NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

The Expressive Power of Higher-order Datalog: An XSB Implementation

Evangelos N. Protopapas

Supervisor: Panos Rondogiannis, Professor

ATHENS

OCTOBER 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Η Εκφραστική Ισχύς της Datalog Υψηλής-τάξης: Μία Υλοποίηση στο XSB

Ευάγγελος Ν. Πρωτόπαπας

Επιβλέπων: Πάνος Ροντογιάννης, Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2018

BSc THESIS

The Expressive Power of Higher-order Datalog: An XSB Implementation

Evangelos N. Protopapas

S.N.: 1115201400169

Supervisor: Panos Rondogiannis, Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Η Εκφραστική Ισχύς της Datalog Υψηλής-τάξης: Μία Υλοποίηση στο XSB

Ευάγγελος Ν. Πρωτόπαπας

Α.Μ.: 1115201400169

Επιβλέπων: Πάνος Ροντογιάννης, Καθηγητής

ABSTRACT

This thesis follows the results in the yet unpublished paper of *A. Charalambidis, Ch. Nomikos and P. Rondogiannis* namely *“The Expressive Power of Higher-order Datalog”*. That paper proposes a proof which shows that Higher-order Datalog is equivalent in computational power to exponentially time bounded Turing Machines. In other words that higher-order Datalog captures the complexity class of decision problems EXP^kTIME. This thesis will review the above result in detail while demonstrating and proposing solutions for the flaws in the programs written in that paper. In addition a working implementation of the programs in the XSB system will be provided which shows that the proposed results hold.

SUBJECT AREA: Proof on expressive power equivalency in programming languages

KEYWORDS: Higher-order Datalog, Expressivity, Complexity Theory, Turing machine simulation

ΠΕΡΙΛΗΨΗ

Η πτυχιακή εργασία ακολουθεί τα αποτελέσματα του μη δημοσιευμένου ακόμα άρθρου των, *Α. Χαραλαμπίδη, Χ. Νομικού και Π. Ροντογιάννη* με τίτλο *“The Expressive Power of Higher-order Datalog”*. Το άρθρο παρουσιάζει μία απόδειξη ισοδυναμίας σε εκφραστική ισχύ της Datalog υψηλής-τάξης, με τις χρονικά εκθετικά περιορισμένες μηχανές Turing. Με άλλα λόγια ότι η Datalog υψηλής-τάξης εκφράζει τις κλάσεις πολυπλοκότητας των προβλημάτων απόφασης EXP^{k}TIME. Η πτυχιακή εργασία αυτή, θα παρουσιάσει με λεπτομέρεια τα παραπάνω αποτελέσματα, καθώς και θα υποδείξει κάποια σφάλματα στα προγράμματα που αναγράφονται στο άρθρο, καθώς και θα προτείνει τρόπους επίλυσής τους. Επιπλέον θα παρατεθεί μία λειτουργική υλοποίηση των προγραμμάτων στο σύστημα XSB, με στόχο την τεκμηρίωση των παραπάνω αποτελεσμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Απόδειξη εκφραστικής ισοδυναμίας γλωσσών προγραμματισμού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Datalog υψηλής τάξης, Εκφραστικότητα, Θεωρία πολυπλοκότητας, Εξομοίωση μηχανής Turing

ACKNOWLEDGEMENTS

I would like to thank professor Panos Rondogiannis for being a great supervisor and mentor throughout the development of my thesis and for having such a strong influence on my current academic interests by introducing me to a more theoretical side of Computer Science.

I would also like to thank my family for their great support and understanding during my Bachelor Studies and especially during these last few months developing my thesis.

CONTENTS

1. INTRODUCTION

1.1. Introduction to expressivity

Before proceeding with the developments of the paper we introduce the notion of *expressivity* of a programming language. Intuitively, if every program that can be written in a language **X** can also be written in a language **Y** using only local transformations, while there are programs written in language **Y** that cannot be written in language **X** without changing their entire structure (i.e. not purely through local transformations), then we can say that language **Y** is more **expressive** than language **X**.One might argue that the above definition admits the possibility that there are pairs of languages where there are programs in **X** which cannot be expressed in **Y** and programs in **Y** which cannot be expressed in **X**, thus neither language is more expressive than the other and we cannot compare them. This is true in a sense and it justifies our real-world experience where one language is better at some things that the other and vice versa.

As stated in [1], comparing the set of computable functions that a language can represent is useless because the languages in question are usually universal and no other measures exist. Thus the need for developing a formal framework for comparing programming languages arises.

As a result it is a common practice when studying expressivity to use *complexity theory* as a means to characterize the expressive power of a programming language through *complexity classes* of *decision problems*. In [2] *complexity theory* is used to determine the expressive power of functional programming. In that paper expressivity is distinguished in *”absolute expressivity”* and *”relative expressivity”*. An *”absolute expressivity”* question on a programming language feature **X** is: ”Do there exist problems that **can** be solved by programs with feature **X**, and **cannot** be solved **without** feature **X**?”. A *”relative expressivity”* question is: ”Is there a problem that can be solved both with and without feature **X**, but such that *any solution without feature* **X** is **necessarily less efficient** than some solution using **X**?”.

It is difficult to answer a *”relative expressivity”* question as there exist many different efficient simulations of one programming language feature by others. This leads to very small complexity differences. *”Relative expressivity”* is a field with many conceptualized ideas but very few proven results.

Additionally there exists an issue when studying *”absolute expressivity”* (which is the expressivity we discuss in this thesis. From this point forward we will refer to *”absolute expressivity”* simply as expressivity). Studying expressivity in a strong language (i.e. Turing complete) is futile. The reason is that any Turing complete language can compute all μ-recursive functions (i.e. partial recursive) and thus in an absolute sense are all equally expressive. A solution to this problem is to restrict a language of features until a *Turing-incomplete* language is obtained. Then it is possible to use *complexity theory* to study the expressivity of different language features.

In [2] a particularly interesting class of functional programs are studied. Those without the **CONS** feature. This essentially means that no construction of new data is allowed. This instance of functional programs are very similar to the logic programs that we are studying in this thesis. A *Higher-order Datalog* program is in reality a program of a more powerful *Higher-order Logic Programming Language* without the ability to construct new data due to the lack of *function symbols*. As expected this class of functional programs is proven to solve all decision problems in $\text{EXP}^{k}\text{TIME}$ where k is the order of the input to the program. Which is the same result that the paper of *A. Charalambidis, Ch. Nomikos and P. Rondogiannis* demonstrates.

Complexity theory focuses on classifying computational problems in *complexity classes* according to their solution's difficulty. To be able to capture this inherent difficulty to solve a problem, the need for a *model of computation* arises to study these problems and quantify their *complexity*. The most commonly used *model of computation* in *complexity theory* is the *Turing machine*. There exists a belief in the field of *theoretical computer science* which states that every physically realizable computational device can be simulated by a Turing machine. This belief is known as the **Church-Turing (CT) thesis**. Though it is not proven, the CT thesis is a belief about the nature of the world as we currently understand it and no realized computational system available today has been able to falsify this belief.

The most common type of problems studied in complexity theory are *decision problems*. A decision problem is informally defined as a problem whose answer is either *yes* or *no* (i.e. either **1** or **0**). A decision problem can also be viewed as a *formal language* where instances of inputs whose output is yes are members of the language. As expected the complexity classes of these problems are defined using Turing machines. Complexity classes can be defined in terms of a resource of computation consumed such as *time* or *space*. In the following chapter we will be discussing Turing machines and complexity classes in detail.

At this point it may have become obvious that in order to prove that some *Turing-incomplete* programming language captures some *complexity class* defined in terms of some *Turing machine* it may suffice to show that the language can simulate the Turing machine. In reality this is only part of the proof. The other part requires that any program in the language can be simulated by an algorithm that meets the resource requirements of the *complexity class*.

For example in [3] Papadimitriou demonstrated that the class of languages decided by (first-order) Datalog is the class computable by *polynomial-time bounded Turing machines*, or in other words equivalent to the well-known class P. The proof consists of the following:

- Showing that there exists an algorithm that runs in $O((n+c)^{2d})$ (which is clearly a polynomial of n) which simulates a Datalog program.
- Designing a Datalog program that simulates a Turing machine that runs in time n^{d}.

The proof in *A. Charalambidis, Ch. Nomikos and P. Rondogiannis* proceeds in a similar way. In this thesis we will be particularly interested in studying the *Higher-order Datalog* program that simulates an *exponentially-time bounded Turing machine*.

1.2. Structure of thesis

This thesis consists of six chapters. The *Introduction*, *Background*, *A Higher-order Datalog language*, *Simulating in Higher-order Datalog*, *Corrections* and *Conclusion* chapters.

Introduction introduces some basic ideas regarding *complexity theory* that will be needed in the rest of this thesis. A brief explanation of the structure of this thesis is also included.

Background introduces the reader to knowledge necessary to the understanding of the paper. *Complexity classes* and *Turing machines* are among the topics discussed in this chapter.

A Higher-order Datalog language defines a basic language suited for the needs of the proof.

Simulating in Higher-order Datalog discusses what we need and how we can represent that in Higher-order Datalog in order to simulate a Turing machine. Then the program will be presented.

Corrections discusses minor faults that appear in the program of the original paper and proposes some possible solutions.

Conclusion sums up the topics discussed in this thesis and explains intuitively why the proof is valid.

In the *Appendix* instructions on where to find the program implemented in the XSB system will be given.

2. BACKGROUND

This chapter introduces some basic concepts and definitions that are necessary to follow the proof that is reviewed in chapter 4.

2.1. Languages

The definitions in this section are from [4].

A word over an alphabet can be any finite sequence of letters. The set of all words over an alphabet Σ is usually denoted by Σ^*.

Definition 2.1. Suppose an alphabet Σ and a function $f : \Sigma^* \to \{0, 1\}$. We shall identify such a function f with the set $L_f = \{x \mid f(x) = 1\}$ and call such sets *languages* or *decision problems*.

The problem of deciding the language L_f (i.e., given x decide whether $x \in L_f$) is equivalent to computing the function f (i.e., given x compute $f(x)$).

2.2. Turing machines

The definitions in this section are mostly from [5].

The *Turing machine*, first proposed by Alan Turing in 1936, is a powerful model of computation with an unlimited and unrestricted memory. A Turing machine is an accurate formalization of a general purpose computer, in the sense that it can do everything a real computer can do. Of course even a Turing machine cannot solve some problems. These are the problems that are beyond the theoretical limits of computation.

The machine uses an infinite *tape* as its memory. It has a *tape head* that can move around the tape, *read* or *write* symbols. Initially the tape contains the input string followed my empty symbols. The machine continues computing until certain conditions are met and the machine enters an *accept* or *reject state*. In every other case the machine never halts.

A formal definition of a Turing machine follows.

Definition 2.2. A *Turing machine* is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ, are all finite sets and

- Q is the set of states,
- Σ is the input alphabet not containing the *blank symbol* $\sqcup$,
- Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\textbf{L}, \textbf{S}, \textbf{R}\}$ is the transition function,
- $q_0 \in Q$ is the start state,

- $q_{accept} \in Q$ is the *accept* state, and
- $q_{reject} \in Q$ is the *reject* state, where $q_{reject} \neq q_{accept}$.

in [5] the transition function is defined as $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathbf{L}, \mathbf{R}\}$. **S** means the head stays in place, **L** means the head moves to the left, **R** means the head moves to the right. The original definition allows for two types of transitions:

- The rule "if the head is in symbol σ and in state q then write symbol σ', go to state q' and move the head left" translates to $\delta(q, \sigma) = (q', \sigma', \mathbf{L})$.
- The rule "if the head is in symbol σ and in state q then write symbol σ', go to state q' and move the head right" translates to $\delta(q, \sigma) = (q', \sigma', \mathbf{R})$

The altered definition allows the additional transition rule "if the head is in symbol σ and in state q then write symbol σ' and go to state q'" which translates to $\delta(q, \sigma) = (q', \sigma', \mathbf{S})$. It is easy to see that adding **S** to the set of possible moves does not alter the functionality of the machine. If the additional transition rule is used the original transition rules can be simulated as follows. The two rules $\delta(q, \sigma) = (q'', \sigma', \mathbf{S})$, $\delta(q, \sigma') = (q', \sigma', \mathbf{L})$ are equivalent to $\delta(q, \sigma) = (q', \sigma', \mathbf{L})$. Symmetrically for **R**. Even though one rule is now defined using two rules the computational complexity of the machine is not affected as there are only finitely many rules used.

Now let's see how a Turing machine M computes. Suppose w is the input string with length n. M is initialized with the leftmost n tape squares containing w and the rest blank. The head starts on the leftmost symbol on the tape (i.e the first symbol of s). M proceeds computing in *steps* according to the transition function δ. As M computes changes occur in the current state, the current tape contents and the current head location. These three items compose a *configuration* of M defined as follows.

Definition 2.3. A *configuration* of a Turing machine is a tuple, $\langle q, w \rangle$, where $q \in Q$ and $w \in \Gamma^*$. q denotes the current *state*. w denotes the current *tape contents* (finite string). A *dotted* $\dot{\sigma} \in w$ denotes the symbol pointed to by the *head*.

The intuitive understanding of a *step of computation* is captured through *configurations*. Suppose two configurations

$$C_1 = \langle q_i, \sigma_1\sigma_2 \ldots \dot{\sigma_k} \ldots \sigma_n \rangle \text{ and } C_2 = \langle q_j, \sigma_1\sigma_2 \ldots \sigma_k\dot{\sigma_{k+1}} \ldots \sigma_n \rangle.$$

We say that

$$\text{if } \delta(q_i, \sigma_k) = (q_j, \sigma_k, \mathbf{R}) \text{ then } C_1 \textit{ yields } C_2.$$

The configuration $\langle q_0, \dot{\sigma_1}\sigma_2 \ldots \sigma_n \rangle$ is a *start configuration*, $\langle q_{accept}, \ldots \dot{\sigma_i} \ldots \rangle$ is an *accepting configuration* and $\langle q_{reject}, \ldots \dot{\sigma_j} \ldots \rangle$ is a *rejecting configuration*.

Definition 2.4. A Turing machine M *accepts* input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where

- C_1 is the *start configuration* of M on input w,

- each C_i yields C_{i+1} and
- C_k is an *accepting configuration*.

The set of strings that M *accepts* is the *language* of M, or the *language recognized* by M and is denoted $L(M)$.

In order to better comprehend the aforementioned definitions we continue with a Turing machine example.

Example. In this example we define a Turing machine M that decides $L = \{0^{2^n} \mid n \geq 0\}$ (i.e. the language consisting of all strings of 0's whose length is a power of 2). In other words $L(M) = \{0^{2^n} \mid n \geq 0\}$.

Informally the Turing machine sweeps across the tape keeping track of whether the number of 0's is even or odd. If that number is odd and greater than 1, the input cannot be a power of 2. Therefore, the machine rejects. Otherwise if the number is exactly 1, the machine accepts.

Formally we define $M = (Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$ where $Q = \{q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$, $\Sigma = \{0\}$, $\Gamma = \{0, x, \sqcup\}$ and δ is described in the following state diagram.

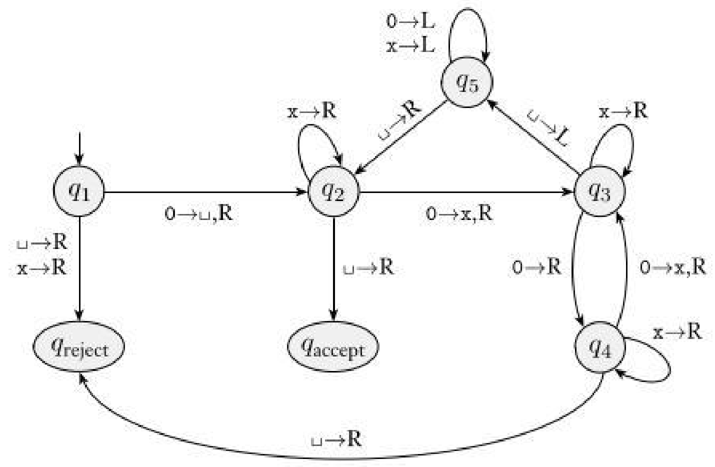

Figure 1: State diagram for M.

Running the machine with input 0000 gives the following sequence of *configurations*. (read column wise)

$$\begin{array}{llll}
\langle q_1, \dot{0}000\rangle & \rightarrow \langle q_5, \sqcup\dot{x}0x\sqcup\rangle & \rightarrow \langle q_5, \sqcup x\dot{x}x\sqcup\rangle & \rightarrow \ \langle q_{accept}, \sqcup xxx \sqcup \dot{\sqcup}\rangle \\
\langle q_2, \sqcup\dot{0}00\rangle & \rightarrow \langle q_5, \dot{\sqcup}x0x\sqcup\rangle & \rightarrow \langle q_5, \sqcup\dot{x}xx\sqcup\rangle & \rightarrow \\
\langle q_3, \sqcup x\dot{0}0\rangle & \rightarrow \langle q_2, \sqcup\dot{x}0x\sqcup\rangle & \rightarrow \langle q_5, \dot{\sqcup}xxx\sqcup\rangle & \rightarrow \\
\langle q_4, \sqcup x0\dot{0}\rangle & \rightarrow \langle q_2, \sqcup x\dot{0}x\sqcup\rangle & \rightarrow \langle q_2, \sqcup\dot{x}xx\sqcup\rangle & \rightarrow \\
\langle q_3, \sqcup x0x\dot{\sqcup}\rangle & \rightarrow \langle q_3, \sqcup xx\dot{x}\sqcup\rangle & \rightarrow \langle q_2, \sqcup x\dot{x}x\sqcup\rangle & \rightarrow \\
\langle q_5, \sqcup x0\dot{x}\sqcup\rangle & \rightarrow \langle q_3, \sqcup xxx\dot{\sqcup}\rangle & \rightarrow \langle q_2, \sqcup xx\dot{x}\sqcup\rangle & \rightarrow \\
\langle q_5, \sqcup x\dot{0}x\sqcup\rangle & \rightarrow \langle q_5, \sqcup xx\dot{x}\sqcup\rangle & \rightarrow \langle q_2, \sqcup xxx\dot{\sqcup}\rangle & \rightarrow
\end{array}$$

2.3. Complexity Classes

The definitions in this section are mostly from [5].

As already stated, a *complexity class* is a set of functions that can be computed with respect to a given *resource*. Throughout this thesis we restrict our attention to the type of functions defined in Section 2.1 which represent *decision problems*.

We first define the notion of a *time bounded deterministic Turing machine*.

Definition 2.5. Let M be a deterministic Turing machine that halts on all inputs. The *running time* or *time complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the *maximum number of steps* that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine.

The first complexity class that we define is the following.

Definition 2.6. Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. TIME$(t(n)) = \{L \mid L$ is a language decided by an $O(t(n))$ time deterministic Turing machine$\}$

Now we can define the classes needed to follow the proof that will be demonstrated later in this thesis.

Definition 2.7. Suppose $T_0(n)(d) = n^d$ and $T_k(n)(d) = 2^{T_{k-1}(n)(d)}$ for $k, d \geq 0$. We define the *complexity classes* EXP$^{\text{k}}$TIME as

$$\text{EXP}^{\text{k}}\text{TIME} = \bigcup_{d \in \mathbb{N}} \text{TIME}(T_k(n)(d)) \quad \text{for} \quad k \geq 0.$$

Additionally

$$\text{EXP}^{0}\text{TIME} \subsetneq \text{EXP}^{1}\text{TIME} \subsetneq \text{EXP}^{2}\text{TIME} \subsetneq \cdots \subsetneq \text{EXP}^{\text{k}}\text{TIME} \ldots$$

proven from the classical *time hierarchy theorem*.

In simple words EXP^{0}TIME is the set of all decision problems that are solvable by a deterministic Turing machine in time $O(n^d)$ where $d \geq 0$, EXP^{1}TIME in time $O(2^{n^d})$, EXP^{2}TIME in time $O(2^{2^{n^d}})$ and so on and so forth. The class EXP^{0}TIME is the well-known class P.

This concludes the background knowledge required regarding *computability and complexity theory*, to sufficiently comprehend the proof that follows in chapter 4.

3. A HIGHER-ORDER DATALOG LANGUAGE

Unlike functional programming which is a programming paradigm that heavily promotes Higher-order programming, logic programming is traditionally restricted to *first-order logic theories*. This convention is largely based on the fact that even *second-order logic theories* are in general undecidable and *extensional semantics* (i.e. semantics in which predicates are represented as sets of arguments for which they hold) for a language implementation cannot be established. In an extensional language, two predicates that succeed for the same instances are considered equal.

The first attempts at defining *Higher-order logic systems* where made by [6] with HiLog and [7] with λ-Prolog. While very useful, the issue with these systems is that they are *intensional*. In general an intensional system classifies its objects based on their *name* without regard for their representation as a *set*. Thus in an intentional system equality between predicates may only be based on the names of the predicates. This leads to *context sensitive* systems which essentially means that a given program that is written for a specific task may not behave in the expected way when present in a different context.

One of the first works (to my knowledge) to propose an *extensional higher order Horn logic* was Wadge in [8]. Wadge identified a subset of Higher-order Horn logic that accepts extensional semantics that in a way extend the classical first-order semantics. The first work to propose a purely extensional theoretical framework for higher-order logic programming was [9]. That paper extended the work in [8] and proposed *minimum model semantics* for the higher-order case in a way that naturally extend the classical first-order minimum model semantics.

Informally, by higher-order, logicians mean a language in which variables are allowed to appear in places where normally predicates or function symbols do. In a higher-order Logic Programming languages we can define atoms like `R(X, Y)`, where `R` will be instantiated with a predicate. Additionally we can pass predicates as arguments to higher-order predicates. For example `p(R, X)` where `R` is a predicate.

Classical Datalog is extended to the higher-order case in the same manner. A *Higher-order Datalog language* can be defined as the *subset* of some *Higher-order Logic Programming language* in which *function symbols* do not exist.

3.1. Syntax

We begin by defining the alphabet of $\mathcal{HD}$.

Definition 3.1. The alphabet of the *Higher-order Datalog* language $\mathcal{HD}$ consists of the *disjoint* sets, $\mathcal{PV}$ of *predicate variables*, $\mathcal{PC}$ of *predicate constants*, $\mathcal{NV}$ of *non-predicate variables*, $\mathcal{C}$ of *non-predicate constants*, the symbols "$\leftarrow$", "$=$", "(", ")", "," and the unique predicate `not/1`. Additionally we define $\mathcal{V} = \mathcal{PV} \cup \mathcal{NV}$ and call it the set of *argument variables*.

Usually we denote predicate variables with $P, Q, R \ldots$, predicate constants with $p, q, r \ldots$, non-predicate variables with $X, Y, Z \ldots$ and non-predicate constants with $a, b, c \ldots$.

Definition 3.2. Suppose a program $Program$ of $\mathcal{HD}$. The *syntax* of $Program$ is given by the following grammar:

$$\begin{array}{rcl}
\langle Program \rangle & ::= & \langle Clause \rangle \mid \langle Clause \rangle\ \langle Program \rangle \\
\langle Clause \rangle & ::= & \langle Head \rangle\ \leftarrow\ (\ \langle Atom \rangle\ )\ ,\ \ldots\ ,\ (\ \langle Atom \rangle\ ).\ \mid \langle Head \rangle. \\
\langle Atom \rangle & ::= & \langle Pos_{atom} \rangle \mid \langle Neg_{atom} \rangle \\
\langle Head \rangle & ::= & \{\mathcal{PC}\}\ \langle Argument_{head} \rangle\ \ldots\ \langle Argument_{head} \rangle \\
\langle Pos_{atom} \rangle & ::= & \langle Name \rangle\ \langle Argument_{pos} \rangle\ \ldots\ \langle Argument_{pos} \rangle \\
 & \mid & \langle Argument_{fo} \rangle\ =\ \langle Argument_{fo} \rangle \\
\langle Neg_{atom} \rangle & ::= & \texttt{not}\ (\{\mathcal{PC}\}\ \langle Argument_{fo} \rangle\ \ldots\ \langle Argument_{fo} \rangle) \\
 & \mid & \langle Argument_{fo} \rangle\ =\ \langle Argument_{fo} \rangle \\
\langle Argument_{pos} \rangle & ::= & (\ \langle Pos_{atom} \rangle\ ) \mid \{\mathcal{V}\} \mid \{\mathcal{PC}\} \mid \{\mathcal{C}\} \\
\langle Argument_{head} \rangle & ::= & \{\mathcal{V}\} \mid \{\mathcal{C}\} \\
\langle Argument_{fo} \rangle & ::= & \{\mathcal{NV}\} \mid \{\mathcal{C}\} \\
\langle Name \rangle & ::= & \{\mathcal{PC}\} \mid \{\mathcal{PV}\}
\end{array}$$

The notation $\{\mathcal{X}\}$ means an *item* from any set $\mathcal{X}$. In the rule for "$\langle Clause \rangle$" the amount of atoms may vary. The same holds for "$\langle Head \rangle$", "$\langle Pos_{atom} \rangle$" and "$\langle Neg_{Atom} \rangle$". Additionally only first-order predicates can be negated.

The syntax proposed in definition 3.2 imposes the following restrictions to our clauses. Arguments in the head of a clause are only allowed to contain predicate variables and not predicate constants. We will additionally restrict our clauses to contain distinct predicate variables in the head and that these variables are the only predicate variables present in the body. Then as proposed in [8] our clauses are *definitional*.

Definition 3.3. A higher-order Horn clause is *definitional* iff

- in the head of the clause each argument that is a predicate is a *predicate variable* local to the clause,
- the *predicate variables* are all distinct,
- they are the only *predicate variables* local to the clause.

3.2. Semantics

By restricting our language to contain only *definitional* clauses, we could devise extensional higher-order semantics for our language similarly to [9] or [10]. Since this is not the subject of this thesis, we avoid the implications of defining extensional semantics and choose to define intensional semantics based on [6].

In [6] it is shown that there exists an encoding of Hilog programs into classical first-order Prolog programs. This encoding allows Hilog to be a powerful Higher-order language while having simple First-order semantics.

For our language we shall define a translation of $\mathcal{HD}$ to Hilog and the use Hilog's encoding to Prolog.

Definition 3.4. A clause in $\mathcal{HD}$ can be translated to HiLog with the following set of rules:

$$\begin{aligned}
&\mathbf{T}_{\mathcal{HD}}(H \leftarrow A_1,\ \ldots\ ,\ A_n)\ =\ \mathbf{T}_{\mathcal{HD}}(H)\ \leftarrow\ \mathbf{T}_{\mathcal{HD}}(A_1),\ \ldots\ ,\ \mathbf{T}_{\mathcal{HD}}(A_n)\\
&\mathbf{T}_{\mathcal{HD}}(\sigma_{pred}\,\sigma_{arg}\ \ldots\ \sigma_{arg})\ =\ \mathbf{T}_{\mathcal{HD}}(\sigma_{pred})(\mathbf{T}_{\mathcal{HD}}(\sigma_{arg}))\ldots(\mathbf{T}_{\mathcal{HD}}(\sigma_{arg}))\\
&\mathbf{T}_{\mathcal{HD}}((\sigma_{pred}\,\sigma_{arg}\ \ldots\ \sigma_{arg}))\ =\ \mathbf{T}_{\mathcal{HD}}(\sigma_{pred})(\mathbf{T}_{\mathcal{HD}}(\sigma_{arg}))\ldots(\mathbf{T}_{\mathcal{HD}}(\sigma_{arg}))\\
&\mathbf{T}_{\mathcal{HD}}((\sigma_{arg}\ =\ \sigma_{arg}))\ =\ \mathbf{T}_{\mathcal{HD}}(\sigma_{arg})\ =\ \mathbf{T}_{\mathcal{HD}}(\sigma_{arg})\\
&\mathbf{T}_{\mathcal{HD}}(\{\mathcal{X}\}) = \{\mathcal{X}\}
\end{aligned}$$

Again $\{\mathcal{X}\}$ means an item from any *set* $\mathcal{X}$.

Using Hilog's encoding to Prolog, our language can be interpreted using the classical first-order semantics, such as the *well-founded semantics*. Additionally the system XSB implements HiLog using an encoding to Prolog, combined with SLG resolution which is equivalent to the well-founded semantics. This means that our language has the same semantics as the programs we practically implement using XSB. For detailed information on HiLog's encoding, the well-founded semantics and SLG resolution the interested reader is redirected to [6], [11], [12].

For example a valid program in our language

```
closure R X Y  ←  (R X Y).
closure R X Y  ←  (R X Z), (closure R Z Y).
```

Translates to

```
closure(R)(X)(Y)  ←  R(X)(Y).
closure(R)(X)(Y)  ←  R(X)(Z), closure(R)(Z)(Y).
```

Which is a valid HiLog program using *general predicates*. Notice that we chose to translate to general predicates like `closure(R)(X)(Y)` instead of the common `closure(R, X, Y)`. General predicates can be used for *partial application* which is a desirable property for our language to possess.

Intuitively partial application allows us to represent objects through relations, meaning that `closure(R)(X)(Y)` is not interpreted as a 3-ary predicate. Instead `closure(R)` represents the transitive closure of the relation `R`, `closure(R)(X)` represents the the transitive closure of the relation `R` on item `X` and only when `closure(R)(X)(Y)` is called, which represents the the transitive closure of the relation `R` on items `X,Y`, the code executes.

4. SIMULATING IN HIGHER-ORDER DATALOG

This chapter reviews the simulation present in *A. Charalambidis, Ch. Nomikos and P. Rondogiannis*.

The *logic programming paradigm* differs drastically from the computational model of *Turing machines*. On one hand, Turing machines represent more of an *imperative* style of execution. An input stream is represented through the tape, while the transition function defines a clear set of *instructions* that can be followed imperatively to execute the machine.

On the other hand, logic programming represents a declarative style of programming. Logic programs express facts and rules to prove *truth* or *falsity* of objects in some domain. The resolution used in logic programming systems incorporates *backward reasoning* constructing trees that constitute the search space for solving a goal.

4.1. Representation

Therefore is it clear that in order to simulate a Turing machine using Higher-order Datalog we first need to develop a way to bridge the gaps between these two styles of execution.

4.1.1. Input / Acceptance

The most straightforward way to encode a string of length n in Higher-order Datalog is by using a *relation*. Suppose an input string $w = \sigma_1\sigma_2 \dots \sigma_n \in \Sigma^*$. The first idea that comes to mind is a relation of the form $\{\sigma_1,\ \sigma_2,\ \dots,\ \sigma_n\}$. However this format would not allow us to pick out an arbitrary symbol based on its *position* (recall that sets are unordered). A more suitable representation would be the relation $\{(0, \sigma_1, 1),\ (1, \sigma_2, 2),\ \dots,\ (n-1, \sigma_n, n)\}$. Notice that each symbol is paired with its position along with the position of the next symbol on the tape. We call this relation the `input` relation.

Acceptance can be simply represented as a 0-ary predicate. Proving *truth* or *falsity* for the predicate is equivalent to *accepting* or *rejecting* the input on a Turing machine. We call this predicate `accept`.

We shall say that the program decides a language $L \subseteq \Sigma^*$ if for any $w \in \Sigma^* : w \in L$ *iff* the program, when given w encoded through the `input` relation, has `accept` as a logical consequence.

4.1.2. Transition Rules

The transition rules of δ can be expressed very naturally as logical rules. Recall that an instance of δ is of the form $\delta(q, \sigma) = (q', \sigma', \mathbf{S})$ which means “if the head is in symbol σ and in state q then write symbol σ' and go to state q'”. Therefore the left side of the equation states *facts* that are true at the *current step* of computation, while the right side states *facts* that should be true at the *next step* of computation *if* the facts stated on the left hold at the *current step*.

As a result we can describe the above transition with the following informal logical clauses

$$\begin{aligned}&(\texttt{at step}\,T)(\texttt{symbol}\,\sigma\,\texttt{is in position}\,X \wedge\\ &\qquad\texttt{head is in position}\,X \wedge\\ &\qquad\texttt{in state}\,q) \;\rightarrow\; (\texttt{at step}\,T+1)(\texttt{symbol}\,\sigma'\,\texttt{is in position}\,X).\end{aligned}$$

$$\begin{aligned}&(\texttt{at step}\,T)(\texttt{symbol}\,\sigma\,\texttt{is in position}\,X \wedge\\ &\qquad\texttt{head is in position}\,X \wedge\\ &\qquad\texttt{in state}\,q) \;\rightarrow\; (\texttt{at step}\,T+1)(\texttt{in state}\,q').\end{aligned}$$

$$\begin{aligned}&(\texttt{at step}\,T)(\texttt{symbol}\,\sigma\,\texttt{is in position}\,X \wedge\\ &\qquad\texttt{head is in position}\,X \wedge\\ &\qquad\texttt{in state}\,q) \;\rightarrow\; (\texttt{at step}\,T+1)(\texttt{head is in position}\,X).\end{aligned}$$

A set of clauses in the above form can represent the δ function of any deterministic Turing machine. Since logic programming systems use *backward reasoning* the logical consequence of `accept` will be proven by simulating the Turing machine *backwards*. The following example demonstrates this idea.

Example. Consider a Turing machine that only accepts the string ab. The transition function δ is defined as follows:

$$\begin{aligned}\delta(q_0,a) &= (q_1,a,\mathbf{R})\\ \delta(q_1,b) &= (q_2,b,\mathbf{R})\\ \delta(q_2,\sqcup) &= (q_{accept},\sqcup,\mathbf{S})\end{aligned}$$

A forward execution of the Turing machine would be:

$$\begin{aligned}&\texttt{step 0: read}\,a\,\texttt{in state}\,q_0,\,\texttt{go to state}\,q_1,\,\texttt{move head right}.\\ &\texttt{step 1: read}\,b\,\texttt{in state}\,q_1,\,\texttt{go to state}\,q_2,\,\texttt{move head right}.\\ &\texttt{step 2: read}\,\sqcup\,\texttt{in state}\,q_2,\,\texttt{go to state}\,q_{accept}.\\ &\texttt{step 3: in state}\,q_{accept}\,\texttt{so accept}.\end{aligned}$$

A backward reasoning proof of `accept`:

$$\begin{aligned}&\texttt{step 3: To prove accept, prove in step 2 that state is}\,q_2,\\ &\qquad\texttt{head does not move from}\,q_2\,\texttt{to}\,q_{accept},\\ &\qquad\texttt{reading}\,\sqcup.\\ &\texttt{step 2: Prove in step 1 that state is}\,q_1,\\ &\qquad\texttt{head comes by moving right from}\,q_1\,\texttt{to}\,q_2,\\ &\qquad\texttt{reading}\,b.\end{aligned}$$

```
step 1: Prove in step 0 that state is $q_0$,
        head comes by moving right from $q_0$ to $q_1$,
        reading $a$.
step 0: Since machine is in $q_0$ in step 0, accept.
```

This simplified explanation captures the way that the simulating program executes the Turing machine.

4.1.3. Numbering

First note that our Higher-order Datalog language does not have any built-in numerical system. Using such a system defeats the purpose of the proof as it is possible to encode data using numbers and in a sense it is the same as including function symbols in our language. Thus our language would not be a Datalog language and more than likely we would obtain a *Turing complete* language that studying its expressivity would be pointless.

Finally in order to simulate an *exponentially time bounded Turing machine* we need to be able to count the *steps* of computation. In other words we need to devise a way to represent numbers and count using Higher-order Datalog *relations*.

Papadimitriou in [3] already demonstrated that we can use first-order relations to represent any number up to n^d where n is the length of the input. In the proof we are reviewing this is achieved by using the positions of each fact in the `input` relation. The way `input` is defined allows us to count from 0 up to $n-1$. We extend the range of numbers we can represent up to $n^d - 1$ by considering tuples of d elements. A number in this range is represented as d individual numbers in the range $\{0 \dots n-1\}$.

Recall that we want to prove that k $+$ 1-order Datalog captures EXP$^{\mathrm{k}}$TIME. This means that we have to extend the range of represented numbers to $T_k(n)(d)$ using the notation in definition 2.7.

We start with second-order Datalog. The range we have to represent is 2^{n^d}. Obviously, we must utilize the ability of our language to define second-order predicates. The idea is that a function $f_1\colon \{0, \dots, n^d - 1\} \to \{\mathtt{low}, \mathtt{high}\}$ can be used to represent a number in this range. Such a function is equivalent to a string of n^d bits, and such a string can represent any number in the required range.

Suppose for example that we have an input of length 2 and we are only using tuples of 1 element. Then the base numbers that we can represent are $\{0, 1\}$. Observe that under these assumptions the numbers we can represent with f_1 as *sets* are

$$\begin{aligned}
0 &: \{(0, \mathtt{low}), (1, \mathtt{low})\} \\
1 &: \{(0, \mathtt{low}), (1, \mathtt{high})\} \\
2 &: \{(0, \mathtt{high}), (1, \mathtt{low})\} \\
3 &: \{(0, \mathtt{high}), (1, \mathtt{high})\}
\end{aligned}$$

To generalize for k $+$ 1-order Datalog assume the following set of functions:

$$\{f_i : f_{i-1} \to \{\texttt{low}, \texttt{high}\}\} \text{ for all } 2 \leq i \leq k$$
$$\text{where } f_1 \colon \{0, \dots, n^d - 1\} \to \{\texttt{low}, \texttt{high}\}$$

Then any number in the range $T_k(n)(d)$ can be represented by f_k.

4.2. Simulation

4.2.1. d-tuple and order$_1$ number representations

We now present the program. The `input` relation as already stated represents an input tape over a fixed alphabet Σ. Suppose $\Sigma = \{a, b\}$. An input of length n can be encoded with n facts of the `input` predicate as follows:

```
input 0 a 1.
input 1 b 2.
⋮
input n-2 b n-1.
input n-1 a n.
```

We proceed by defining predicates `base_zero`, `base_last`, `base_succ`, `base_pred` to simulate the numbers in the range $\{0, \dots, n-1\}$. `base_zero` is true of 0 (i.e. of the left index of the *first* tuple in the `input` relation), `base_last` is true of $n-1$ (i.e. of the left index of the *last* tuple in the `input` relation), `base_succ` given a number k returns $k+1$ and `base_pred` given a number $k+1$ returns k.

```
base_zero 0.
base_last I   ← (input I X J), (not (input J X1 K)).
base_succ I J ← (input I X J).
base_pred I J ← (input J X I).
```

We proceed by defining `tuple_zero`, `tuple_last`, `tuple_succ`, `tuple_pred` to extend the range of numbers to $\{0, \dots, n^d - 1\}$. As stated before these predicates act on collections of d arguments to represent a number in tuple notation. We use the notation $\overline{\texttt{X}}$ to abbreviate d arguments $\texttt{X}_1 \ \dots \ \texttt{X}_d$.

```
tuple_zero X̄ ← (base_zero X₁) ,..., (base_zero X_d).
tuple_last X̄ ← (base_last X₁) ,..., (base_last X_d).
```

`tuple_zero` is true of the d-tuple which represents "0" and `tuple_last` is true of "$n^d - 1$". To define `tuple_succ` we need d clauses that act on two d-tuples to simulate the way we find the successor of a number that consists of d digits of base n. `tuple_pred` can be easily defined using `tuple_succ`. As expected `tuple_succ` given the d-tuple representation of "k" returns the d-tuple representation of "$k+1$" while `tuple_pred` given "$k+1$" returns "k".

$$
\begin{aligned}
\texttt{tuple\_succ}\ \overline{\texttt{X}}\ \overline{\texttt{Y}} \leftarrow\ & (\texttt{X}_1 = \texttt{Y}_1)\ ,\dots,\ (\texttt{X}_{d-1} = \texttt{Y}_{d-1}),\\
& (\texttt{base\_succ}\ \texttt{X}_d\ \texttt{Y}_d).\\
\texttt{tuple\_succ}\ \overline{\texttt{X}}\ \overline{\texttt{Y}} \leftarrow\ & (\texttt{X}_1 = \texttt{Y}_1)\ ,\dots,\ (\texttt{X}_{d-2} = \texttt{Y}_{d-2}),\\
& (\texttt{base\_succ}\ \texttt{X}_{d-1}\ \texttt{Y}_{d-1}),\\
& (\texttt{base\_last}\ \texttt{X}_d),\ (\texttt{base\_zero}\ \texttt{Y}_d).\\
& \vdots\\
\texttt{tuple\_succ}\ \overline{\texttt{X}}\ \overline{\texttt{Y}} \leftarrow\ & (\texttt{base\_succ}\ \texttt{X}_1\ \texttt{Y}_1),\\
& (\texttt{base\_last}\ \texttt{X}_2)\ ,\dots,\ (\texttt{base\_last}\ \texttt{X}_d),\\
& (\texttt{base\_zero}\ \texttt{Y}_2)\ ,\dots,\ (\texttt{base\_zero}\ \texttt{Y}_d).\\
\texttt{tuple\_pred}\ \overline{\texttt{X}}\ \overline{\texttt{Y}} \leftarrow\ & (\texttt{tuple\_succ}\ \overline{\texttt{Y}}\ \overline{\texttt{X}}).
\end{aligned}
$$

Additionally we define the predicates `less_than`, `tuple_non_zero` working on d-tuples. `less_than` defines the "$<$" relation on d-tuples and `tuple_non_zero` succeeds if its argument is not equal to the d-tuple representing "0".

$$
\begin{aligned}
\texttt{less\_than}\ \overline{\texttt{X}}\ \overline{\texttt{Y}} &\leftarrow (\texttt{tuple\_succ}\ \overline{\texttt{X}}\ \overline{\texttt{Y}}).\\
\texttt{less\_than}\ \overline{\texttt{X}}\ \overline{\texttt{Y}} &\leftarrow (\texttt{tuple\_succ}\ \overline{\texttt{X}}\ \overline{\texttt{Z}}),\ (\texttt{less\_than}\ \overline{\texttt{Z}}\ \overline{\texttt{Y}}).\\
\texttt{tuple\_non\_zero}\ \overline{\texttt{X}} &\leftarrow (\texttt{tuple\_zero}\ \overline{\texttt{Z}}),\ (\texttt{less\_than}\ \overline{\texttt{Z}}\ \overline{\texttt{X}}).
\end{aligned}
$$

We now proceed to define the predicates necessary to extend the range of numbers to $\{0,\dots,2^{n^d}\}$ using the technique we described in subsection 4.1.3. As in the case of f_1 we will subscript all predicates related to this range by 1. We shall call these numbers $\texttt{order}_1$ `numbers`.

We first define the predicates $\texttt{zero}_1$ and $\texttt{last}_1$ that represent the first and last numbers in the range (i.e. 0 and $2^{n^d}-1$). $\texttt{zero}_1$ is equivalent to a string of n^d bits where all bits are `low`, while $\texttt{last}_1$ is equivalent to a string of n^d bits where all bits are `high`. As *sets* these numbers are equivalent to:

$$
\begin{aligned}
\texttt{zero}_1 &: \{(0,\texttt{low}),(1,\texttt{low}),\dots,(n^d-1,\texttt{low})\}\\
\texttt{last}_1 &: \{(0,\texttt{high}),(1,\texttt{high}),\dots,(n^d-1,\texttt{high})\}
\end{aligned}
$$

We now define the predicates, where the first argument which determines the *bit position* is a d-tuple and the second argument which determines the *bit value* is $\{\texttt{low},\texttt{high}\}$. Note that we consider the least significant bit position to be that of `tuple_zero`, while the most significant bit position to be that of `tuple_last`.

$$
\begin{aligned}
&\texttt{zero}_1\ \overline{\texttt{X}}\ \texttt{low}.\\
&\texttt{last}_1\ \overline{\texttt{X}}\ \texttt{high}.
\end{aligned}
$$

The predicate $\texttt{is\_zero}_1$ follows, that checks if its argument is equal to $\texttt{zero}_1$. A number in our current representation is "0" iff every d-tuple bit position has the value `low`. So we define an additional predicate, namely $\texttt{all}_1$, to determine if every bit is `low`. The syntactically higher-order literal $(\texttt{N}\ \overline{\texttt{X}}\ \texttt{V})$, applies $\overline{\texttt{X}}$ and `V` to the relation passed to the variable `N`. Essentially we are asking if the number represented by `N` at the bit position denoted by the d-tuple $\overline{\texttt{X}}$ has value `V`.

$$\begin{aligned}
&\texttt{is\_zero}_1\ \texttt{N} \leftarrow (\texttt{tuple\_last}\ \overline{\texttt{X}}),\ (\texttt{all}_1\ \texttt{low}\ \texttt{N}\ \overline{\texttt{X}}).\\
&\texttt{all}_1\ \texttt{V}\ \texttt{N}\ \overline{\texttt{X}} \leftarrow (\texttt{tuple\_zero}\ \overline{\texttt{X}}),\ (\texttt{N}\ \overline{\texttt{X}}\ \texttt{V}).\\
&\texttt{all}_1\ \texttt{V}\ \texttt{N}\ \overline{\texttt{X}} \leftarrow (\texttt{tuple\_non\_zero}\ \overline{\texttt{X}}),\ (\texttt{N}\ \overline{\texttt{X}}\ \texttt{V}),\\
&\qquad\qquad (\texttt{tuple\_pred}\ \overline{\texttt{X}}\ \overline{\texttt{Y}}),\ (\texttt{all}_1\ \texttt{V}\ \texttt{N}\ \overline{\texttt{Y}}).
\end{aligned}$$

In a similar manner we define the predicate $\texttt{non\_zero}_1$ that succeeds if its argument is not equal to $\texttt{zero}_1$. In this case a number in our representation is $\neq$ "0" if at least one d-tuple bit position has the value `high`. The predicate $\texttt{exists}_1$ checks whether the above holds.

$$\begin{aligned}
&\texttt{non\_zero}_1\ \texttt{N} \leftarrow (\texttt{tuple\_last}\ \overline{\texttt{X}}),\ (\texttt{exists}_1\ \texttt{high}\ \texttt{N}\ \overline{\texttt{X}}).\\
&\texttt{exists}_1\ \texttt{V}\ \texttt{N}\ \overline{\texttt{X}} \leftarrow (\texttt{N}\ \overline{\texttt{X}}\ \texttt{V}).\\
&\texttt{exists}_1\ \texttt{V}\ \texttt{N}\ \overline{\texttt{X}} \leftarrow (\texttt{tuple\_non\_zero}\ \overline{\texttt{X}}),\\
&\qquad\qquad (\texttt{tuple\_pred}\ \overline{\texttt{X}}\ \overline{\texttt{Y}}),\ (\texttt{exists}_1\ \texttt{V}\ \texttt{N}\ \overline{\texttt{Y}}).
\end{aligned}$$

Symmetrically, we define the predicates $\texttt{is\_last}_1$, by using $\texttt{all}_1$ to determine whether all d-tuple bit positions are `high`, and $\texttt{non\_last}_1$, by using $\texttt{exists}_1$ to determine whether at least one d-tuple bit position has the value `low`, by reversing the value passed to `V`.

$$\begin{aligned}
&\texttt{is\_last}_1\ \texttt{N} \leftarrow (\texttt{tuple\_last}\ \overline{\texttt{X}}),\ (\texttt{all}_1\ \texttt{high}\ \texttt{N}\ \overline{\texttt{X}}).\\
&\texttt{non\_last}_1\ \texttt{N} \leftarrow (\texttt{tuple\_last}\ \overline{\texttt{X}}),\ (\texttt{exists}_1\ \texttt{low}\ \texttt{N}\ \overline{\texttt{X}}).
\end{aligned}$$

After sufficiently having defined the way to represent the bounds of $\{0,\ldots,2^{n^d}\}$, namely $\texttt{zero}_1$ and $\texttt{last}_1$, we can represent any number in the required range based on these bounds. We shall define the predicates $\texttt{pred}_1$ and $\texttt{succ}_1$ to respectively capture the notion of the predecessor and successor of a number.

The approach taken to define $\texttt{pred}_1$ and $\texttt{succ}_1$ differs drastically to that of `tuple_pred` and `tuple_succ`. Intuitively the reason for this is that `tuple_pred` is able to check whether a d-tuple number is the predecessor of another d-tuple number, and even generate the predecessor, because a number in d-tuple notation has a specific form that can be manipulated to produce another number from it. Recall that a d-tuple is a sequence of d individual variables that take values from the *constants* used by the `input` relation. Since in $\texttt{order}_1$ notation, numbers are *relations* and the only relations defined are $\texttt{zero}_1$ and $\texttt{last}_1$ we must find a way to *build* any other number using $\texttt{zero}_1$ or $\texttt{last}_1$. To achieve this the predicates $\texttt{pred}_1$ and $\texttt{succ}_1$ have to be defined in a way that their *partial application* on a given number `N` (i.e. $(\texttt{pred}_1\ \texttt{N})$ and $(\texttt{succ}_1\ \texttt{N})$) represents the predecessor or successor of N respectively. Recall than in section 3.2 of chapter 3 we discussed briefly that partial application can be used to represent objects. For example the number ($\texttt{succ}_1$ ($\texttt{succ}_1$ ($\texttt{succ}_1$ $\texttt{zero}_1$))) can be defined in terms of ($\texttt{succ}_1$ ($\texttt{succ}_1$ $\texttt{zero}_1$)) which will recursively lead to $\texttt{zero}_1$ which is fully defined.

We start of with $\texttt{pred}_1$ and then symmetrically define $\texttt{succ}_1$. Suppose a number in the form of a binary string. The predecessor of the number can be obtained by inverting all bits to the right of the last 1 including the last 1. We shall define three auxiliary predicates namely $\texttt{bits}_1$, $\texttt{exists\_to\_right}_1$ and $\texttt{all\_to\_right}_1$. $\texttt{bits}_1$ returns the value of the bit in position $\overline{X}$ which is inverted if all bits to the right of $\overline{X}$ are low. $\texttt{exists\_to\_right}_1$ checks whether there is at least one bit to the right of the given position $\overline{X}$ with value V. $\texttt{all\_to\_right}_1$ checks if all bits to the right of $\overline{X}$ have value V.

$$
\begin{array}{ll}
\texttt{pred}_1\ N\ \overline{X}\ V & \leftarrow (\texttt{is\_zero}_1\ N),\ (N\ \overline{X}\ V).\\
\texttt{pred}_1\ N\ \overline{X}\ V & \leftarrow (\texttt{non\_zero}_1\ N),\ (\texttt{bits}_1\ N\ \overline{X}\ V).\\
\\
\texttt{bits}_1\ N\ \overline{X}\ V & \leftarrow (\texttt{exists\_to\_right}_1\ \texttt{high}\ N\ \overline{X}),\ (N\ \overline{X}\ V).\\
\texttt{bits}_1\ N\ \overline{X}\ V & \leftarrow (\texttt{all\_to\_right}_1\ \texttt{low}\ N\ \overline{X}),\\
 & \quad (N\ \overline{X}\ V1),\ (\texttt{invert}\ V1\ V).\\
\\
\texttt{exists\_to\_right}_1\ V\ N\ \overline{X} & \leftarrow (\texttt{tuple\_non\_zero}\ \overline{X}),\\
 & \quad (\texttt{tuple\_pred}\ \overline{X}\ \overline{Y}),\ (N\ \overline{Y}\ V).\\
\texttt{exists\_to\_right}_1\ V\ N\ \overline{X} & \leftarrow (\texttt{tuple\_non\_zero}\ \overline{X}),\ (\texttt{tuple\_pred}\ \overline{X}\ \overline{Y}),\\
 & \quad (\texttt{exists\_to\_right}_1\ V\ N\ \overline{Y}).\\
\\
\texttt{all\_to\_right}_1\ V\ N\ \overline{X} & \leftarrow (\texttt{tuple\_zero}\ \overline{X}).\\
\texttt{all\_to\_right}_1\ V\ N\ \overline{X} & \leftarrow (\texttt{tuple\_non\_zero}\ \overline{X}),\ (\texttt{tuple\_pred}\ \overline{X}\ \overline{Y}),\\
 & \quad (N\ \overline{Y}\ V),\ (\texttt{all\_to\_right}_1\ V\ N\ \overline{Y}).
\end{array}
$$

Symmetrically the successor of a number can be obtained by inverting all bits to the right of the last 0 including the last 0. We define $\texttt{succ}_1$ as follows.

$$
\begin{array}{ll}
\texttt{succ}_1\ N\ \overline{X}\ V & \leftarrow (\texttt{is\_last}_1\ N),\ (N\ \overline{X}\ V).\\
\texttt{succ}_1\ N\ \overline{X}\ V & \leftarrow (\texttt{non\_last}_1\ N),\\
 & \quad (\texttt{exists\_to\_right}_1\ \texttt{low}\ N\ \overline{X}),\ (N\ \overline{X}\ V).\\
\texttt{succ}_1\ N\ \overline{X}\ V & \leftarrow (\texttt{non\_last}_1\ N),\\
 & \quad (\texttt{all\_to\_right}_1\ \texttt{high}\ N\ \overline{X}),\\
 & \quad (N\ \overline{X}\ V1),\ (\texttt{invert}\ V1\ V).
\end{array}
$$

Invert is easily defined as

```
invert low high.
invert high low.
```

Notice that the predicates $\texttt{pred}_1$ and $\texttt{succ}_1$, only give us information about the value V of a bit at the given position $\overline{X}$ based on a given number N. As we already pointed out we cannot construct a concrete representation of an $\texttt{order}_1$ number, but instead we can use the representation of another $\texttt{order}_1$ number to determine each bit individually. Thus we can say that the partial applications of $\texttt{pred}_1$ and $\texttt{succ}_1$ are indeed the numbers we wish to construct.

We will also need the equality between two numbers. We can easily define the predicate $\texttt{equal}_1$ with an additional predicate $\texttt{equal\_test}_1$ that checks if every d-tuple bit position has the same value `V`.

$$
\begin{array}{lll}
\texttt{equal}_1\ \texttt{N M} & \leftarrow & \texttt{(tuple\_last}\ \overline{\texttt{X}}\texttt{), (equal\_test}_1\ \texttt{N M}\ \overline{\texttt{X}}\texttt{).} \\
\texttt{equal\_test}_1\ \texttt{N M}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_zero}\ \overline{\texttt{X}}\texttt{), (N}\ \overline{\texttt{X}}\ \texttt{V), (M}\ \overline{\texttt{X}}\ \texttt{V).} \\
\texttt{equal\_test}_1\ \texttt{N M}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_non\_zero}\ \overline{\texttt{X}}\texttt{), (N}\ \overline{\texttt{X}}\ \texttt{V), (M}\ \overline{\texttt{X}}\ \texttt{V),} \\
 & & \texttt{(tuple\_pred}\ \overline{\texttt{X}}\ \overline{\texttt{Y}}\texttt{), (equal\_test}_1\ \texttt{N M}\ \overline{\texttt{Y}}\texttt{).}
\end{array}
$$

4.2.2. Arbitrary $\texttt{order}_{k+1}$ number representation

Now we can easily define any arbitrary $\texttt{order}_{k+1}$ number based on $\texttt{order}_k$ numbers. Using the notation of definition 2.7 the range of numbers that $\texttt{order}_{k+1}$ numbers need to represent is defined recursively as $T_{k+1}(n)(d) = 2^{T_k(n)(d)}$. The technique we use is the same as for $\texttt{order}_1$ numbers, the only difference being that now each bit position is represented by an $\texttt{order}_k$ number instead of the d-tuples we used only for $\texttt{order}_1$ numbers.

First we define $\texttt{zero}_{k+1}$ and $\texttt{last}_{k+1}$.

$$
\begin{array}{l}
\texttt{zero}_{k+1}\ \texttt{X low.} \\
\texttt{last}_{k+1}\ \texttt{X high.}
\end{array}
$$

Next we define $\texttt{is\_zero}_{k+1}$ that succeeds if its argument is $\texttt{zero}_{k+1}$.

$$
\begin{array}{lll}
\texttt{is\_zero}_{k+1}\ \texttt{N} & \leftarrow & \texttt{(all}_{k+1}\ \texttt{low N last}_k\texttt{).} \\
\texttt{all}_{k+1}\ \texttt{V N X} & \leftarrow & \texttt{(is\_zero}_k\ \texttt{X), (N X V).} \\
\texttt{all}_{k+1}\ \texttt{V N X} & \leftarrow & \texttt{(non\_zero}_k\ \texttt{X), (N X V), (all}_{k+1}\ \texttt{V N (pred}_k\ \texttt{X)).}
\end{array}
$$

Next we define $\texttt{non\_zero}_{k+1}$ that succeeds if its argument is not $\texttt{zero}_{k+1}$.

$$
\begin{array}{lll}
\texttt{non\_zero}_{k+1}\ \texttt{N} & \leftarrow & \texttt{(exists}_{k+1}\ \texttt{high N last}_k\texttt{).} \\
\texttt{exists}_{k+1}\ \texttt{V N X} & \leftarrow & \texttt{(N X V).} \\
\texttt{exists}_{k+1}\ \texttt{V N X} & \leftarrow & \texttt{(non\_zero}_k\ \texttt{X), (exists}_{k+1}\ \texttt{V N (pred}_k\ \texttt{X)).}
\end{array}
$$

Symmetrically we define $\texttt{is\_last}_{k+1}$ and $\texttt{non\_last}_{k+1}$.

$$
\begin{array}{lll}
\texttt{is\_last}_{k+1}\ \texttt{N} & \leftarrow & \texttt{(all}_{k+1}\ \texttt{high N last}_k\texttt{).} \\
\texttt{non\_last}_{k+1}\ \texttt{N} & \leftarrow & \texttt{(exists}_{k+1}\ \texttt{low N last}_k\texttt{).}
\end{array}
$$

Next we define pred$_{k+1}$ using the above definitions.

```
pred$_{k+1}$ N X V            ← (is_zero$_{k+1}$ N), (N X V).
pred$_{k+1}$ N X V            ← (non_zero$_{k+1}$ N), (bits$_{k+1}$ N X V).

bits$_{k+1}$ N X V            ← (exists_to_right$_{k+1}$ high N X), (N X V).
bits$_{k+1}$ N X V            ← (all_to_right$_{k+1}$ low N X),
                                (N X V1), (invert V1 V).

exists_to_right$_{k+1}$ V N X ← (non_zero$_k$ X), (N (pred$_k$ X) V).
exists_to_right$_{k+1}$ V N X ← (non_zero$_k$ X),
                                (exists_to_right$_{k+1}$ V N (pred$_k$ X)).

all_to_right$_{k+1}$ V N X    ← (is_zero$_k$ X).
all_to_right$_{k+1}$ V N X    ← (non_zero$_k$ X), (N (pred$_k$ X) V),
                                (all_to_right$_{k+1}$ V N (pred$_k$ X)).
```

Symmetrically we define succ$_{k+1}$.

```
succ$_{k+1}$ N X V ← (is_last$_{k+1}$ N), (N X V).
succ$_{k+1}$ N X V ← (non_last$_{k+1}$ N),
                     (exists_to_right$_{k+1}$ low N X), (N X V).
succ$_{k+1}$ N X V ← (non_last$_{k+1}$ N),
                     (all_to_right$_{k+1}$ high N X),
                     (N X V1), (invert V1 V).
```

They equality predicates is defined as follows.

```
equal$_{k+1}$ N M          ← (equal_test$_{k+1}$ N M last$_k$).

equal_test$_{k+1}$ N M I  ← (is_zero$_k$ I), ( N I V), ( M I V).
equal_test$_{k+1}$ N M I  ← (non_zero$_k$ I), ( N I V), ( M I V),
                            (equal_test$_{k+1}$ N M (pred$_k$ I)).
```

Finally we will additionally define the predicate less_than$_{k+1}$ which defines the "<" relation on order$_{k+1}$ numbers. This predicate also applies to order$_1$ numbers. The predicate recursively passes to itself the predecessors of the given numbers. If the leftmost number reaches zero$_{k+1}$ first then the predicate succeeds.

```
less_than$_{k+1}$ N M ← (is_zero$_{k+1}$ N), (non_zero$_{k+1}$ M).
less_than$_{k+1}$ N M ← (non_zero$_{k+1}$ N), (non_zero$_{k+1}$ M),
                        (less_than$_{k+1}$ (pred$_{k+1}$ N) (pred$_{k+1}$ M)).
```

4.2.3. Turing machine simulation

Recall that in subsection 4.1.2, we introduced the idea that logical rules can express any transition rule of the δ function of a Turing machine. The simulation will follow along those observations. For every transition rule of δ we will define a set of clauses that utilize the notion of a *step* of computation. As already discussed these sets of clauses will express what is true in the *current time-step*, based on a transition rule and what is true in the *previous time-step*.

For this purpose we introduce the predicates $\texttt{symbol}_\sigma$, $\texttt{state}_s$ and `cursor`. $\texttt{symbol}_\sigma$ acts on two arguments `T` (i.e. the *time-step*) and `X` (i.e. the position on the tape). If the predicate succeeds for given `T`, `X`, we interpret the result as symbol σ appears on position `X` of the tape at time-step `T`. $\texttt{state}_s$ acts on argument `T` (i.e. the time-step) and is interpreted as the machine is in state s at time-step `T`. Finally `cursor` acts on argument `T` and represents an $\texttt{order}_k$ number which indicates the position of the head at time-step `T`.

All predicates that follow are defined using $\texttt{order}_k$ numbers, the value of k depends on the Turing machine that we want to simulate (i.e. the number of steps needed for termination). We first define the predicate $\texttt{base\_to\_higher}_k$ that succeeds if the two arguments represent the same number, where the first argument is in base notation (i.e. constants of `input`) whereas the second argument is in $\texttt{order}_k$ notation.

$$
\begin{aligned}
&\texttt{base\_to\_higher}_k\ \texttt{0 N} \leftarrow (\texttt{equal}_k\ \texttt{N zero}_k).\\
&\texttt{base\_to\_higher}_k\ \texttt{M N} \leftarrow (\texttt{input J}\ \sigma\ \texttt{M}),\\
&\qquad\qquad\qquad\qquad\quad (\texttt{base\_to\_higher}_k\ \texttt{J}\ (\texttt{pred}_k\ \texttt{N})).
\end{aligned}
$$

Intuitively the above predicate recursively reduces the base notation number by 1 through `input`, and the $\texttt{order}_k$ notation number through $\texttt{pred}_k$ and succeeds if both numbers reach zero simultaneously.

We proceed with the initialization rules of the Turing machine. The initialization rules state that at time-step $\texttt{zero}_k$, the machine starts at the starting state denoted by s_0, the first n tape squares hold the input followed by the empty character ⊔ and the head starts at the first tape square (i.e. at position $\texttt{zero}_k$). We define one predicate for each symbol $\sigma \in \Sigma$.

$$
\begin{aligned}
&\texttt{symbol}_\sigma\ \texttt{T X} &&\leftarrow (\texttt{is\_zero}_k\ \texttt{T}),\ (\texttt{input Y}\ \sigma\ \texttt{W}),\\
& &&\quad (\texttt{base\_to\_higher Y X}).\\
&\texttt{symbol}_\sqcup\ \texttt{T X} &&\leftarrow (\texttt{is\_zero}_k\ \texttt{T}),\ (\texttt{base\_last Y}),\\
& &&\quad (\texttt{base\_to\_higher Y X}).\\
&\texttt{state}_{s_0}\ \texttt{T} &&\leftarrow (\texttt{is\_zero}_k\ \texttt{T}).\\
&\texttt{cursor T X low} &&\leftarrow (\texttt{is\_zero}_k\ \texttt{T}).
\end{aligned}
$$

Note that `cursor` returns for every bit position the value `low`. For $\texttt{T = zero}_k$, `(cursor T)` represents the number $\texttt{zero}_k$ (i.e. "0"). The set of $\texttt{symbol}_\sigma$ succeed if the position (i.e. `Y`) of σ on tape matches with the position given in `X`.

Now we can proceed to define the transition rules. In section 4.1.2 we translated a transition rule into a set of informal logical clauses. Since a transition rules affects the symbol that is written on the tape, the state and the position of the head in the next *time-step* we define clauses for $\texttt{symbol}_\sigma$, $\texttt{state}_s$ and `cursor`.

We start with the rule "if the head is in symbol σ and in state s then write symbol σ' and go to state s'"

$$
\begin{array}{lll}
\texttt{symbol}_{\sigma'}\ \texttt{T X} & \leftarrow & (\texttt{non\_zero}_k\ \texttt{T}), \\
 & & (\texttt{equal}_k\ \texttt{X (cursor (pred}_k\ \texttt{T)))}, \\
 & & (\texttt{state}_s\ \texttt{(pred}_k\ \texttt{T))}, \\
 & & (\texttt{symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T)))}. \\
\\
\texttt{state}_{s'}\ \texttt{T} & \leftarrow & (\texttt{non\_zero}_k\ \texttt{T}), \\
 & & (\texttt{state}_s\ \texttt{(pred}_k\ \texttt{T))}, \\
 & & (\texttt{symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T)))}. \\
\\
\texttt{cursor T I V} & \leftarrow & (\texttt{non\_zero}_k\ \texttt{T}), \\
 & & (\texttt{state}_s\ \texttt{(pred}_k\ \texttt{T))}, \\
 & & (\texttt{symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T)))}. \\
 & & (\texttt{cursor (prev}_k\ \texttt{T) I V)}.
\end{array}
$$

The definitions are straightforward. The preconditions are the same for each predicate. The machine must be in state s in the previous time-step (i.e. $\texttt{(pred}_k\ \texttt{T)}$), and the symbol σ must be read in the previous time-step, at the position of the head in the previous time-step (i.e. $\texttt{(cursor (pred}_k\ \texttt{T))}$). In the clause for $\texttt{symbol}_{\sigma'}$, $\texttt{(equal}_k\ \texttt{X (cursor (pred}_k\ \texttt{T)))}$ states that the given position `X` must be the same with the position of the head in the previous time-step. In the clause for `cursor T I V`, $\texttt{(cursor (prev}_k\ \texttt{T) I V)}$ states that the head stays in place (i.e. `(cursor T)` represents the same number as $\texttt{(cursor (prev}_k\ \texttt{T))}$).

Similarly we define clauses for the rule "if the head is in symbol σ and in state s then go to state s' and move the head right".

$$
\begin{array}{lll}
\texttt{symbol}_{\sigma'}\ \texttt{T X} & \leftarrow & (\texttt{non\_zero}_k\ \texttt{T}), \\
 & & (\texttt{equal}_k\ \texttt{X (succ}_k\ \texttt{(cursor (pred}_k\ \texttt{T))))}, \\
 & & (\texttt{state}_s\ \texttt{(pred}_k\ \texttt{T))}, \\
 & & (\texttt{symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T)))}, \\
 & & (\texttt{symbol}_{\sigma'}\ \texttt{(pred}_k\ \texttt{T) (succ}_k\ \texttt{(cursor (pred}_k\ \texttt{T))))}. \\
\\
\texttt{state}_{s'}\ \texttt{T} & \leftarrow & (\texttt{non\_zero}_k\ \texttt{T}), \\
 & & (\texttt{state}_s\ \texttt{(pred}_k\ \texttt{T))}, \\
 & & (\texttt{symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T)))}.
\end{array}
$$

$$
\begin{aligned}
\texttt{cursor T I V} \leftarrow\ & \texttt{(non\_zero}_k\ \texttt{T),} \\
& \texttt{(state}_s\ \texttt{(pred}_k\ \texttt{T)),} \\
& \texttt{(symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T))).} \\
& \texttt{((succ}_k\ \texttt{(cursor (prev}_k\ \texttt{T))) I V).}
\end{aligned}
$$

Note that we define a clause for some symbol σ' even though the transition rule does not alter any symbols. This can be explained by the fact that no symbol will automatically be transferred in a sense to the next time-step but instead it must be proven to exist in the next time-step by some clause. As a result a clause must be defined for every symbol σ' that might appear to the right of the head before the rule is applied, in order to transfer that symbol to the same position at the next time-step. Hence the literals (equal$_k$ X (succ$_k$ (cursor (pred$_k$ T)))) and (symbol$_\sigma$ (pred$_k$ T) (cursor (pred$_k$ T))) in the clause for σ'. The same applies for symbol σ. We do not implicitly define a rule for it here since afterwards we will define a set of general rules that transfer all symbols to the following time-steps as long as their position is not the same as the head, at the time-step that is to be proven. Here symbol σ falls into this category since at time-step T the head points to the position next to the symbol. Additionally in the clause for cursor T I V, ((succ$_k$ (cursor (prev$_k$ T))) I V) states that the number represented by (cursor T) is the successor of (cursor (prev$_k$ T)) (i.e. the head at time-step T points to the right of the head at time-step T-1).

Symmetrically we define the last rule “if the head is in symbol σ and in state s then go to state s' and move the head left”.

$$
\begin{aligned}
\texttt{symbol}_{\sigma'}\ \texttt{T X} \leftarrow\ & \texttt{(non\_zero}_k\ \texttt{T),} \\
& \texttt{(equal}_k\ \texttt{X (pred}_k\ \texttt{(cursor (pred}_k\ \texttt{T)))),} \\
& \texttt{(state}_s\ \texttt{(pred}_k\ \texttt{T)),} \\
& \texttt{(symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T))),} \\
& \texttt{(symbol}_{\sigma'}\ \texttt{(pred}_k\ \texttt{T) (pred}_k\ \texttt{(cursor (pred}_k\ \texttt{T)))).}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{state}_{s'}\ \texttt{T} \leftarrow\ & \texttt{(non\_zero}_k\ \texttt{T),} \\
& \texttt{(state}_s\ \texttt{(pred}_k\ \texttt{T)),} \\
& \texttt{(symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T))).}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{cursor T I V} \leftarrow\ & \texttt{(non\_zero}_k\ \texttt{T),} \\
& \texttt{(state}_s\ \texttt{(pred}_k\ \texttt{T)),} \\
& \texttt{(symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) (cursor (pred}_k\ \texttt{T))).} \\
& \texttt{((pred}_k\ \texttt{(cursor (prev}_k\ \texttt{T))) I V).}
\end{aligned}
$$

The general rules that transfer a symbol to following time-steps are called inertia rules and are defined as follows for all $\sigma \in \Gamma$ (i.e. including $\sqcup$). Note that the inertia rules apply only to positions that differ from the one pointed to by the head.

$$
\begin{aligned}
\texttt{symbol}_\sigma\ \texttt{T X} &\leftarrow \texttt{(less\_than}_k\ \texttt{X (cursor T)), (symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) X).} \\
\texttt{symbol}_\sigma\ \texttt{T X} &\leftarrow \texttt{(less\_than}_k\ \texttt{(cursor T) X), (symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) X).}
\end{aligned}
$$

Finally the following rule concerns acceptance. (state yes equivalent to q_{accept} in the formal definition).

$$\texttt{accept} \leftarrow (\texttt{state}_{yes}\ \texttt{last}_k).$$

Thus any Turing machine that accepts or rejects its input after at most $T_k(n)(d)$ steps, can be simulated by the above Higher-order Datalog program using $\texttt{order}_k$ number representation based on d-tuples.

5. CORRECTIONS

In this chapter we discuss some faults that appear in the program reviewed in chapter 4, and propose solutions where possible.

The first issue arises in the predicates `base_succ` and `base_pred`. These predicates are defined to allow counting from 0 to $n-1$ and from $n-1$ to 0 respectively. Thus the predicates

```
base_succ I J ← (input I X J).
base_pred I J ← (input J X I).
```

on a `input` relation

$$\{(0,\sigma_1,1),\ (1,\sigma_2,2),\ \ldots,\ (n-1,\sigma_n,n)\}$$

will result in the following relations

$$\texttt{base\_succ}:\ \{(0,1),\ (1,2)\ ,\ldots,\ (n-1,n)\}$$
$$\texttt{base\_pred}:\ \{(1,0),\ (2,1)\ ,\ldots,\ (n,n-1)\}$$

which means that we can count up to and including n. If we allow negation-as-failure on first-order defined predicates a solution could be the following which does not hold for the tuples $(n-1,n)$ and $(n,n-1)$.

```
base_succ I J ← (input I X J), (not (base_last I)).
base_pred I J ← (input J X I), (not (base_last J)).
```

Another issue lies in the definition of $\texttt{base\_to\_higher}_k$. Recall that the intended purpose of this predicate is to succeed only if the base notation number equals the $\texttt{order}_k$ notation number. Thus the predicate

```
base_to_higher_k 0 N ← (equal_k N zero_k).
base_to_higher_k M N ← (input J σ M),
                       (base_to_higher_k J (pred_k N)).
```

for a given number in base notation `M`, succeeds for every number `N` in $\texttt{order}_k$ notation, less than or equal to `M` as by definition ($\texttt{pred}_k$ $\texttt{zero}_k$) equals $\texttt{zero}_k$. A correct definition is as follows.

```
base_to_higher_k 0 N ← (equal_k N zero_k).
base_to_higher_k M N ← (non_zero_k N), (input J σ M),
                       (base_to_higher_k J (pred_k N)).
```

An important issue lies with the initialization rule for $\texttt{symbol}_\sqcup$. This predicate is intended to succeed for every position `X` to the right of the last position of the input on tape. Recall its definition.

$$\begin{aligned}\texttt{symbol}_{\sqcup}\ \texttt{T X} \leftarrow\ & \texttt{(is\_zero}_k\ \texttt{T), (base\_last Y),}\\ & \texttt{(base\_to\_higher Y X).}\end{aligned}$$

Instead the predicate only succeeds if the given position `X` is the $\texttt{order}_k$ equivalent to the base number which indicates the position of the last symbol of input on the tape. For example given input $\sigma_1\sigma_2\ldots\sigma_n$, $\texttt{symbol}_{\sqcup}$ succeeds for n in $\texttt{order}_k$ notation. A correct definition is as follows.

$$\begin{aligned}\texttt{symbol}_{\sqcup}\ \texttt{T X} \leftarrow\ & \texttt{(is\_zero}_k\ \texttt{T), (base\_last Y),}\\ & \texttt{(greater\_in\_order}_k\ \texttt{Y X).}\end{aligned}$$

where we define an additional predicate $\texttt{greater\_in\_order}_k$ which succeeds if the input `N` in $\texttt{order}_k$ notation represents a greater number than `M` which is in base notation.

$$\begin{aligned}&\texttt{greater\_in\_order}_k\ \texttt{0 N} \leftarrow \texttt{(non\_zero}_k\ \texttt{N).}\\ &\texttt{greater\_in\_order}_k\ \texttt{M N} \leftarrow \texttt{(non\_zero}_k\ \texttt{N), (input J}\ \sigma\ \texttt{M),}\\ &\qquad\qquad\qquad\qquad\qquad \texttt{(greater\_in\_order}_k\ \texttt{J (pred}_k\ \texttt{N)).}\end{aligned}$$

Another issue is identified with the inertia rules. While the functionality of this predicate is correct the case where the time-step `T` given is $\texttt{zero}_k$, needs to be excluded as intuitively for time-step `T` $\texttt{zero}_k$, the position of symbols is fully defined from the initialization rules and no “transfer” can occur. Operationally this leads to an infinite loop when executed. Thus the predicate

$$\begin{aligned}&\texttt{symbol}_\sigma\ \texttt{T X} \leftarrow\ \texttt{(less\_than}_k\ \texttt{X (cursor T)), (symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) X).}\\ &\texttt{symbol}_\sigma\ \texttt{T X} \leftarrow\ \texttt{(less\_than}_k\ \texttt{(cursor T) X), (symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) X).}\end{aligned}$$

transforms to

$$\begin{aligned}\texttt{symbol}_\sigma\ \texttt{T X} \leftarrow\ & \texttt{(non\_zero}_k\ \texttt{T),}\\ & \texttt{(less\_than}_k\ \texttt{X (cursor T)), (symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) X).}\\ \texttt{symbol}_\sigma\ \texttt{T X} \leftarrow\ & \texttt{(non\_zero}_k\ \texttt{T),}\\ & \texttt{(less\_than}_k\ \texttt{(cursor T) X), (symbol}_\sigma\ \texttt{(pred}_k\ \texttt{T) X).}\end{aligned}$$

Finally since truth for a state at a given time-step `T` does not transfer to successor time-steps, we must define a predicate that, if the accepting state yes is proven true at least once, this truth is transferred to the last time-step representable in order for `accept` to succeed. We define:

$$\texttt{state}_{yes}\ \texttt{T} \leftarrow\ \texttt{(non\_zero}_k\ \texttt{T), (state}_{yes}\ \texttt{(pred}_k\ \texttt{T)).}$$

With this we conclude the main developments in this thesis.

6. CONCLUSION

We conclude this thesis with some interesting remarks on the proof we reviewed. The point of the simulation was to show that Higher-order Datalog is as expressive as exponentially time-bounded Turing machines. These Turing machines can decide all problems that lie in EXP^{k}TIME for any k and as a result so does Higher-order Datalog.

However, this does *not* necessarily imply that the simulation program runs in *exponential time* with respect to the *input*. On the contrary it can easily be seen that the program runs exponentially slower that the appropriate time bound defined by the complexity class of a problem it decides. For example consider the definition of $\texttt{pred}_k$. To calculate a bit of $(\texttt{pred}_k\ \texttt{N})$, we need to calculate numerous times, multiple bits of `N`, and this process is repeated recursively for `N` until $\texttt{zero}_k$. Thus $\texttt{pred}_k$ runs in time exponential to the size of the number it is representing which in general is already exponential with respect to the input size.

This result, from a theoretical viewpoint, is valid and the proof is correct. The simulation program that we presented was never intended to actually run in the appropriate time-bound. It was intended to show that Higher-order Datalog is rich enough to decide these problems. While this result may seem paradoxical, it is not since by definition, there exist algorithms that decide these problems in the appropriate time-bound and it can be shown that the same holds in Higher-order Datalog.

From a practical viewpoint, this program is very inefficient and, even for $\texttt{order}_1$ numbers, will almost always take exponential time to run. In [2], a non standard *memoization* technique is used to run these programs in the appropriate time-bound and thus the part of the proof that requires to show that any given program in the language we examine can be run by an algorithm that respects the resource bounds of the complexity class, is proven. This technique involves *storing* already computed *subproblems*, and using these stored results to avoid repeated solution of the same subproblems.

The same approach is used in the implementation of the simulation program in XSB. The XSB system supports a *tabling mechanism* that is primarily used to implement the well-founded semantics for logic programs. Another useful property of *tabling* is that whenever a predicate is executed and a solution is found, this solution is stored in a table. Whenever a predicate is called, the table is searched for a solution to the predicate, if none is found the predicate is executed normally. The interested reader is referenced to [12] for more information on tabling.

APPENDIX: XSB & IMPLEMENTATION

The XSB system can be downloaded from:
`https://sourceforge.net/projects/xsb/`

A very useful technical overview of XSB, including Tabled Resolution and HiLog Compilation can be found in:
`http://xsb.sourceforge.net/about.html`

For a more in-depth understanding of the system, as well as usage instructions the manual should be consulted at:
`http://xsb.sourceforge.net/manual1/manual1.pdf`

Since the program implemented in XSB is quite lengthy, in order to keep this thesis compact and readable the reader interested in using the programs can follow the links below to download the files.

Before loading the examples the following file, which contains basic predicate definitions (numbers etc.) should be loaded first:
`http://cgi.di.uoa.gr/~sdi1400169/power.P`

The first example decides the language $L = \{a^n b^n \mid n \geq 0\}$:
`http://cgi.di.uoa.gr/~sdi1400169/TManbn.P`

The second example decides the 3-SAT problem:
`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem#3-satisfiability`
`http://cgi.di.uoa.gr/~sdi1400169/TMsat.P`

The code supports a maximum of three variables used (`x, y, z`). A sample input is:

```
input(0, '?', 1). input(6, ')', 7).
input(1, '(', 2). input(7, '#', 8).
input(2, 'x', 3). input(8, 'F', 9).
input(3, ')', 4). input(9, 'F', 10).
input(4, '(', 5). input(10, '#', 11).
input(5, 'y', 6).
```

Which is equivalent to the CNF formula $(x) \wedge (y)$. The input should always start with the '?' symbol and end with '`#F…#`' depending on the number of variables used.

A negated variable $(\neg x)$ is encoded as such:

```
input(0, '(', 1).
input(1, '-', 2).
input(2, 'x', 3).
input(3, ')', 4).
```

As already stated the simulation has very high complexity even with tabling. So the above program can only be used for small inputs. For any inquiries concerning the implementation feel free to contact me at: **vagelis.protopapas@gmail.com**

REFERENCES

[1] Matthias Felleisen. “On the Expressive Power of Programming Languages”. In: *Science of Computer Programming*. Springer-Verlag, 1990, pp. 134–151.

[2] Neil D. Jones. *The Expressive Power of Higher-order Types or, Life without CONS*. 2001.

[3] Christos H. Papadimitriou. “A note on the expressive power of Prolog”. In: *Bulletin of the EATCS* 26 (1985), pp. 21–22.

[4] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Jan. 2009.

[5] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[6] Weidong Chen, Michael Kifer, and David S. Warren. “HiLog: A foundation for higher-order logic programming”. In: *The Journal of Logic Programming* 15.3 (1993), pp. 187–230. ISSN: 0743-1066.

[7] Gopalan Nadathur and Dale Miller. “An Overview of Lambda-Prolog”. In: (June 1988), pp. 810–827.

[8] William W. Wadge. “Higher-Order Horn Logic Programming”. In: *ISLP*. 1991.

[9] Angelos Charalambidis et al. “Extensional Higher-Order Logic Programming”. In: (2011).

[10] Vassilis Kountouriotis, Panos Rondogiannis, and William W. Wadge. “Extensional Higher-Order Datalog ?” In: 2005.

[11] Panos Rondogiannis and William W. Wadge. “Minimum Model Semantics for Logic Programs with Negation-as-failure”. In: *ACM Trans. Comput. Logic* 6.2 (Apr. 2005), pp. 441–467. ISSN: 1529-3785.

[12] Weidong Chen and David S. Warren. “Tabled Evaluation with Delaying for General Logic Programs”. In: *J. ACM* 43.1 (Jan. 1996), pp. 20–74. ISSN: 0004-5411.