_{(i,j)}$, the (sub)program $(z := z + y; w := w + 1)$ executed on s results in a state in $P_{(i,f(j))}$.

6.3.2 Programs With Finite Control State Partitions

Using the notion of control state partitions defined above, we are able to identify a suitable fragment of the programming language *While*. This fragment is the set of those *While* programs for which there is a control state partition of finite size. In order to prove that deciding termination and equivalence of *While* programs from this fragment, we need some auxiliary results. The first of these auxiliary results concerns the notion of derivation structures, and states that the control state of the initial state of a derivation determines the structure of the derivation.

Definition 6.3.6. *For any finite derivation* d *of the form* $(p_1, s_1) \Rightarrow \cdots \Rightarrow (p_n, s_n) \Rightarrow s_{n+1}$ *in the operational semantics of the programming language* While*, we define the* structure *of* d *to be the (finite) sequence* $p_1, \ldots, p_n$ *of programs occurring in the derivation.*

Similarly, for any infinite derivation d *of the form* $(p_1, s_1) \Rightarrow (p_2, s_2) \Rightarrow \ldots$*, we define the* structure *of* d *to be the (infinite) sequence* $p_1, p_2, \ldots$*.*

Lemma 6.3.7. *Let* p *be an arbitrary* While *program, and let* $(P_i)_{i \in I}$ *be a control state partition for* p*. For any* $i \in I$ *and for any two states* $s, t \in P_i$ *we have that the derivation starting with* (p, s) *and the derivation starting with* (p, t) *have the same structure.*

Proof. In this proof we will use the fact that the operational semantics for *While* is deterministic, i.e. for any (p, s) there is exactly one derivation starting at (p, s). Fix arbitrary $i \in I$ and $s, t \in P_i$. Firstly, consider the case in which both derivation starting with (p, s) and (p, t) are finite. Let $(p, s) = (p_1, s_1) \Rightarrow \cdots \Rightarrow (p_n, s_n) \Rightarrow s'$ be the derivation starting with (p, s), and let $(p, t) = (p'_1, t_1) \Rightarrow \cdots \Rightarrow (p'_m, t_m) \Rightarrow t'$ be the derivation starting with (p, t). We assume w.l.o.g. that $n \leq m$. By complete induction on the index i of programs p_i we show for all i that $p_i = p'_i$, and that $P(s_i) = P(t_i)$ for all i such that $i = 1$, $i = n$, or at least one of p_i and p_{i-1} is not of the form $x := e; q$. The base case for $i = 1$, follows by assumption.

For the inductive case, assume that $p_{i-1} = p'_{i-1}$. We distinguish several cases. Consider the case in which p_{i-1} is of the form $x := e; q$. By definition of the derivational semantics, we get that $p_i = q = p'_i$. If in this case $p_i = q$ is not of the form $x := e; q'$, we need to show $P(s_i) = P(t_i)$. Assume thus that q is not of the form $x := e; q'$. Now, let k be the maximal number such that for all $1 \leq j \leq k$ we have that p_{i-j} is of the form $x := e; q'$. We then know that $p_{i-k} = g; q$ for some $g \in \mathcal{G}^p$, by construction of $\mathcal{G}^p$. We then also know by the induction hypothesis that $P(s_{i-k}) = P(t_{i-k})$, since either $i - k = 1$ or p_{i-k-1} is not of the form $x := e; q'$. Now, by the facts that $s_i = g(s_{i-k})$, $t_i = g(t_{i-k})$, $P(s_{i-k}) = P(t_{i-k})$, and by condition (3) in the definition of control state partitions, we know that $P(s_i) = P(t_i)$.

Consider the case in which p_{i-1} is of the form $(\textbf{if } b \textbf{ then } q_1 \textbf{ else } q_2); q$. By the induction hypothesis, we know that $p'_{i-1} = p_{i-1}$. Also, by the induction hypothesis, we know that $P(s_{i-1}) = P(t_{i-1})$. Then, since $b \in \mathcal{H}^p$, by condition (2) in the definition of control state partitions, we know that $\mathcal{B}^X(s_{i-1}, b) = \mathcal{B}^X(t_{i-1}, b)$. From this we can conclude that $p_i = p'_i = q_j; q$ for some $j \in \{1, 2\}$. Also, by the definition of the operational semantics we get that $s_i = s_{i-1}$ and $t_i = t_{i-1}$, and thus that $P(s_i) = P(t_i)$.

Consider the case in which p_{i-1} is of the form $(\textbf{while } b \textbf{ do } q'); q$. By the induction hypothesis, we know that $p'_{i-1} = p_{i-1}$. By the definition of the operational semantics we can conclude that $p_i = p'_i = (\textbf{if } b \textbf{ then } q'; (\textbf{while } b \textbf{ do } q') \textbf{ else } skip); q$. Also, by the definition of the operational semantics we get that $s_i = s_{i-1}$ and $t_i = t_{i-1}$, and since we know $P(s_{i-1}) = P(t_{i-1})$ by the induction hypothesis, we have that $P(s_i) = P(t_i)$.

In the case in which $p_{i-1} = p'_{i-1}$ is of the form $skip; q$, we know by the operational semantics that $p_i = p'_i = q$. By the operational semantics, we also know that $s_i = s_{i-1}$ and

$t_i = t_{i-1}$, and since we know $P(s_{i-1}) = P(t_{i-1})$ by the induction hypothesis, we have that $P(s_i) = P(t_i)$.

Finally, we have that $p_{i-1} = p'_{i-1} = skip$ if and only if $i - 1 = n$, and thus by the definition of the operational semantics, there are no p_i and p'_i. This implies that $n = m$. This concludes our induction, proving that the structure of the derivations starting at (p, s) and (p, t) are the same.

The inductive argument presented above also proves the following statements. If one of the derivations starting at (p, s) and (p, t) is finite, then so is the other. If both derivations starting at (p, s) and (p, t) are infinite, then they have the same structure. This concludes our proof that for any $i \in I$ and any $s, t \in P_i$, the structures of the derivations starting at (p, s) and (p, t) are the same. □

This result immediately leads to the insight that for programs with a control state partition of finite size only finitely many different, structurally distinct derivations needs to be considered. This insight directly allows us to decide termination for such programs.

Corollary 6.3.8. *For any* While *program p for which there exists a control state partition $(P_i)_{i \in I}$ with finitely many control states (i.e. I is finite), then there is a finite set of derivation structures such that each derivation of p has a structure in this set.*

Theorem 6.3.9. *For any* While *program p for which there exists a control state partition $(P_i)_{i \in I}$ with finitely many control states (i.e. I is finite), it is decidable to check whether p terminates on all inputs.*

Proof. Let p be a program with control state partition $(P_i)_{i \in I}$ for finite I. Existence of a nonterminating execution of p on some state s can be checked by executing p on all the representative states s_i for each P_i (which are finitely many). If executing p on s_i terminates, then by Lemma 6.3.2 we know that it terminates on all $s' \in P_i$. If executing p on all s_i terminates, we thus know p terminates on all states, since $\mathcal{S}_X = \bigcup_{i \in I} P_i$. Also, it can be decided if executing p on s_i does not terminate. If at any point in the derivation there occurs a pair (p', s') such that there has occurred a pair (p', s'') before with $\{s', s''\} \subseteq P_j$ for some $j \in I$, then we know the derivation loops infinitely (by Lemma 6.3.2). Since only a bounded number of programs p' can occur (namely $|cl(p)|$ many) and only a bounded number of control states exist (namely $|I|$ many), we know that within a certain number of derivation steps (namely $|cl(p)| \cdot |I|$ many) either the derivation has terminated or such a loop has occurred. □

In order to show that equivalence of *While* programs allowing finite control state partitions is decidable, we introduce the notion of derivation frames. In order to define this notion, we need to consider the definition of stepwise divisions of *While* derivations, defined in Section 3.1.1.

Definition 6.3.10 (Derivation frames). *For any* While *program p containing variables X and any state $s \in \mathcal{S}_X$, we define the* derivation frame *corresponding to the (finite) derivation starting with (p, s) as the interpretation $\mathcal{J}^{(p,s)} = (\Delta^{\mathcal{J}}, \cdot^{\mathcal{J}})$ that is defined as follows. Let $d_1, \ldots, d_n$ be the stepwise division of the derivation starting with (p, s). We let $\Delta^{\mathcal{J}} = \{d_1, \ldots, d_n\}$. For each $d_i \in \Delta^{\mathcal{J}}$ with structure $p_1, \ldots, p_n$ and for each $q \in cl(p)$ we let $d_i \in C_q^{\mathcal{J}}$ iff $q \in \{p_1, \ldots, p_n\}$. We let* $\mathsf{nextState}^{\mathcal{J}} = \{(d_i, d_{i+1}) \mid 1 \leq i < n\}$.

Observation 6.3.11. *The derivation frame of any derivation is uniquely determined by the structure of the derivation.*

In order to prove that equivalence of *While* programs with finite control state partitions is decidable, the following result will play an essential role. We show that for any model of the encoding of a *While* program into $\mathcal{ALC}(\mathcal{D})$, the maximal $\mathsf{nextState}$ connected component containing the element corresponding to the start of a derivation has the structure of the frame corresponding to this derivation.

Lemma 6.3.12. *Let p be any* While *program containing variables $X = \{x_1, \dots, x_n\}$, and fix an arbitrary state $s \in \mathcal{S}_X$. For any model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^p$ containing an element $d \in \Delta^{\mathcal{I}}$ such that for all x_i it holds $(d, s(x_i)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_i}$, we have that there exists a homomorphic mapping $\mu : \mathcal{J}^{(p,s)} \to \mathcal{I}$ from the derivation frame $\mathcal{J}^{(p,s)}$ into $\mathcal{I}$ such that $\mu(p, s) = d$, i.e.:*

- *for any concept C and any object $e \in \Delta^{\mathcal{J}^{(p,s)}}$ we have $e \in C^{\mathcal{J}^{(p,s)}}$ implies $\mu(e) \in C^{\mathcal{I}}$;*
- *for any role R and any objects $e, f \in \Delta^{\mathcal{J}^{(p,s)}}$ we have $(e, f) \in R^{\mathcal{J}^{(p,s)}}$ implies $(\mu(e), \mu(f)) \in R^{\mathcal{I}}$;*
- *$\mu(p, s) = d$.*

Furthermore, the submodel of $\mathcal{I}$ induced by $rng(\mu)$ is maximally connected with respect to $\mathsf{nextState}$*, i.e. there exists no $e \in rng(\mu)$ and $f \in \Delta^{\mathcal{I}} \backslash rng(\mu)$ such that $(e, f) \in \mathsf{nextState}^{\mathcal{I}}$.*

Proof. We construct a suitable mapping μ by induction on the length of the stepwise division $d_1, \dots, d_n$ of the derivation starting with (p, s) (we omit the final element s' in this derivation). Simultaneously, we prove by induction on the length of the stepwise devision that (1) for each d_i and each x_j we have that $(\mu(d_i), s_i(x_j)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_j}$ where s_i is the state of the subderivation d_i, that (2) for each d_i we have that $\mu(d_i) \in C^{\mathcal{I}}_{p'}$ for all p' occurring in d_i, and that (3) for each $1 \leq i < n$ we have that $(\mu(d_i), \mu(d_{i+1})) \in \mathsf{nextState}^{\mathcal{I}}$.

In the base case, for d_1, we define $\mu(d_1) = d \in \Delta^{\mathcal{I}}$. Let $d_1 = [(p_1, s) \Rightarrow \cdots \Rightarrow (p_r, s)]$. Clearly, we have that $(\mu(d_1), s(x_j)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_j}$ for all x_j. We prove by induction on the length of d_1 that $\mu(d_1) \in C^{\mathcal{I}}_{p_k}$ for all $1 \leq k \leq r$. The case for p_1 holds by assumption, since $d \in C^{\mathcal{I}}_{p_1}$. In the inductive case for p_k we distinguish different cases based on the form of p_{k-1}. The case for $p_{k-1} = (\textbf{if } b \textbf{ then } q_1 \textbf{ else } q_2); p'$ follows from the facts that $\mu(d_1) \in C^{\mathcal{I}}_{p_{k-1}}$, $(\mu(d_1), s(x_j)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_j}$ for all x_j, from the fact that $\mathcal{I}$ satisfies Axiom (4.14), by the definition of the operational semantics and by Lemma 4.3.1 in Chapter 4. Similarly, the case for $p_{k-1} = (\textbf{while } b \textbf{ do } q); p'$ follows by using the induction hypothesis, the definition of the operational semantics the fact that $\mathcal{I}$ satisfies Axiom (4.15) and Lemma 4.3.1 in Chapter 4. The case for $p_{k-1} = \textbf{skip}; p'$ follows in a similar way by using the induction hypothesis, the definition of the operational semantics and the fact that $\mathcal{I}$ satisfies Axiom (4.8). This completes our induction on the length of d_1.

In the inductive case (for the induction on the stepwise division) we define $\mu(d_i)$. By the induction hypothesis, we know $\mu(d_{i-1})$ is already defined, and we know that $(\mu(d_{i-1}), s_{i-1}(x_j)) \in$

$\mathsf{valueOf}^{\mathcal{I}}_{x_j}$ for all x_j, where s_{i-1} is the state of d_{i-1}. Then, by the fact that $\mathcal{I}$ satisfies Axiom (4.7), we know there exists some $d' \in \Delta^{\mathcal{I}}$ such that $(\mu(d_{i-1}), d') \in \mathsf{nextState}^{\mathcal{I}}$. Define $\mu(d_i) = d'$. Then, by the fact that $\mathcal{I}$ satisfies Axioms (4.9)-(4.13), and by the definition of the operational semantics, we know that $(\mu(d_i), s_i(x_j)) \in \mathsf{valueOf}^{\mathcal{I}}_{x_j}$ for all x_j. Then, by induction on the length of the subderivation $d_i = [(p_1, s) \Rightarrow \cdots \Rightarrow (p_r, s)]$ (similar to the inductive argument given for the base case for d_1), we know that $\mu(d_i) \in C^{\mathcal{I}}_{p_k}$ for all $1 \leq k \leq r$.

Since the only relevant role is nextState and the only relevant concepts are the concepts C_{p_k} for the programs p_k occurring in each subderivation, this inductive construction of μ combined with the inductive proof suffices to show the existence of a suitable homomorphic mapping μ.

The fact that the submodel of $\mathcal{I}$ induced by $\{\mu(d_1), \ldots, \mu(d_n)\}$ is maximally connected w.r.t. nextState follows from the fact that nextState is an abstract feature, the fact that for each $1 \leq i < n$ we have $(\mu(d_i), \mu(d_{i+1})) \in \mathsf{nextState}^{\mathcal{I}}$ and the fact that for no $e \in \Delta^{\mathcal{I}}$ it holds that $(\mu(d_n), e) \in \mathsf{nextState}^{\mathcal{I}}$ (since $e \in C^{\mathcal{I}}_{skip}$ and $\mathcal{I}$ satisfies Axiom (4.2)). □

Lemma tells us that for programs with a finite control state partition, there are only finitely many different derivation frames. This fact, together with Lemma 6.3.12, gives us the following insight.

Corollary 6.3.13. *Let p be a program for which there exists a control state partition $(P_i)_{i \in I}$ with finitely many control states (i.e. I is finite). Then for any model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{T}^p$ with some $d \in \Delta^{\mathcal{I}}$ such that $d \in C^{\mathcal{I}}_p$, there is a finite set of different derivation frames $J = \{\mathcal{J}_1, \ldots, \mathcal{J}_n\}$ such that for at least one $\mathcal{J} \in J$ there is a homomorphic mapping μ from $\mathcal{J}$ to $\mathcal{I}$ such that the unique* nextState*-initial element from $\mathcal{J}$ is mapped to d.*

With the above results, we are able to devise the following algorithm to decide equivalence of *While* programs that allow for finite control state partitions.

Theorem 6.3.14. *For any two* While *programs p_1 and p_2 for which there exists a control state partition $(P_i)_{i \in I}$ with finitely many control states (i.e. I is finite), it is decidable to check whether p_1 and p_2 are equivalent.*

Proof. As shown before, in Chapter 4, we know this problem can be reduced to the problem of $\mathcal{ALCO}(\mathcal{D})$ unsatisfiability of the ABox

$$\mathcal{A}^{p_1,p_2} = \{o : C_{p_1}, o : C_{p_2}, s : C_{test}\}$$

with respect to the TBox $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$ where $\mathcal{S}^{p_i}$ is the modification of $\mathcal{T}^{p_i}$ where C_{skip} is replaced by C^i_{skip}, and where C_{test} is an abbreviation for

$$\begin{aligned}
&\exists \mathsf{res}_1.C^1_{skip} \sqcap \exists \mathsf{res}_2.C^2_{skip} \sqcap \\
&(\exists(\mathsf{res}_1\ \mathsf{valueOf}_{x_1})(\mathsf{res}_2\ \mathsf{valueOf}_{x_1}).\neq \\
&\sqcup \cdots \sqcup \\
&\exists(\mathsf{res}_1\ \mathsf{valueOf}_{x_n})(\mathsf{res}_2\ \mathsf{valueOf}_{x_n}).\neq)
\end{aligned}$$

where res_1 and res_2 are abstract features, and C^1_{skip}, C^2_{skip} and C_{test} nominal concepts.

We know that for any model $\mathcal{M}$ of $\mathcal{A}^{p_1,p_2}$ and $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$ the interpretation $\mathcal{J} = (\Delta^{\mathcal{J}}, \cdot^{\mathcal{J}})$, defined below, must be homomorphically embeddable in $\mathcal{M}$. We define $\mathcal{J}$ by letting $\Delta^{\mathcal{J}} = \{i, f_1, f_2, s\}$, $C_{p_1}^{\mathcal{J}} = C_{p_2}^{\mathcal{J}} = \{i\}$, $C_{test}^{\mathcal{J}} = \{s\}$, $(C_{skip}^1)^{\mathcal{J}} = \{f_1\}$, $(C_{skip}^2)^{\mathcal{J}} = \{f_2\}$, $\mathsf{res}_1^{\mathcal{J}} = \{(s, f_1)\}$ and $\mathsf{res}_2^{\mathcal{J}} = \{(s, f_2)\}$.

We can check the satisfiability of $\mathcal{A}^{p_1,p_2}$ w.r.t. $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$ by checking for a finite number of structures $\mathcal{L} = (\Delta^{\mathcal{L}}, \cdot^{\mathcal{L}})$ whether the interpretation function $\cdot^{\mathcal{L}}$ can be extended in such a way that the (extended) structure satisfies the ABox and TBox. These structures $\mathcal{L}$ are all the different possibilities of combining $\mathcal{J}$ with $\mathcal{J}^{(p_1,s_i)}$ and $\mathcal{J}^{(p_2,s_i)}$, for all $i \in I$ where s_i is representative for P_i, where we identify i with the unique $\mathsf{nextState}$-initial objects in $\mathcal{J}^{(p_1,s_i)}$ and $\mathcal{J}^{(p_2,s_i)}$.

Clearly, if any such structure $\mathcal{L}$ can be extended to a model, then we have found a witness of satisfiability. Conversely, we show that if no such structure can be extended to a satisfying structure (by extending $\cdot^{\mathcal{L}}$), then we know that $\mathcal{A}^{p_1,p_2}$ is unsatisfiable w.r.t. $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$.

Assume no structure $\mathcal{L}$ can be extended to a model of $\mathcal{A}^{p_1,p_2}$ and $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$. Now, assume to the contrary that there exists a model $\mathcal{I}$ of $\mathcal{A}^{p_1,p_2}$ and $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$. We know that $\mathcal{J}$ can be homomorphically embedded in $\mathcal{I}$. Let μ be the homomorphic mapping witnessing this. Since $\mathcal{I}$ satisfies $\mathcal{S}^{p_1}$ and $\mathcal{S}^{p_2}$, we know $(\mu(i), s(x_j)) \in \mathsf{valueOf}_{x_j}^{\mathcal{I}}$ for some state $s \in \mathcal{S}_X$. We also know then, by Lemma 6.3.12, that $\mathcal{J}^{(p_1,s)}$ and $\mathcal{J}^{(p_2,s)}$ are homomorphically embeddable into $\mathcal{I}$, say with homomorphic mappings μ_1 and μ_2, respectively. Since C_{skip}^1 and C_{skip}^2 are nominal concepts, we know that the unique $\mathsf{nextState}$-final elements in $\mathcal{J}^{(p_1,s)}$ and $\mathcal{J}^{(p_2,s)}$ are mapped to $\mu(f_1)$ with μ_1 and to $\mu(f_2)$ with μ_2, respectively. Now, consider the submodel $\mathcal{I}'$ of $\mathcal{I}$ induced by $rng(\mu) \cup rng(\mu_1) \cup rng(\mu_2)$, with the interpretation function $\cdot^{\mathcal{I}}$ restricted to $rng(\mu_1) \cup rng(\mu_2)$ for all concepts and roles occurring in $\mathcal{S}^{p_1}$ and $\mathcal{S}^{p_2}$. It is straightforward to verify that $\mathcal{I}'$ is a model of $\mathcal{A}^{p_1,p_2}$ and $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$. Also, it is straightforward to verify that $\mathcal{I}'$ can be obtained by extending the combination of $\mathcal{J}$ with $\mathcal{J}^{(p_1,s)}$ and $\mathcal{J}^{(p_2,s)}$. This is a contradiction with our first assumption. Thus, we can conclude that no model $\mathcal{I}$ of $\mathcal{A}^{p_1,p_2}$ and $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$ exists.

Furthermore, we know there are only finitely many candidate structures $\mathcal{L}$ that are to be extended, and there are only finitely many relevant extensions of $\cdot^{\mathcal{L}}$ (since each $\mathcal{L}$ has finitely many objects and only finitely many concepts occur in the TBox $\mathcal{S}^{p_1} \cup \mathcal{S}^{p_2}$). Also, by the admissibility of the concrete domain, we know for each extension of $\cdot^{\mathcal{L}}$ to the interpretation of the abstract concepts and features, it is decidable to check whether this interpretation can be extended to an interpretation of the concrete features. Hence, we get decidability of the equivalence problem. □

Note that Theorem 6.3.14 assumes that there is a single finite control state partition $(P_i)_{i \in I}$ that satisfies the properties given in Definition 6.3.4 for two programs p and q at the same time. For the decidability result, it suffices, however, to have a finite control state partition $(P_i)_{i \in I}$ for program p and a (possibly different) finite control state partition $(Q_j)_{j \in J}$ for program q. We can namely combine any two such finite control state partitions for separate programs into a single finite control state partition for both programs. The following result shows us that we can assume without loss of generality that if two programs have a finite control state partition, then they have the same finite control state partition.

Theorem 6.3.15. *Let p be a* While *program and $(P_i)_{i \in I}$ a control state partition for p, with finite I. Let q be a* While *program and $(Q_j)_{j \in J}$ a control state partition for q, with finite J.*

Then there exists a control state partition $(R_k)_{k \in K}$ with finite K that is a control state partition for both p and q.

Proof. Assume without loss of generality that $(P_i)_{i \in I}$ and $(Q_j)_{j \in J}$ partition $\mathcal{S}_X$ for a common X. If this is not the case, i.e. $(P_i)_{i \in I}$ partitions $\mathcal{S}_{X_1}$ and $(Q_j)_{j \in J}$ partitions $\mathcal{S}_{X_2}$ for $X_1 \neq X_2$, we simply take $X = X_1 \cup X_2$ and we take the extensions of $(P_i)_{i \in I}$ and $(Q_j)_{j \in J}$ to X.

Let $K = I \times J$. Clearly, since both I and J are finite, K is finite. For each $(i, j) \in K$, we define $R_{(i,j)} = P_i \cap Q_j$.

It remains to show that $(R_k)_{k \in K}$ satisfies the requirements of a control state partition for both p and q. Because both $(P_i)_{i \in I}$ and $(Q_j)_{j \in J}$ partition $\mathcal{S}_X$, so does $(R_k)_{k \in K}$. Also, since for each state $s \in \mathcal{S}_X$, we have that $P(s)$ determines the truth value of each $h \in \mathcal{H}^p$, and that $Q(s)$ determines the truth value of each $h \in \mathcal{H}^q$. Thus, since $R(s)$ determines both $P(s)$ and $Q(s)$, we get that $R(s)$ determines the truth value of each $h \in \mathcal{H}^p \cup \mathcal{H}^q$ for each $s \in \mathcal{S}_X$. By a similar argument, we get that $R(s)$ determines $g(s)$ for each $s \in \mathcal{S}_X$ and each $g \in \mathcal{G}^p \cup \mathcal{G}^q$. The fourth condition, finally, follows by similar reasoning. Let $S \subseteq \mathcal{S}_X$ be an arbitrary subset, and consider $R_{(i,j)}$ for arbitrary $(i, j) \in K$. Since we know $R_{(i,j)} \cap S = P_i \cap (Q_j \cap S)$, by $(Q_j \cap S) \subseteq \mathcal{S}_X$ and by the fact that for any $S' \subseteq \mathcal{S}_X$ such that $P_i \cap S' \neq \emptyset$ it is decidable to find a witness $s \in P_i \cap S'$, we know it is decidable to find a witness $s \in R_{(i,j)} \cap S$ if $R_{(i,j)} \cap S \neq \emptyset$. □

6.3.2.1 Computational Complexity

In Theorem 6.3.14 above, we showed that checking equivalence of *While* programs with finite control state partitions is decidable. We sketch a CO-NEXPTIME upper bound on the computational complexity of this reasoning problem.

Theorem 6.3.16. *For any two* While *programs p_1 and p_2 for which there exists a control state partition $(P_i)_{i \in I}$ with finitely many control states (i.e. I is finite), checking if p_1 and p_2 are equivalent can be done in* CO-NEXPTIME *in the size of the programs p_1 and p_2, i.e. its negation can be verified in $\mathcal{O}(|I|^4 \cdot 2^{|p_1|+|p_2|})$ time by a nondeterministic algorithm.*

Proof (sketch). We use the decision procedure from the proof of Theorem 6.3.14. We check whether (one of) a finite number of structures $\mathcal{L}$ can be extended to a model for the appropriate ABox and TBox. We know each such structure $\mathcal{L}$ has size $\mathcal{O}(|I|^2)$ and there are at most $\mathcal{O}(|I|^2)$ many such structures. We know that the size $|\mathcal{A}|$ of the considered ABox is $\mathcal{O}(1)$ and that the size $|\mathcal{T}|$ of the considered TBox is $\mathcal{O}(|p|^k)$ for some constant k, where $|p| = |p_1| + |p_2|$. Thus, the number of relevant extensions of any structure $\mathcal{L}$ are $\mathcal{O}(|I|^2 \cdot 2^{|p|})$ many, since at most $\mathcal{O}(|p|^k)$ many concepts occur in $\mathcal{A}$ and $\mathcal{T}$, and for each of the $\mathcal{O}(|I|^2)$ many objects in $\mathcal{L}$ one subset of the $\mathcal{O}(|p|^k)$ many concepts must be chosen. Thus, for each structure $\mathcal{L}$ it can be checked nondeterministically in $\mathcal{O}(|I|^2 \cdot 2^{|p|})$ time whether it can be extended to a model. This results in a $\mathcal{O}(|I|^4 \cdot 2^{|p|})$ upper time bound on the total nondeterministic procedure (checking non-equivalence). □

6.3.2.2 Some Programs With Finitely Many Control States

In order to illustrate the class of *While* programs that allow control state partitions of finite cardinality, as used above, we give an example of a subclass of this class of programs. Intuitively, programs in this class are characterized by the fact that there are only a fixed number of variables that affect the control flow of the program in any execution, and that for these variables only a fixed number of values matter. In other words, for any values of these variables above a certain threshold, the control flow of the program is the same. This results in a bounded number of different control flows that can occur in executions of the program (and thus that need to be taken into consideration). In order to characterize this class of programs, we make use of the notions of maximal condition free subprograms and maximal Boolean conditions.

Example 6.3.17. *Let p be a* While *program containing variables $Var(p) = X \subseteq \mathcal{X}$. Furthermore, assume there is some sequence of variables $(x_1, \ldots, x_k)$ for $\{x_1, \ldots, x_k\} = X_m \subseteq X$ and there is some sequence of limits for these variables $l_1, \ldots, l_k \in \mathbb{N}$, such that:*

- *for each $s, s' \in \mathcal{S}_X$ with $s(x_i) \geq l_i$ and $s'(x_i) \geq l_i$ for all $1 \leq i \leq k$, it holds that for each $h \in \mathcal{H}^p$ we have $\mathcal{B}^X(s, h) = \mathcal{B}^X(s', h)$;*

- *no variable $x \in X \backslash X_m$ occurs in any $h \in \mathcal{H}^p$; and*

- *for each $1 \leq i \leq k$ we have that (for any state) the value of x_i is monotonically increasing under application of each $g \in \mathcal{G}^p$.*

In other words, we consider programs p that have a fixed set of variables (i) such that only those variables occur in any $h \in \mathcal{H}^p$, (ii) whose values increase monotonically, and (iii) for which it holds that when their values exceed a certain threshold, the value of any $h \in \mathcal{H}^p$ stays the same.

Assuming that p satisfies these conditions, we define the following control state partition $(P_i)_{i \in I}$ for the program p:

$$f(l, n) = \begin{cases} l & \text{if } n \geq l \\ n & \text{otherwise} \end{cases}$$

$$I = \{(n_1, \ldots, n_k) \mid 0 \leq n_1 \leq l_1, \ldots, 0 \leq n_k \leq l_k\}$$

$$s \in P_{(f(s(x_1)), \ldots, f(s(x_k)))} \text{ for any } s \in \mathcal{S}_X$$

Clearly, since X is finite, we know X_m is finite, and therefore also I is finite. The argumentation why $(P_i)_{i \in I}$ is a control state partition is analogous to the reasoning in example 6.3.5.

Note that the program and the control state partition from Examples 6.3.3 and 6.3.5 fit these conditions.

In order to illustrate what kind of programs belong to the tractable subclass of programs specified in Example 6.3.17, we give two examples of applications for which there exist programs in this subclass. The following example shows how to use *While* programs to compute the value of (arbitrary) linear expressions.

Example 6.3.18. *Let $\varphi(x_1, \ldots, x_k) = c_1 x_1 + \cdots + c_k x_k + c$ be a linear expression on variables $x_1, \ldots, x_k$ and using constants $c_1, \ldots, c_k \in \mathbb{N}$. Then the following* While *program p_φ takes* *the values $n_1, \dots, n_k$ of variables $x_1, \dots, x_k$, respectively, as inputs, computes the value of $\varphi(n_1, \dots, n_k)$ and returns this as the value of variable z.*

```
$p_\varphi =$  $z := 0; x_{index} := 1; x_{done} := 0;$
      $x^1_{counter} := 0; \dots ; x^k_{counter} := 0;$
      while $(x_{done} < 1)$ do (
        if $(x_{index} = 1 \wedge x^1_{counter} \geq c_1)$ then $(x_{index} := x_{index} + 1)$ else
        $\dots;$
        if $(x_{index} = k \wedge x^k_{counter} \geq c_k)$ then $(x_{index} := x_{index} + 1)$ else
        if $(x_{index} = 1)$ then $(z := z + x_1; x^1_{counter} := x^1_{counter} + 1)$ else
        $\dots;$
        if $(x_{index} = k)$ then $(z := z + x_k; x^k_{counter} := x^k_{counter} + 1)$ else
        $x_{done} := x_{done} + 1);$
      $z := z + c$
```

In order to see that this example program p_φ fits the assumptions of Example 6.3.17, let $(x_{done}, x_{index}, x^1_{counter}, \dots, x^k_{counter})$ be the sequence of variables and $(1, k+1, c_1, \dots, c_k)$ be the sequence of limits corresponding to the variables, according to the assumptions.

The intuition behind the class of programs defined in Example 6.3.17 can be seen in the program in Example 6.3.18. Remember that this intuition is that only finitely many variables and finitely many values matter for the control flow of the program. The control flow of the program in Example 6.3.18 has a relatively fixed form, that depends on only a few variables. This control flow consists of a cascade of incrementing variable values: the program increments the variables $x^i_{counter}$ (up to values c_i, respectively), incrementing the variable x_{index} each time a variable $x^i_{counter}$ reaches the threshold. Similarly, it increments x_{done} (to the threshold value 1) when x_{index} reaches the threshold value $(k+1)$. If the values of these variables are higher than the thresholds, then the program breaks out of the loop. In other words, the control flow of the program depends only on the variable values below the thresholds.

The next example shows how to use *While* programs to compute the absolute difference of (arbitrary) values up to a (constant) maximum.

Example 6.3.19. *Let $c \in \mathbb{N}$ be an arbitrary constant, and let x_1, x_2 be input variables. The following example program p_{diff} computes the value of $\min\{|x_1 - x_2|, c\}$ and returns this as the value of variable z.*

```
$p_{diff} =$  $y := 0; z := 0;$
      if $(z < c \wedge (x_1 - x_2 = 0 \wedge x_2 - x_1 = 0))$ then $skip$ else
      while $(y = 0)$ do
        $(z := z + 1;$
        if $(z \geq c)$ do $y := y + 1$ else
        if $(z < c \wedge (x_1 - z = x_2 \vee x_2 - z = x_1))$ do $y := y + 1$ else $skip)$
```

In order to see that this example program p_{diff} fits the assumptions of Example 6.3.17, let (y, z) be the sequence of variables and $(1, c)$ be the sequence of limits corresponding to the variables, according to the assumptions.

Intuitively, the program from Example 6.3.19 has a bounded number of different control flows because the program increments variable z from 0 up to maximum value c, breaking out of the while loop when z equals $|x_1 - x_2|$. Concretely, this results in a maximum of $c + 1$ different control flows of the program.

For further discussion of how the characterization of Example 6.3.17 can be used practically, see Section 7.3.

Note that the class of programs defined in Example 6.3.17 is only one subclass for which deciding termination and equivalence is decidable. However, the class of programs for which there are finite control state partitions is more general. One suggestion for identifying additional classes of programs that allow finite control state partitions is to create a characterization that does not restrict the relevant values of certain variables to be only a constant number of values, but that divides the values that these variables can take into finitely many sets. A concrete example would be to divide the values of a certain variable into two sets: the even and the odd values. Additionally, in order to make good use of such partitionings of variable values, it might be useful to introduce additional built-in operators on values (e.g. an even and an odd operator that can be used in conditionals). Classifying such additional classes of programs that allow for finite control state partitions is a topic of further research.

6.3.3 Programs With Infinite Control State Partitions

We show that the general case of deciding equivalence of *While* programs with infinite control state partitions is undecidable. In order to show this, we give a reduction from the undecidable problem whether a Diophantine equation (an equation of the form $p(x_1, \ldots, x_n) = 0$ where p is a polynomial with integer coefficients) has a solution of natural numbers. This problem is central to Hilbert's Tenth Problem. For such a Diophantine equation, we construct a *While* program that computes the absolute value of $p(x_1, \ldots, x_n)$ for the input variables $x_1, \ldots, x_n$.

Theorem 6.3.20. *Deciding termination and equivalence of* While *programs with infinite control state partitions is undecidable in general.*

Proof. In both cases, we reduce from the undecidable problem whether a multivariate polynomial equation has a solution of natural numbers. Let $\varphi(x_1, \ldots, x_n) = c_1 x_1^{n_1^1} \ldots x_m^{n_m^1} + \cdots + c_j x_1^{n_1^j} \ldots x_m^{n_m^j} - c_{j+1} x_1^{n_1^{j+1}} \ldots x_m^{n_m^{j+1}} - \cdots - c_k x_1^{n_1^k} \ldots x_m^{n_m^k}$ be an arbitrary multivariate polynomial expression over the variables $x_1, \ldots, x_m$, where all $c_1, \ldots, c_k$ are integers, all n_j^i are natural numbers, and each $\pm^i$ is either $+$ or $-$.

For both reductions, we will use the *While* program p_φ from Figure 6.1 that computes the absolute value of $\varphi(x_1, \ldots, x_n)$ for input variables $x_1, \ldots, x_n$. It is straightforward to verify that it computes this value. It does so by looping through all expressions that are to be added, by using variable x_{index}. For each such expression, it calculates the resulting value in x_{subres} and adds this to either x_{res}^{pos} or x_{res}^{neg}, depending on whether it is part of the positive or negative part of the polynomial. In order to calculate the value of such an expression it loops through all the parts of the expression that are to be multiplied, by using variable x_{elem}. It keeps the subtotal in x_{subres}, uses x_{mcount} to perform the right number of multiplications (multiplying with the input values x_i), each of which is performed using variables $x_{multiplier}^1$ and $x_{multiplier}^2$. In the end, it computes the absolute value of $\varphi(x_1,\ldots,x_n)$ based on x_{res}^{pos} and x_{res}^{neg}, and puts this value in x_{res}. Afterwards, it sets all auxiliary variables it used (except x_{res}) to 0. Furthermore, it is straightforward to verify that this program always terminates.

Now, for the case of termination of *While* programs, consider the program

$$q = p_\varphi; (\textbf{while}\ p = 0\ \textbf{do}\ skip); skip$$

We have that q terminates on all inputs if and only if there is no solution of natural numbers for the multivariate polynomial equation $\varphi(x_1,\ldots,x_n) = 0$.

For the case of equivalence of *While* programs, consider the programs

$$r = p_\varphi; (\textbf{if}\ p > 0\ \textbf{then}\ (x_{res} := 1)\ \textbf{else}\ skip); skip$$

and

$$\begin{array}{rl} r' = & x_{aux} := 0;\ x_{elem} := 0;\ x_{index} := 0;\ x_{res} := 1;\ x_{subres} := 0;\ x_{mcount} := 0; \\ & x^1_{multiplier} := 0;\ x^2_{multiplier} := 0;\ x^{pos}_{res} := 0;\ x^{neg}_{res} := 0; skip \end{array}$$

We have that r and r' are equivalent if and only if there is no solution of natural numbers for the multivariate polynomial equation $\varphi(x_1,\ldots,x_n) = 0$.

Finally, we show that there exists a control state partition $(P_i)_{i\in I}$ of countably infinite size for all programs above (p_φ, q, r and r'). We define $I = \mathbb{N}^{7+k}$. A state $s \in \mathcal{S}_X$ is in a control state $P_{(i_1,\ldots,i_{7+n})}$ iff $s(x_{index}) = i_1$, $s(x_{aux}) = i_2$, $s(x^1_{multiplier}) = i_3$, $s(x_{elem}) = i_4$, $s(x^{pos}_{res}) = i_5$, $s(x^{neg}_{res}) = i_6$, $s(x_{mcount}) = i_7$, and $s(x_j) = i_{7+j}$ for all $1 \leq j \leq n$. It is straightforward to verify that $(P_i)_{i\in I}$ is a control state partition for the programs above. □

Theorem 6.3.21. *Deciding whether a* While *program p containing variables $X \subseteq \mathcal{X}$ terminates on a given input state $s \in \mathcal{S}_X$ is undecidable in general.*

Proof (sketch). This can be proven quite straightforwardly by reduction from the Halting Problem. We know that the problem whether a Turing machine halts on a given input tape is undecidable. Let M be an arbitrary Turing machine operating on the alphabet $\{0,1\}$. Consider the function $f : \mathbb{N} \to \mathbb{N}$ given by:

$$f(n) = \begin{cases} 1 & \text{if } M \text{ halts on the binary representation of } n \text{ as input tape} \\ undefined & \text{otherwise} \end{cases}$$

We get that f is a Turing-computable function. Therefore, we know there is a μ-recursive function that specifies this function f. Hence, by Proposition 3.3.2, we can express f as a *While* program p that operates on an input variable $x \in X \subseteq \mathcal{X}$. We then get that p terminates on a state $s \in \mathcal{S}_X$ with $s(x) = n$ if and only if M halts on the binary representation of n. □

Program p_φ computing the absolute value of a multivariate polynomial expression $c_1x_1^{n_1^1}\dots x_m^{n_m^1}+\dots+c_jx_1^{n_1^j}\dots x_m^{n_m^j}-c_{j+1}x_1^{n_1^{j+1}}\dots x_m^{n_m^{j+1}}-\dots-c_kx_1^{n_1^k}\dots x_m^{n_m^k}$, over the variables $x_1,\dots,x_m$, where all $c_1,\dots,c_k$ are integers, all n_j^i are natural numbers.

```
$p_\varphi =$  $x_{aux} := 0;\ x_{elem} := 0;\ x_{index} := 1;\ x_{res} := 0;$
     $x_{subres} := 0;\ x_{mcount} := 0;$
     $x^1_{multiplier} := 0;\ x^2_{multiplier} := 0;\ x^{pos}_{res} := 0;\ x^{neg}_{res} := 0;$
     while $x_{index} \leq k$ do (
       if $(x_{aux} > 0 \land x_{aux} \geq x^1_{multiplier})$ then
         $x_{aux} := 0$
       else if $(x_{aux} > 0 \land x_{aux} < x^1_{multiplier})$ then
         $x_{subres} := x_{subres} + x^2_{multiplier};$
         $x_{aux} := x_{aux} + 1$
       else if $(x_{aux} = 0 \land x_{elem} = 0)$ then
         $x_{subres} := c_{x_{index}};$
         $x_{elem} := x_{elem} + 1$
       else if $(x_{aux} = 0 \land x_{elem} > m)$ then
         if $(x_{index} \leq j)$ then
           $x^{pos}_{res} := x^{pos}_{res} + x_{subres}$
         else if $(x_{index} > j)$ then
           $x^{neg}_{res} := x^{neg}_{res} + x_{subres}$
         else $(skip)$;
         $x_{elem} := 0;$
         $x_{index} := x_{index} + 1$
       else if $(x_{aux} = 0 \land x_{elem} > 0 \land x_{elem} \leq m \land x_{mcount} = 0)$ then
         $x^1_{multiplier} := x_{x_{elem}};$
         $x_{mcount} := n^{x_{elem}}_{x_{index}}$
       else if $(x_{aux} = 0 \land x_{elem} > 0 \land x_{elem} \leq m \land x_{mcount} > 0)$ then
         $x^2_{multiplier} := x_{subres};$
         $x_{aux} := 1;$
         $x_{mcount} := x_{mcount} - 1;$
         if $(x_{mcount} = 0)$ then
           $x_{elem} := x_{elem} + 1;$
         else $skip$;
       else $skip$);
     if $(x^{pos}_{res} > x^{neg}_{res})$ then $(x_{res} := x^{pos}_{res} - x^{neg}_{res})$ else $(x_{res} := x^{neg}_{res} - x^{pos}_{res})$;
     $x_{aux} := 0;\ x_{elem} := 0;\ x_{index} := 0;\ x_{subres} := 0;\ x_{mcount} := 0;$
     $x^1_{multiplier} := 0;\ x^2_{multiplier} := 0;\ x^{pos}_{res} := 0;\ x^{neg}_{res} := 0;\ skip$
```

In the above formulation of p_φ, we let $(x_{subres} := c_{x_{index}})$ abbreviate (**if** $x_{index} = 1$ **then** $x_{subres} := c_1$ **else** $\dots$ **else if** $x_{index} = k$ **then** $x_{subres} := c_k$ **else** $skip$) (similarly for other expressions containing variables x_{index} or x_{elem} as indices).

Figure 6.1: *While* program to compute the absolute value of a multivariate polynomial expression.

6.3.4 Alternative Method Without Using $\mathcal{ALC}(\mathcal{D})$

In order to get a better understanding of why using our encoding into $\mathcal{ALC}(\mathcal{D})$ is beneficial, we will transfer the decidability result we obtained above to a setting in which no logic nor any model-theoretic semantics is used. Of course, it is possible to state this result in such a setting. However, we will argue that this has certain disadvantages compared to the method using formal logic. We will firstly sketch a procedure to decide equivalence of *While* programs allowing finite control state partitions that does not use any of the logical machinery used previously in this thesis, i.e., we reprove Theorem 6.3.14:

Proof (sketch). Let p and p' be two arbitrary *While* programs using variables $X \subseteq \mathcal{X}$ such that $(P_i)_{i \in I}$ is a control state partition for both of them and let $|I| = k$ for some $k \in \mathbb{N}$. For the sake of simplicity, assume that p and p' are both (uniformly) terminating. Let $s_1, \ldots, s_k$ be representative states for the control state $P_1, \ldots, P_k$, respectively. Then, let $d_1, \ldots, d_k$ be the finite derivations obtained from executing p on $s_1, \ldots, s_k$, respectively, and let $d'_1, \ldots, d'_k$ be the finite derivations obtained from executing p' on $s_1, \ldots, s_k$. Now, for each $1 \leq i \leq k$, let e_i be the derivation structure (see Definition 6.3.6) of d_i and let f_i be the derivation structure of d'_i.

Then, using these derivation structures e_i and f_i, we instantiate a number of Boolean problems over atomic linear arithmetic constraints. For each $1 \leq i, j \leq k$, we create such a problem instance. Fix e_i and f_j, and let $p_1^{e_i}, \ldots, p_n^{e_i}$ be the programs occurring in e_i, and $p_1^{f_j}, \ldots, p_m^{f_j}$ be the programs occurring in f_j. In the problem instance, we introduce a variable $z(y)$ for each element y in $X \times \{p_1^{e_i}, \ldots, p_n^{e_i}, p_1^{f_j}, \ldots, p_m^{f_j}\}$. We add the following constraints on these variables. For each $x \in X$ we require $z(x, p_1^{e_i}) = z(x, p_1^{f_j})$. Intuitively, this ensures that the initial states of both derivations are the same. We also require $\bigvee_{x \in X}(z(x, p_n^{e_i}) \neq z(x, p_m^{f_j}))$, which intuitively ensures that the final states are different on at least one variable. Then, we add constraints to enforce that the variables behave according to the derivation. For instance, if $p_k^{e_i} = x := x'; q$ and $p_{k+1}^{e_i} = q$, we require that $z(x, p_{k+1}^{e_i}) = z(x', p_k^{e_i})$. This involves more than just variable assignments. For instance, if $p_k^{e_i} = (\textbf{if } b \textbf{ then } q_1 \textbf{ else } q_2); q$ and $p_{k+1}^{e_i} = q_1; q$, then we need to enforce (the constraint corresponding to) b on the variables $z(x, p_k^{e_i})$ involved.

Using the Boolean arithmetic problems we constructed, we are able to determine whether p and p' are equivalent. Namely, p and p' are equivalent if and only if none of the constructed Boolean problems are satisfiable. The one direction of this equivalence can be shown by constructing a counterexample derivation witnessing the non-equivalence of p and p' by using a satisfying instance of one of the Boolean problems. The other direction can be shown by constructing a satisfying instantiation for one of the Boolean problems from a counterexample derivation witnessing that p and p' are not equivalent. To show this latter implication, Lemma 6.3.2 is crucial. □

Note that in the algorithm given above to decide equivalence of *While* programs, we essentially reduce the task of deciding equivalence to a number of Boolean problems, without referring to any particular logic language. Conceptually, what happens in this algorithm is exactly the same as what happens in the method using $\mathcal{ALC}(\mathcal{D})$ (i.e. in the proof of Theorem 6.3.14), except for the fact that it is stated in different terms. The specification of the Boolean problems can be given in the $\mathcal{ALC}(\mathcal{D})$ language, and solutions to the Boolean problems correspond to models of such $\mathcal{ALC}(\mathcal{D})$ expressions. In other words, the above alternative procedure is essentially a logical method in disguise.

However, explicitly using formal logic has a number of advantages over using a disguised logical method as above. For instance, stating the problem in the syntax of a logical language, together with the fact that the algorithms used to find solutions are based on the semantics of this logic, allows us to directly reuse results, methods and algorithms for similar settings. This is also illustrated, for instance, by the fact that the same methods are used to show decidability of reasoning for the fragments identified in Sections 6.1 and 6.2. Similarly, the framework developed in this thesis can straightforwardly be extended to additional programming languages, because in all cases the same underlying logic is used.

Also, since using formal logic allows us to use the full expressivity of the logic language, the logical approach is easily extendable to other semantic properties of programs (besides termination and equivalence). Algorithms for such additional reasoning problems can then use (much of) the same underlying model-theoretic techniques. Such a reuse of methods is not so straightforward with the alternative, logic-free approach described above.

6.3.5 More Fragments Allowing Decidable Reasoning

Partitioning the state space into a finite number of partitions, as done above, leads to a fragment of *While* programs for which reasoning tasks such as termination and equivalence are decidable. In fact, for all programs for which there is such a finite control state partition, these reasoning problems are decidable. This does not imply the converse, however. There might be a class of programs that can not be characterized by the existence of finite control state partitions, but for which these reasoning problems are decidable. We suggest a few directions of further research for identifying additional classes of programs for which deciding termination and equivalence is possible.

It might be possible that some part of the state space is never reachable in certain parts of the program, in any execution. For instance, for any execution of a program of the form $(x := 0; y := 1; p)$ it is impossible to reach any state s with $s(x) = 2$ at the subprogram $(y := 1; p)$. Such unreachable parts of the state space do not have to be taken into account when capturing the different possible control flows that can occur in executions of programs. Using the notion of control state partitions, ignoring parts of the state space for particular subprograms is not possible. A refinement of the notion of control state partitions could result in a larger class of programs for which the reasoning tasks of termination and equivalence are decidable.

Another suggestion would be to find a class of programs for which the different control flows that can occur in executions are not bounded in number, but are of a particular regular shape. Consider for instance the following program p_{even} that sets variable z to 1 if input variable x is even, and sets z to 0 otherwise.

```
p_even =  z := 0;
          while (x > 0) do
            ((if (z = 0) do z := 1 else z := 0);
            x := x − 1)
```

Even though there are infinitely many different executions of this program, each with a different derivation structure, they are structurally similar (namely the while loop is executed exactly c times, for c the initial value of x). Such programs do not allow a finite control state partition. Nevertheless, the structure in the different possible executions of such programs might be exploited, possibly resulting in an additional class of programs for which the reasoning tasks of termination and equivalence are decidable.

6.4 Structuring Variable States for *Goto*

We will define the notion of control states (and control state partitions) for *Goto* programs, much like we defined this notion for *While* programs. We do this with the goal of identifying a fragment of the programming language that allows for termination and equivalence of programs to be decided algorithmically.

We could use an approach to identifying this fragment of the *Goto* programming language, for which deciding termination and equivalence of programs is decidable, that is completely analogous to the method we used for identifying such a fragment for *While*. This would involve, firstly, showing that control states entirely determine the (program) structure of *Goto* derivations. Then, like in the case for *While*, we would have to define the notion of derivation frames for *Goto* programs, and show that any model of the encoding of a *Goto* program into $\mathcal{ALC}(\mathcal{D})$ contains the derivation frame (up to isomorphism) of the derivation starting with the state corresponding to the object witnessing the satisfiability of the concept related to the *Goto* program. This could be done quite straightforwardly (yet tediously), analogously to the method we worked out for *While* programs.

However, for the sake of brevity, and in order not to be too much repetitive (after all, only the details would be different from the case for *While*), we will use a different approach. We show that *While* programs can be translated to equivalent *Goto* programs and vice versa. Also, we will show that these translations only results in (at most) a linear grow in the size of control state partitions. This will allow us to extend the (decidability and undecidability) results we obtained for *While* to the setting of *Goto* as well. Furthermore, this alternative approach illustrates the possibility of easily extending the results obtained for the programming language *While* to different imperative languages, by simply specifying translations that preserve equivalence and finiteness of control state partitions.

6.4.1 Control State Partitions

In order to define the notion of control state partitions for *Goto* programs κ, we must firstly define the notions of maximal condition free subprograms and maximal Boolean conditions, like we did for the programming language *While*.

Definition 6.4.1 (Maximal condition free subprograms)**.** *Let κ be a* Goto *program of size l We assume w.l.o.g. that $\kappa(l) = return$. We define the set $\mathcal{G}^\kappa$ of maximal condition free subprograms as the set of all $sub(\kappa, n_1, n_2)$ such that:*

- $1 \leq n_1 \leq n_2 \leq l$;
- $n_1 \in N^\kappa_{begin}$;

- *for all $n_1 \leq i \leq n_2$, it holds that $\kappa(i) = (x := e)$;*
- *for all $n_1 < i \leq n_2$, it holds that $i \notin N_{begin}$*
- *either (i) $n_2 + 1 \in N^{\kappa}_{begin}$, or (ii) $\kappa(n_2 + 1) =$ **if** b **goto** m_1 **else** m_2, or (iii) $n_2 = l$.*

where the set N^{κ}_{begin} of begin points of condition-free subprograms is defined as those $1 \leq n \leq l$ such that either

- *$n = 1$ or $n = l$; or*
- *there exists some $1 \leq i \leq l$ such that $\kappa(i) =$ **if** b **goto** m_1 **else** m_2 and $n \in \{m_1, m_2\}$.*

We will refer to the set N^{κ}_{begin} below. We will also use the following abbreviation. For any $g \in \mathcal{G}^{\kappa}$ of size l_g taken from κ starting at line $j \in N^{\kappa}_{begin}$ (i.e. for all $1 \leq i \leq l_g$ we have $g(i) = \kappa(j + i - 1)$) we let $\mathcal{G}^{\kappa}_{While}(j)$ denote the *While* program $g(1); \ldots ; g(l_g)$. Furthermore, in this case we let $\mathcal{G}^{\kappa}_{size}(j)$ denote l_g.

Definition 6.4.2 (Maximal Boolean conditions)**.** *For any given* Goto *program κ of size l, we define the set $\mathcal{H}^{\kappa}$ of all maximal Boolean conditions occurring in κ as $\mathcal{H}^{\kappa} = \bigcup_{1 \leq i \leq l} \mathcal{H}^{\kappa,i}$, where:*

$$\mathcal{H}^{\kappa,i} = \begin{cases} \{b\} & \text{if } \kappa(i) = \textbf{\textit{if }} b \textbf{\textit{ goto }} m_1 \textbf{\textit{ else }} m_2 \\ \emptyset & \text{otherwise} \end{cases}$$

Note that for any *Goto* program κ, both sets $\mathcal{G}^{\kappa}$ and $\mathcal{H}^{\kappa}$ can be determined purely syntactically.

Clearly, for any *Goto* program κ we have that all $g \in \mathcal{G}^{\kappa}$ are terminating (on all input states). In order to fix notation, given any *Goto* program λ containing variables $X \subseteq \mathcal{X}$ and any input state $s \in \mathcal{S}_X$, we denote the unique state $t \in \mathcal{S}_X$ such that for λ we have $(1, s) \Rightarrow^{*}_{\lambda} t$ with $\lambda(s)$.

In order to illustrate these notions, we will give the sets $\mathcal{G}^{\kappa}$ and $\mathcal{H}^{\kappa}$ for the *Goto* program κ in our running example.

Example 6.4.3. *Remember from Example 3.1.4 that*

$$\kappa = \begin{array}{rl} 1: & z := 0 \\ 2: & w := 0 \\ 3: & \textbf{\textit{if}}\,(w < x \wedge w < 3)\ \textbf{\textit{goto}}\ 4\ \textbf{\textit{else}}\ 7 \\ 4: & z := z + y \\ 5: & w := w + 1 \\ 6: & \textbf{\textit{if}}\ \top\ \textbf{\textit{goto}}\ 3\ \textbf{\textit{else}}\ 3 \\ 7: & return \end{array}$$

We then have that

$$\mathcal{G}^{\kappa} = \left\{ \begin{array}{rl} 1: & z := 0 \\ 2: & w := 0 \\ 3: & return \end{array}, \begin{array}{rl} 1: & z := z + y \\ 2: & w := w + 1 \\ 3: & return \end{array} \right\}$$

$$\mathcal{H}^{\kappa} = \{(w < x \wedge w < 3), \top\}$$

Similarly to the case of *While*, we define the notion of control states and control state partitions.

Definition 6.4.4 (Control states)**.** *For any* Goto *program κ containing variables $X \subseteq \mathcal{X}$, we define a* control state partition for κ *to be a family $(P_i)_{i\in I}$ of subsets of states (called* control states*) $P_i \subseteq \mathcal{S}_X$ such that:*

1. *$\cup_{i\in I} P_i = \mathcal{S}_X$ and $P_i \cap P_j = \emptyset$ for all $i, j \in I$ such that $i \neq j$;*

2. *for all $i \in I$, all $s, t \in P_i$ and all $h \in \mathcal{H}^\kappa$ we have that $\mathcal{B}^X(s, h) = \mathcal{B}^X(t, h)$; and*

3. *for all $i, j \in I$, all $s, t \in P_i$ and all $g \in \mathcal{G}^\kappa$ we have that $g(s) \in P_j$ iff $g(t) \in P_j$; and*

4. *for all $i \in I$, and for each subset $S \subseteq \mathcal{S}_X$, if $P_i \cap S \neq \emptyset$ then some state $s \in P_i \cap S$ must be effectively constructible*

For any state $s \in \mathcal{S}_{X'}$ for $X' \supset X$, we will often implicitly use $s' \in \mathcal{S}_X$ such that for all $x \in X$ we have $s(x) = s'(x)$, when talking about membership of s in control states. Note that this definition of control states for *Goto* programs is completely analogous to the definition of control states for *While* programs.

Example 6.4.5. *For our running example program κ, we can use the control state partition $(P_i)_{i\in I}$ from Example 6.3.5.*

$$I = \{(n, m) \mid 0 \leq n, m \leq 3\}$$

$$f(n) = \begin{cases} 3 & \textit{if } n \geq 3 \\ n & \textit{otherwise} \end{cases}$$

$$s \in P_{(f(s(x)), f(s(w)))} \textit{ for any } s \in \mathcal{S}_{\{w,x,y,z\}}$$

The reasoning why this is indeed a control state partition for κ is completely analogous to the reasoning in Example 6.3.5.

6.4.2 Translating *Goto* to *While*

We firstly define the translations from arbitrary *Goto* programs to *While* programs, and from arbitrary *While* programs to *Goto* programs. Afterwards, we show that these translations are equivalence-preserving, and that the translations only introduce a linear increase in the size of control state partitions. We begin with the case of translating from *Goto* to *While*.

Definition 6.4.6 (Translation)**.** *Let κ be an arbitrary* Goto *program of size l. We define the* translation of κ in *While to be the* While *program p that is of the form*

```
p =  $x_{line} := 1$;
     while $(x_{line} \leq l)$ do
       (if $(x_{line} = i_1)$ then $\sigma(\kappa, i_1)$ else
       if $(x_{line} = i_2)$ then $\sigma(\kappa, i_2)$ else
       ...
       if $(x_{line} = i_m)$ then $\sigma(\kappa, i_m)$ else
       $skip$);
     $skip$
```

*where $\{i_1, \ldots, i_m\} = N^{\kappa}_{begin} \cup \{1 \leq i \leq l \mid \kappa(i) =$ **if** b **goto** m_1 **else** $m_2\}$, where x_{line} is a fresh variable, and where for all $1 \leq j \leq m$ we define $\sigma(\kappa, i_j)$ as*

$$\sigma(\kappa, i) = \begin{cases} \mathcal{G}^{\kappa}_{While}(i_j); x_{line} := x_{line} + \mathcal{G}^{\kappa}_{size}(i_j) & \textit{if } i_j \in N^{\kappa}_{begin} \textit{ and } i_j \neq l \\ x_{line} := x_{line} + 1 & \textit{if } i_j = l \\ \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ x_{line} := m_1\ \textbf{\textit{else}}\ x_{line} := m_2 & \textit{if } \kappa(i_j) = \textbf{\textit{if}}\ b\ \textbf{goto}\ m_1\ \textbf{\textit{else}}\ m_2 \end{cases}$$

Clearly, for any *Goto* program κ we have that the size of its translation p into *While* is linear in the size of κ.

Theorem 6.4.7. *For any* Goto *program κ on variables $X \subseteq \mathcal{X}$ and its translation p in* While*, and for any state $s \in \mathcal{S}_{X'}$, for $X' = X \cup \{x_{line}\}$, we have that $(1, s) \Rightarrow^*_{\kappa} s'$ if and only if $(p, s) \Rightarrow^* s''$, such that $s'(x) = s''(x)$ for all $x \in X$.*

Proof. Let $\{i_1, \ldots, i_m\}$ be as defined for the translation of κ. For each $i_j \in \{i_1, \ldots, i_m\}$, we define an auxiliary program p_{i_m} as $\sigma(\kappa, i_m); p$. In this proof, we will use the fact that derivations for *While* and *Goto* programs are deterministic. Clearly, we have $(1, s) \Rightarrow^*_{\kappa} (1, s)$, and we also have $(p, s) \Rightarrow^* (p_1, s[x_{line} \mapsto 1])$.

We show that for each $1 \leq n, n' \leq m$, and each $t, t' \in \mathcal{S}_{X'}$ we have that $(i_n, t) \Rightarrow^*_{\kappa} (i_{n'}, t')$ implies $(p_{i_n}, t[x_{line} \mapsto i_n]) \Rightarrow^* (p_{i_{n'}}, t'[x_{line} \mapsto i_{n'}])$, by induction on the length of the $\Rightarrow_{\kappa}$ derivation. The base case, where the length is 0, and thus $i_n = i_{n'}$ and $t = t'$ is trivial.

Consider the inductive case where $i_n \in N^{\kappa}_{begin}$. Then we have that $(i_n, t) \Rightarrow^*_{\kappa} (i_n + \mathcal{G}^{\kappa}_{size}(i_n), t'') \Rightarrow^*_{\kappa} (i_{n'}, t')$. It is easy to verify, by (induction on) the structure of p_{i_n} that then $(p_{i_n}, t[x_{line} \mapsto i_n]) \Rightarrow^* (p_{i_n + \mathcal{G}^{\kappa}_{size}(i_n)}, t''[x_{line} \mapsto i_n + \mathcal{G}^{\kappa}_{size}(i_n)])$. Since $i_n + \mathcal{G}^{\kappa}_{size}(i_n) \in \{i_1, \ldots, i_m\}$, by the induction hypothesis we get that $(p_{i_n + \mathcal{G}^{\kappa}_{size}(i_n)}, t''[x_{line} \mapsto i_n + \mathcal{G}^{\kappa}_{size}(i_n)]) \Rightarrow^* (p_{i_{n'}}, t'[x_{line} \mapsto i_{n'}])$, from which the result follows.

The inductive case where $\kappa(i_n) =$ (**if** b **goto** m_1 **else** m_2) is also analogous. Assume w.l.o.g. that $\mathcal{B}^{X'}(t, b) = \top$. We then have that $(i_n, t) \Rightarrow_{\kappa} (m_1, t) \Rightarrow^* (i_{n'}, t')$. By the structure of p_{i_n} we then get $(p_{i_n}, t[x_{line} \mapsto i_n]) \Rightarrow^* (p_{m_1}, t[x_{line} \mapsto m_1])$. Since we know that $m_1 \in \{i_1, \ldots, i_m\}$, by the induction hypothesis we get that $(p_{m_1}, t[x_{line} \mapsto m_1]) \Rightarrow^* (p_{i_{n'}}, t'[x_{line} \mapsto i_{n'}])$, from which the result follows.

Now consider an arbitrary derivation $(1, s) \Rightarrow^*_{\kappa} s'$. Without loss of generality, we assume that $\kappa(l) = return$. We then know that $(1, s) \Rightarrow^*_{\kappa} (l, s') \Rightarrow_{\kappa} s'$. By the above results, we know that $(p_1, s) \Rightarrow^* (p_1, s[x_{line} \mapsto 1]) \Rightarrow^* (p_l, s'[x_{line} \mapsto l+1])$. By the structure of p_l we get that $(p_l, s'[x_{line} \mapsto l+1]) \Rightarrow^* s'[x_{line} \mapsto l+1]$. Thus the result follows. □

Theorem 6.4.8. *For any* Goto *program κ on variables $X \subseteq \mathcal{X}$ and any control state partition $(P_i)_{i \in I}$ for κ, there is a control state partition $(Q_j)_{j \in J}$ for its translation p into* While *on $X' = X \cup \{x_{line}\}$ whose size is linear in the size of $(P_i)_{i \in I}$ and the size of κ.*

Proof. We let $J = \{(i,n) \mid 1 \leq n \leq l+1, i \in I\}$. Then we define the control state partition $(Q_j)_{j \in J}$ as follows. For each $s \in \mathcal{S}_{X'}$ and for $1 \leq n \leq l$, we let $s \in Q_{(i,n)}$ iff $s \in P_i$ and $s(x_{line}) = n$. For each $s \in \mathcal{S}_{X'}$ and for $n = l+1$, we let $s \in Q_{(i,n)}$ iff $s \in P_i$ and $s(x_{line}) > l$.

We know $\mathcal{H}^p = \mathcal{H}^\kappa$. Also, by construction of p, we know $\mathcal{G}^p = \mathcal{G}^\kappa$. It is straightforward to verify that $(Q_j)_{j \in J}$ is a control state partition for p and that the size of $(Q_j)_{j \in J}$ is $(l+1) \cdot |I|$. □

6.4.3 Translating *While* to *Goto*

We continue with translating from *While* to *Goto*, and showing that this translation is equivalence preserving. Furthermore, we show that control state partitions for any *While* program can be used as a control state partition for its translation as well.

In the following, for the programming language *Goto*, we consider $skip$ to be an abbreviation for a statement of the form $x := x$, for some $x \in X$.

Definition 6.4.9 (Translation). *Let p be an arbitrary* While *program. We define the* translation of p in *Goto to be the program κ, where the partial* Goto *program λ of size l is defined inductively, and where κ of size $l+1$ is such that for all $1 \leq i \leq l$ we have $\kappa(i) = \lambda(i)$ and $\kappa(l+1) = return$.*

- *If $p = skip$, then we let λ be of length 1 and $\lambda(1) = skip$ (remember that skip abbreviates $x := x$ for some x).*
- *If $p = (x := e)$, then we let λ be of length 1 and we let $\lambda(1) = (x := e)$.*
- *If $p = p_1; p_2$, we let λ of length l_1 be the (partial) translation of p_1, and λ of length l_2 be the (partial) translation of p_2. We let λ be of length $l_1 + l_2$ and:*
  - *for $1 \leq i \leq l_1$, we let $\lambda(i) = \lambda_1(i)$; and*
  - *for $1 \leq i \leq l_2$, we let $\lambda(l_1 + i) = \lambda_2(i)$.*
- *If $p =$ **if** b **then** p_1 **else** p_2, then we let λ_1 of length l_1 and λ_2 of length l_2 be the (partial) translations of p_1 and p_2, respectively. We let κ be of length $l_1 + l_2 + 3$, and:*
  - *$\lambda(1) =$ **if** b **goto** 2 **else** $3 + l_1$;*
  - *for $1 \leq i \leq l_1$, we let $\lambda(i+1) = \lambda_1(i)$;*
  - *$\lambda(l_1 + 2) =$ **if** $\top$ **goto** $l_1 + l_2 + 3$ **else** 1;*
  - *for $1 \leq i \leq l_2$, we let $\lambda(i + l_1 + 2) = \lambda_2(i)$; and*
  - *$\lambda(l_1 + l_2 + 3) = skip$.*

- *If $p =$ **while** p **do** p', then we let λ' of length l' be the (partial) translation of p'. We let λ be of length $l' + 3$, and:*
  - $\lambda(1) =$ ***if*** $\neg b$ ***goto*** $l' + 3$ ***else*** 2*;*
  - *for $1 \leq i \leq l'$, we let $\lambda(i + 1) = \lambda'(i)$;*
  - $\lambda(l' + 2) =$ ***if*** $\top$ ***goto*** 1 ***else*** 1*; and*
  - $\lambda(l' + 3) = skip$.

Theorem 6.4.10. *For any* While *program p on variables $X \subseteq \mathcal{X}$ and its translation κ into* Goto*, and for any state $s \in \mathcal{S}_X$, we have that $(p, s) \Rightarrow^* s'$ if and only if $(1, s) \Rightarrow^*_\kappa s'$.*

Proof. Assume without loss of generality that each derivation starting at p ends with $(skip, s')$, for some $s' \in \mathcal{S}_X$. Notice that for each *While* program p' and its translation κ' of length l', we have that $\kappa'(l') = skip$. We will make use of the fact that *While* and *Goto* derivations are deterministic. We prove by induction on the structure of *While* programs p that for their translation κ of length l, and for all states $s, s' \in \mathcal{S}_X$ we have $(p, s) \Rightarrow^* (skip, s')$ implies $(1, s) \Rightarrow^*_\kappa (l, s')$. The base case for $p = skip$ is trivial. The other base case for $p = (x := e; skip)$ is straightforward. We have $(p, s) \Rightarrow (skip, s[x \mapsto \mathcal{A}^X(s, e)])$, and clearly we also have $(1, s) \Rightarrow_\kappa (2, s[x \mapsto \mathcal{A}^X(s, e)])$.

The inductive case where $p = p_1; p_2$ follows immediately from the induction hypothesis and the fact that the translations of p_1 and p_2 are sequentially contained in the translation of p.

For the inductive case where $p =$ **if** b **then** p_1 **else** p_2, we let κ_1 of length l_1 and κ_2 of length l_2 be the translations of p_1 and p_2, respectively. Assume w.l.o.g. that $\mathcal{B}^X(s, b) = \top$. Then we have $(p, s) \Rightarrow (p_1, s) \Rightarrow^* (skip, s')$. By the structure of κ, we get that $(1, s) \Rightarrow_\kappa (2, s)$. By the induction hypothesis, since $(p_1, s) \Rightarrow^* (skip, s')$, and by the fact that κ_1 is contained in κ starting at $\kappa(2)$, we know that $(2, s) \Rightarrow^*_\kappa (l_1 + 1, s')$. By the structure of κ, then, we know that $(l_1 + 1, s') \Rightarrow^*_\kappa (l_1 + l_2 + 3, s')$. Thus, $(1, s) \Rightarrow^*_\kappa (l, s')$.

For the inductive case, where $p =$ **while** b **do** p', we let κ' of length l' be the translation of p'. Assume $(p, s) \Rightarrow^* (skip, s')$. We know then that $(p, s) = (p, s_1) \Rightarrow^* (p, s_2) \Rightarrow^* \cdots \Rightarrow^* (p, s_n) \Rightarrow^* (skip, s')$, such that all occurrences of p in this derivation are in a pair (p, s_i) for some $1 \leq i \leq n$. We show by induction on the number of times p occurs in this derivation that $(1, s) \Rightarrow^*_\kappa (l, s')$. In the base case, we know $i = 1$. Thus $\mathcal{B}^X(b, s) = \bot$, and we get $(p, s) \Rightarrow (skip, s)$. By the structure of κ, we directly get $(1, s) \Rightarrow^*_\kappa (l, s)$. In the inductive case for $i > 1$, we know $(p, s) \Rightarrow^* (p, s_2) \Rightarrow^* (skip, s')$. Thus $\mathcal{B}^X(b, s) = \top$, and thus $(p, s) \Rightarrow^* (p'; p, s)$. By the structure of κ, we get $(1, s) \Rightarrow^*_\kappa (2, s)$. Since $(p'; p, s) \Rightarrow^* (p, s_2)$, we know $(p', s) \Rightarrow^* (skip, s_2)$. Then by the induction hypothesis of our initial induction, and since κ' is contained in κ starting at $\kappa(2)$, we know $(2, s) \Rightarrow^*_\kappa (l' + 1, s_2)$. We also know $(l' + 1, s_2) \Rightarrow_\kappa (1, s_2)$ By the induction hypothesis, since $(p, s_2) \Rightarrow^* (skip, s')$, we know $(1, s_2) \Rightarrow^*_\kappa (l, s')$. Putting this all together, we get $(1, s) \Rightarrow^*_\kappa (l, s')$.

Finally, since we know $\kappa(l) = return$, we get $(l, s') \Rightarrow_\kappa s'$, and the result follows. □

Theorem 6.4.11. *For any* While *program p on variables $X \subseteq \mathcal{X}$ we have that any control state partition $(P_i)_{i \in I}$ for p is also a control state partition for its translation κ into* Goto*.*

Proof. We have $\mathcal{G}^\kappa = \mathcal{G}^p$. We also have that $\mathcal{H}^\kappa \subseteq \mathcal{H}^p \cup \{\neg h \mid h \in \mathcal{H}^p\} \cup \{\top\}$. From this and the fact that $(P_i)_{i\in I}$ is a control state partition for p, it follows that $(P_i)_{i\in I}$ is a control state partition for κ. □

6.4.4 Programs with Finite Control State Partitions

Using the above results, we can straightforwardly extend the decidability results we got for *While* programs that allow finitely many control states, to similar programs of the programming language *Goto*. In other words, we show that for such *Goto* programs termination and equivalence are decidable.

Theorem 6.4.12. *For any* Goto *program κ for which there exists a control state partition $(P_i)_{i\in I}$ with finitely many control states (i.e. I is finite), it is decidable to check whether κ terminates on all inputs.*

Proof. Let κ be an arbitrary program, and $(P_i)_{i\in I}$ a control state partition for κ for finite I. Now let p be the translation of κ into *While*. By Theorem 6.4.7, we know that κ terminates on all inputs if and only if p terminates on all inputs. By Theorem 6.4.8, and by the fact that I is finite, we know there exists a finite control state partition $(Q_j)_{j\in J}$ for p. Finally, by Theorem 6.3.9 we know that checking termination of p is decidable. □

Theorem 6.4.13. *For any two* Goto *programs κ_1 and κ_2 for which there exists a control state partition $(P_i)_{i\in I}$ with finitely many control states (i.e. I is finite), it is decidable to check whether κ_1 and κ_2 are equivalent.*

Proof. Assume without loss of generality that κ_1 and κ_2 both contain variables $X \subseteq \mathcal{X}$. Let p_1 and p_2 be the translations of κ_1 and κ_2 into *While*, respectively. By Theorem 6.4.7, we know that κ_1 and κ_2 are equivalent if and only if $(p_1; x_{line} := 0)$ and $(p_2; x_{line} := 0)$ are equivalent. By Theorem 6.4.8, and by the fact that I is finite, we know there exists a finite control state partition $(Q_j)_{j\in J}$ for p_1 and p_2. Clearly, $(Q_j)_{j\in J}$ is also a control state partition for $(p_1; x_{line} := 0)$ and $(p_2; x_{line} := 0)$ (when extended to states in $\mathcal{S}_{X\cup\{x_{line}\}}$). Then, by Theorem 6.3.14, we know checking equivalence of $(p_1; x_{line} := 0)$ and $(p_2; x_{line} := 0)$ is decidable. Thus so is checking equivalence of κ_1 and κ_2. □

In fact, we can also extend the upper bound on the complexity of checking equivalence of *While* programs with finite control state partitions we got in Theorem 6.3.16 to the case of *Goto*.

Theorem 6.4.14. *For any two* Goto *programs κ_1 and κ_2 for which there exists a control state partition $(P_i)_{i\in I}$ with finitely many control states (i.e. I is finite), checking if κ_1 and κ_2 are equivalent can be done in* NEXPTIME *in the size of the programs κ_1 and κ_2, i.e. it can be done in $\mathcal{O}(|I|^m \cdot 2^{|\kappa_1|+|\kappa_2|})$ time, for some constant m, by a nondeterministic algorithm.*

Proof (sketch). By Definition 6.4.6 and by Theorem 6.4.7, we know we can translate κ_1 and κ_2 into equivalent *While* programs p_1 and p_2, respectively. Then, by Theorems 6.4.8 and 6.3.15, we know there exists a control state partition for p_1 and p_2 of size polynomial in $|I|$. The result then follows immediately from Theorem 6.3.16. □

6.4.5 Programs with Infinite Control State Partitions

In order to finish extending the (un)decidability results obtained for *While* to *Goto*, we show that deciding termination and equivalence of *Goto* programs with infinite control state partitions, in general, is undecidable. We do so by using the translations and their properties defined above.

Theorem 6.4.15. *Deciding termination and equivalence of* Goto *programs with infinite control state partitions is undecidable in general.*

Proof. In both cases, we reduce from the undecidable problem whether a multivariate polynomial equation has a solution of natural numbers. Let $\varphi(x_1,\dots,x_n) = c_1 x_1^{n_1^1}\dots x_m^{n_m^1} + \dots + c_j x_1^{n_1^j}\dots x_m^{n_m^j} - c_{j+1} x_1^{n_1^{j+1}}\dots x_m^{n_m^{j+1}} - \dots - c_k x_1^{n_1^k}\dots x_m^{n_m^k}$ be an arbitrary multivariate polynomial expression over the variables $x_1,\dots,x_m$, where all $c_1,\dots,c_k$ are integers, all n_j^i are natural numbers, and each $\pm^i$ is either $+$ or $-$.

The proof is along the same general lines as the proof of Theorem 6.3.20. For both reductions, we will use the *While* program p_φ from Figure 6.1 that computes the absolute value of $\varphi(x_1,\dots,x_n)$ for input variables $x_1,\dots,x_n$. For the case of termination of *While* programs, consider the *While* program

$$q = p_\varphi; (\textbf{while}\; p = 0 \;\textbf{do}\; skip); skip$$

We have that q terminates on all inputs if and only if there is no solution of natural numbers for the multivariate polynomial equation $\varphi(x_1,\dots,x_n) = 0$. For the case of equivalence of *While* programs, consider the *While* programs

$$r = p_\varphi; (\textbf{if}\; p > 0 \;\textbf{then}\; (x_{res} := 1) \;\textbf{else}\; skip); skip$$

and

$$\begin{aligned} r' = \;& x_{aux} := 0;\; x_{elem} := 0;\; x_{index} := 0;\; x_{res} := 1;\; x_{subres} := 0;\; x_{mcount} := 0;\\ & x^1_{multiplier} := 0;\; x^2_{multiplier} := 0;\; x^{pos}_{res} := 0;\; x^{neg}_{res} := 0; skip \end{aligned}$$

We have that r and r' are equivalent if and only if there is no solution of natural numbers for the multivariate polynomial equation $\varphi(x_1,\dots,x_n) = 0$.

Now, let the *Goto* programs κ_q, κ_r and $\kappa_{r'}$ be the translations into *While* of q, r and r', respectively. By Theorem 6.4.7, we know q terminates iff κ_q terminates, and we know r and r' are equivalent iff κ_r and $\kappa_{r'}$ are equivalent. This shows that both checking termination of q and checking equivalence of r and r' are undecidable.

Furthermore, we know there exists a control state partition of countably infinite size for programs κ_q, κ_r and $\kappa_{r'}$. As shown in the proof of Theorem 6.3.20, we know that there exists a control state partition $(P_i)_{i\in I}$ of countably infinite size for all *While* programs above (p_φ, q, r and r'). By Theorem 6.4.11, we know that $(P_i)_{i\in I}$ is a control state partition for κ_q, κ_r and $\kappa_{r'}$ as well. □

6.5 Bounding Control Flow for *FPN* and *LPN*

Analogously to the fragments identified for *While* in Section 6.3 and for *Goto* in Section 6.4, fragments of *FPN* and *LPN* could be defined that allow for decidable equivalence of programs, by structuring the control flow of programs to a bounded number of different operational structures. However, an important difference to defining such fragments for the declarative programming languages (with respect to the imperative programming languages) is that such fragments depend on the operational strategy used to evaluate programs. In the case of imperative programming languages, a particular operational semantics was already fixed, and thus the definition of the fragments with decidable reasoning problems could simply make use of this fixed operational semantics. In the case of declarative programming languages, different fragments with decidable reasoning could be defined for different evaluation strategies. We sketch some suggestions how to define decidable fragments of *FPN* and *LPN*, similar to the fragments from Sections 6.3 and 6.4.

For *FPN* a fragment of bounded control flow could be defined for an evaluation strategy that top-down evaluates function calls, and that eagerly evaluates needed function calls in determining the value of expressions. In order to do so, a notion similar to control state partitions could be defined on the set of all different tuples of natural numbers involved in the execution of a *FPN* program. This would partition this set of tuples into different classes such that (i) the class to which a tuple belongs uniquely determines what conditions match the tuple, for all function definitions in the program, and (ii) the class to which a tuple belongs uniquely determines the resulting class when this tuple is applied to the operations defined by all consequent terms in the program.

Similarly, for *LPN* a fragment of bounded control flow could be defined for a top-down, resolution-like evaluation strategy (similar to the evaluation strategy using in Prolog). In order to do so, a notion similar to control state partitions could be defined on the set of all different tuples of natural numbers involved in the execution of a *LPN* program. This would partition this set of tuples into different classes such that (i) the class to which a tuple belongs uniquely determines, for each rule, whether or not the conjunction of constraints in the rule hold for the tuple, and (ii) the class to which a tuple t belongs uniquely determines, for each relational atom in the body of each rule, the class of the tuple that the relational atom evaluates to, when the rules is evaluated for t.

In both cases, the partition of tuples into different classes would correspond to the partitioning of variables states into control states. Here, in both cases, the condition labeled 'i' corresponds to condition (2) in Definitions 6.3.4 and 6.4.4, and the condition labeled 'ii' corresponds to condition (3) in Definitions 6.3.4 and 6.4.4. Clearly, further research is needed to work out the details of these suggestions.

CHAPTER 7

Practical Applicability

The decidability results for termination and equivalence problems for programs of some of the fragments of the imperative programming languages identified in Chapter 6 (e.g. loop-free programs) are theoretically not entirely surprising, and might at first sight seem of limited practical use. However, there are real-world industrial settings in which very similar imperative programming languages are used in large-scale systems. In this chapter we discuss one such system that uses loop-free imperative programs, and the applications that the framework developed in this thesis has led to.

We consider an industrial system in use within *Siemens AG*, a multinational company. In cooperation with the *Coorporate Technologies* department of the company this system based on a imperative programming language has been investigated, and a prototype reasoning application has been developed. In the following, we discuss the system in general lines, and discuss how it relates to the setting of imperative programming languages as dealt with in this thesis. Furthermore, we discuss how on the basis of the framework developed in this thesis a prototype reasoner has been implemented, and what functionalities this reasoner has. Finally, we discuss some possibilities for further applications of the theoretical results obtained in this thesis.

7.1 A Temporal, Imperative, Rule-Based System

We describe the general working of the industrial system we consider. The system consists of a finite number of rules that are used to monitor a number of sensors. A number of components of the system are essential for understanding it. There are variables $\mathcal{X}_s$ representing the values supplied by the sensors, and there are variables $\mathcal{X}_c$ representing additional computed values. We have that $\mathcal{X} = \mathcal{X}_s \cup \mathcal{X}_c$ and $\mathcal{X}_s \cap \mathcal{X}_c = \emptyset$. In the execution of the system, there are an unbounded number of time points $T = \{1, \dots, k\}$. Each of these variables $x \in \mathcal{X}$ gets a numerical value for each time point $t \in T$. The values of the sensor variables $\mathcal{X}_s$ are given from outside the system, as they are supplied by the sensor measurements. We can thus consider the variables $\mathcal{X}_s$ as input variables. The values of the computed variables $\mathcal{X}_c$ are determined by the set of rules $\mathcal{R}$. For each computed variable $x \in \mathcal{X}_s$ there is exactly one rule $r \in \mathcal{R}$. Rules are of a fixed if-then-conditional form, and can refer to the values of other variables $x \in \mathcal{X}$ at the same or previous time points. Rules can only refer to variables at a bounded number of time points ago. Also, the set of rules does not contain any cycles at the current time point, i.e., the graph $(\mathcal{X}, \{(x, x') \mid \text{the rule for } x \text{ refers to } x' \text{ at the current time point}\})$ is acyclic. Each of the variables $x \in \mathcal{X}$ can also be undefined, which can be considered as a special value.

For the purposes of reasoning on these rule-based systems, we can consider them as imperative programs as follows. There are only a finite number of rules in the system. Each rule can only refer to a bounded number of time points. Even though the execution of the system uses an unbounded number of time points, we get that any possible situation of input values that can occur for any rule occurs when we consider all possible instantiations over only a finite, bounded number of time points T'. When considering possible behavior of rules, we can thus represent the rule system as a imperative program p (e.g. a *While* program) over the set of variables $\mathcal{X}_p = \{x_{v,t} \mid v \in \mathcal{X}, t \in T'\}$, that is loop-free (e.g. contains no subprograms of the form $\mathbf{while}\ b\ \mathbf{do}\ q$). The separate rules are expressed by subprograms containing only conditional statements and variable assignments. Since the set of rules contains no cycles, we can sequentially put together all subprograms corresponding to the separate rules.

7.2 Automated Reasoning Support

The reasoning problems for which we provide automated support are used for the purpose of debugging the rule-based system. This can be seen as an instance of the recently proposed explorative debugging paradigm [23]. In this paradigm, debugging of rule-based systems is based on declarative, semantic properties of rules such as equivalence. More concretely, we provide support for automated decision of the following semantic properties. First of all, we want to check the *consistency* of rules. We say that a rule r for variable $x \in \mathcal{X}_c$ is consistent, if and only if there exists an instantiation to the sensor variables $\mathcal{X}_s$ such that x is not undefined. Secondly, we are interested in *subsumption* of rules. We say that one rule r for variable $x \in \mathcal{X}_c$ is subsumed by another rule r' for variable $x' \in \mathcal{X}_c$, if and only if for each instantiation to the sensor variables $\mathcal{X}_s$ we have that x being defined implies both that x' is defined and that the values of x and x' coincide. Finally, we consider the property of *equivalence* of rules. We say that two rules r and r', for variables $x, x' \in \mathcal{X}_c$ respectively, are equivalent if and only if for each instantiation to the sensor variables $\mathcal{X}_s$ we have that the values of x and x' coincide. In other words, equivalence of rules is mutual subsumption.

The reasoner works by implementing the framework developed in this thesis, i.e. it considers the rules of the system as imperative programs, encodes them into formal logic, and employs already existing reasoning algorithms for the formal logic to solve the original reasoning problems. One difference with the methods proposed in this thesis is that the programs are not encoded into description logic. The expressivity of description logic is not needed for the loop-free fragment of the imperative programming languages. Instead the expressivity of an extension of Boolean logic with (numerical) concrete domains (which is a particular form of SMT, or satisfiability modulo theories) suffices. Implementations of efficient reasoning algorithms are readily available for SMT, and for $\mathcal{ALC}(\mathcal{D})$ less so. This motivated the choice to use SMT methods. In particular, the implementation that is used in the reasoner is Microsoft's Z3 [7].

Summarizing, the implemented reasoner is able to automatically decide the semantic properties of consistency, subsumption and equivalence of rules within an instance of the system (i.e. a set of rules), by means of the encoding framework developed in this thesis.

7.3 Possibilities for Further Applications

The application sketched above is based on the fragment of loop-free programs, identified in Section 6.2. The other fragments identified in Chapter 6 for which reasoning problems such as termination and equivalence are decidable might also lead to practical applications. We sketch a few suggestions how the other fragments can be used in practice.

Example 6.3.17, for instance, characterizes a class of programs for which termination and equivalence are decidable. This characterization singles out a class of programs with the required properties. We can also consider this characterization as sufficient conditions on arbitrary programs. This would give us an incomplete procedure for deciding termination and equivalence of programs, that is defined only on programs that fall within the class of programs defined in Example 6.3.17. Such an incomplete procedure could then be extended with additional cases for which sufficient conditions are known.

Note that it is unclear whether the question if there exists a finite control state partition for arbitrary programs is decidable. An incomplete procedure trying to place programs into one of a number of decidable fragments for which practical characterizations are known might therefore be a practical approach for automated reasoning on programs. Further heuristics might be developed for such procedures. Heuristics for trying to fit programs in the conditions identified in Example 6.3.17, for instance, might include strategies for identifying variables and appropriate upper bounds on the basis of the syntax of programs. For developing additional heuristics for finding finite control state partitions, it might also be useful to identify a number of control state partitions that often work for programs in practice. The heuristics might then involve trying to match arbitrary programs to these common control state partitions.

CHAPTER 8

Conclusions

8.1 Results

In this thesis, we considered the problem of developing automated reasoning methods over programs from various programming paradigms. In order to do so, we considered a number of syntactically simplified programming languages that are representative for different programming paradigms (imperative, functional and logic programming). We showed how one can assign a model-theoretic semantics to programs of the different programming languages. We did so by specifying a mapping encoding programs into statements of the description logic $\mathcal{ALC}(\mathcal{D})$, and showing that the original (operational or relational) semantics of the programs corresponds to the model-theoretic semantics of the encodings in the language of $\mathcal{ALC}(\mathcal{D})$.

We assigned this model-theoretic semantics to programs with the goal of developing algorithms that perform automated reasoning on programs of the different programming languages. In particular, we considered the reasoning problems of deciding termination and equivalence of programs. For some (straightforwardly identifiable) fragments of the programming languages this method of assigning a model-theoretic semantics quite directly leads to decision procedures that use existing $\mathcal{ALC}(\mathcal{D})$ reasoning algorithms. We showed how this can be done. In fact, programs of one such fragment (or rather, such a fragment in a variant of one of the programming languages we considered) occurs in a practical, real-world, industrial use case. We explained how a practical implementation of a reasoning algorithm based on the framework proposed in this thesis has been developed for this practical setting.

Furthermore, we showed how the conceptual framework that we developed, in combination with a number of notions and techniques from the field of description logics, can be used to identify additional classes of programs for which automated reasoning (for semantic properties such as termination and equivalence of programs) is possible. Concretely, we structured the state space that imperative programs operate on by introducing the notion of control state partitions, and showed that the problems of termination and equivalence are decidable for programs that allow finite control state partitions. Also, we argued that this is the maximal class of programs with this decidability property, when considering only the notion of control state partitions.

Deciding termination or equivalence of programs with infinitely many control states, in general, is undecidable.

Summarizing, we introduced a novel approach for a semantic analysis of programming languages, that is based on a translation into formal logic and that allows for automated reasoning algorithms in a number of settings.

8.2 Other Logical Formalisms for Reasoning over Programs

In the formal verification of software, logical methods have been widely used for a long time (cf. [9, 18]). We briefly consider a number of different logical formalisms that can be used to analyze the semantics of programs, and we relate these alternative approaches to the approach we developed in this thesis.

One logical formalism that has been used for formally verifying semantic properties of programs is the Hoare calculus (or Hoare logic) [11, 13]. The Hoare calculus is a proof system that works with preconditions and postconditions to prove correctness of programs. It is mainly used to formally verify whether a program gives the right output (e.g. does a program in fact compute the function it is supposed to compute). It is less suitable, however, to check equivalence of programs by means of automated reasoning. Also, since the Hoare calculus is a proof calculus, it is difficult to get counterexamples when using methods based on Hoare calculus. For description logic based methods, on the other hand, obtaining counterexamples in case these exist is often quite straightforward.

Another logical formalism used to analyze (sequential) programs is propositional dynamic logic (PDL) [10]. Similarly to the description logic used in our approach, PDL has a model-theoretic semantics, which enables a straightforward search for counterexamples. However, there has been much less work on extensions of PDL with concrete domains (i.e. numerical values) than the work that has been done investigating the combination of description logics and concrete domains. Also, since description logics have become popular for many different application domains, including the semantic web, many more implementations and optimizations of reasoning algorithms are available for description logics than for PDL.

The formalism of temporal logics has also been used widely to reason about computational behavior. Within the field of model checking (cf. [3, 6] for textbooks on the subject, for instance), computational processes are represented using formal models and temporal logics are used to check whether the behavior of these processes satisfy certain properties. Logics often used in this setting are linear-time temporal logic (LTL) and computational tree logic (CTL), as well as further logics based on these two logics. The computational processes analyzed within this paradigm are often not specified as (sequential) programs, however, but rather as transition models or using formal languages specifying such transition models.

Note that besides the approaches mentioned above, a magnitude of research has been done on the topic of (automated) reasoning over programs, ever since the field of computer science was established. We certainly do not claim that this section on alternative (logical) approaches to this research area is exhaustive (or even close to exhaustive). Relating the work done in this thesis to this vast body of research remains a topic of further research.

8.2.1 Advantages of the Model-Theoretic Approach

We used the assignment of a model-theoretic semantics to imperative programs (by means of the encoding of programs into the description logic $\mathcal{ALC}(\mathcal{D})$) to obtain the above decidability results. Of course, all of the results could be phrased without using any of these model-theoretic methods as well. But using this model-theoretic approach has a number of (conceptual) advantages.

First of all, it allows us to use the well-known mathematical models of relational structures, which are familiar and intuitive to reason about. Additionally, these relational structures offer a flexible way of representing multiple derivations at the same time, possibly including additional auxiliary elements that relate these derivations (such as is done in Section 4.3.3, for instance). Furthermore, using a model-theoretic semantics makes it possible to use ideas and techniques often used in combination with such mathematical structures. For instance, in our proof of decidability of equivalence of programs with finitely many control states, the idea of using homomorphisms is essential. This notion of homomorphic mappings is often used in combination with relational structures.

Finally, the model-theoretic setting we used in our approach is very general, and can capture any form of computation (as indicated by Theorem 4.8.1). Therefore, we get a very flexible approach towards automated reasoning on programs that works for very different programming paradigms. None of the other approaches for reasoning over programs offer this kind of generality and flexibility.

8.3 Further Research

Possibly the most important direction of further research is to investigate to what extent the theoretical results obtained in this thesis can be used in practical settings. The class of programs for which there exists finite control state partitions, as identified in Chapter 6, is a fragment of the programming language allowing termination and equivalence to be decided. However, it is not yet a practically applicable characterization, since finding out whether finite control state partitions exist for (arbitrary) programs is non-trivial. As described in Section 7.3, some incomplete methods could be developed on the basis of some results in this thesis. This is a significant direction of further research. More generally, further research is needed to design practically more useful characterizations of fragments of programming languages for which reasoning tasks such as termination and equivalence are decidable. Such characterizations could be based on the fragments identified in this thesis, but could also include further fragments of the programming languages.

Additional further research includes extending the model-theoretic framework for automated reasoning on programs developed in this thesis to further settings. For instance, the framework could be extended to programming languages that are used in practice, with concrete domains that include more than just the natural numbers. It could also be extended to settings that include concurrency or nondeterminism. Also, it could be extended to logic programming languages that do allow for built-in search mechanisms using free variables (like Prolog).

Another line of further research would be to characterize more reasoning problems on programs, that are relevant in practical settings, by means of the description logic language used in the framework.

A third possibility for further research would be to identify additional classes of programs (larger than the classes identified in Chapter 6) for which the problems of termination and equivalence of programs are decidable. Also, for the decidable fragments of *While* and *Goto* identified in Sections 6.3 and 6.4, corresponding fragments could be investigated for the languages *FPN* and *LPN*, as suggested in Section 6.5.

Bibliography

[1] R. Alur, T.A. Henzinger, G. Lafferriere, and G.J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971 –984, July 2000.

[2] Franz Baader and Ulrike Sattler. Tableau algorithms for description logics. *Studia Logica*, 69:2001, 2000.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[4] Evert W. Beth. Semantic Entailment and Formal Derivability. *Koninklijke Nederlandse Akademie van Wentenschappen, Proceedings of the Section of Sciences*, 18:309–342, 1955.

[5] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[6] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, volume 4963 of *TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Francesco M. Donini and Fabio Massacci. EXPTIME tableaux for ALC. *Artificial Intelligence*, 124(1):87–138, November 2000.

[9] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[10] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194 – 211, 1979.

[11] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[12] Thomas A. Henzinger. Hybrid automata with finite bisimulations. In *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*, ICALP '95, pages 324–335, London, UK, 1995. Springer-Verlag.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[14] Ian Horrocks and Ulrike Sattler. Ontology reasoning in the SHOQ(D) description logic. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 199–204. Morgan Kaufmann, 2001.

[15] Carsten Lutz. *The Complexity of Description Logics with Concrete Domains*. PhD thesis, RWTH Aachen University, 2002.

[16] Carsten Lutz. Description logics with concrete domains—a survey. In *Advances in Modal Logics, Volume 4*. King's College Publications, 2003.

[17] Carsten Lutz. NExpTime-complete description logics with concrete domains. *ACM Transactions on Computational Logic*, 5(4):669–705, October 2004.

[18] Zohar Manna. Properties of programs and the first-order predicate calculus. *Journal of the ACM*, 16(2):244–255, April 1969.

[19] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[20] Hanne R. Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[21] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, February 1991.

[22] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[23] Valentin Zacharias. *Tool Support for Finding and Preventing Faults in Rule Bases*. PhD thesis, Karlsruhe Institute of Technology, 2008.