EXTENDING LOGIC PROGRAMMING WITH COINDUCTION

APPROVED BY SUPERVISORY COMMITTEE:

Gopal Gupta, Chair

Dung T. Huynh

R. Chandrasekaran

Neeraj Mittal

To my wife

EXTENDING LOGIC PROGRAMMING WITH COINDUCTION

by

LUKE EVANS SIMON, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2006

ACKNOWLEDGEMENTS

I would like to express my gratitude for my adviser Gopal Gupta, whose guidance shaped this research. I would like to thank my colleague Ajay Mallya, for useful discussions and suggestions. I would like to thank my family for motivating me to see my graduate studies through to completion. Finally, I thank my wife Atussa, for her undying love and support.

June 2006

EXTENDING LOGIC PROGRAMMING WITH COINDUCTION

Publication No. _____

Luke Evans Simon, Ph.D.
The University of Texas at Dallas, 2006

Supervising Professor: Gopal Gupta

Traditional logic programming, with its minimal Herbrand model semantics, is useful for declaratively defining finite data structures and properties. A program in traditional logic programming defines a set of inference rules that can be used to automatically construct proofs of various logical statements. The fact that logic programming also has a goal directed, top-down operational semantics, means that these proofs can efficiently be constructed by "executing" the logical statement that is to be proved. However, since traditional logic programming's declarative semantics is given in terms of a least fixed-point, that is, since logic programming's semantics is inductive, it is impossible to directly reason about infinite objects and properties. In programming language terms, this means that the language cannot make use of infinite data structures and corecursion. The contribution of this dissertation is the extension of traditional logic programming with coinduction, by invoking the principle of duality on the declarative semantics of traditional logic programming and by developing an efficient top-down, goal-directed procedure based on the principle of coinduction, for deciding inclusion of a logical statement in the greatest fixed-point model. This gives rise to a new field of programming languages referred to by this author as "co-logic programming".

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Traditional logic programming is actually inductive logic programming[1]: its minimal Herbrand model semantics is useful for declaratively defining finite data structures and properties. A program in traditional logic programming effectively defines a set of inference rules that can be used to automatically construct proofs of various logical statements. The fact that logic programming also has a goal directed, top-down operational semantics (SLD) means that these proofs can efficiently be constructed by "executing" the logical statement that is to be proved. However, since traditional logic programming's declarative semantics is given in terms of a least fixed-point, that is, since logic programming's semantics is inductive, it is impossible to directly reason about infinite objects and properties.

The traditional declarative and operational semantics for logic programming is inadequate for various programming practices such as programming with infinite data structures and corecursion [2]. While such programs are theoretically interesting, their practical applications include improved modularization of programs as seen in lazy functional programming languages [3], rational terms, and applications to model checking [4, 5]. For example, we would like programs such as the following program, which describes infinite binary streams, to be semantically meaningful and finitely derivable.

[1]The name "inductive logic programming" used in this dissertation has no relation to field of using logic programming based systems for learning new inference rules [1] that generalize given sets of facts. Instead, this dissertation uses "inductive logic programming" to refer to SLD resolution based logic programming, which has a least fixed-point or "inductive" declarative semantics.

```
bit(0).
bit(1).
bitstream([H | T]) :- bit(H), bitstream(T).
```

Furthermore, we would like queries such as the following to return a positive answer in a finite amount of time.

```
| ?- X = [0, 1, 1, 0 | X], bitstream(X).
```

However, aside from the `bit` predicate, the least fixed-point semantics of the above program is null, and no finite *SLD* derivation exists for the query. Hence the problems are two-fold. The Herbrand universe does not allow for infinite terms such as `X` and the least Herbrand model does not allow for infinite proofs, such as a proof of `bitstream(X)`. However, the traditional declarative semantics of logic programming can be extended in order to give declarative semantics to such infinite structures and properties, as seen in numerous accounts of rational terms and infinite derivations [6, 7].

Furthermore, the operational semantics must be extended, so as to be able to finitely represent an otherwise infinite derivation. This dissertation proposes such a method which is based on synthesizing a coinductive hypothesis. This work is based on the author's previous work, as well as the work of colleagues [8, 9, 10, 4, 5].

Hence, the contribution of this dissertation is the extension of traditional logic programming with coinduction, which is first accomplished by invoking the principle of duality on the declarative semantics of traditional logic programming, and more importantly, a new sound operational semantics is created, which uses a unique coinductive hypothesis rule for deciding inclusion of a logical statement in the greatest fixed-point of a given *coinductive logic program* in a top-down goal-directed manner. Described in further

detail in chapter 3, this gives rise to the category of programming languages referred to by this author as *coinductive logic programming*. Coinductive logic programming allows for logic programming with infinite data structures and corecursion, that is, coinductive logic programming can directly reason about formal logical statements regarding infinite objects and properties. This is due to the fact that the declarative semantics of coinductive logic programming allows for a proof of a logical statement to be infinite, in the presence of possibility infinite objects.

In the coinductive logic programming paradigm the declarative semantics of the predicate `bitstream/1` above is given in terms of the *co-Herbrand universe, co-Herbrand base, and maximal models*. The operational semantics is given in terms of the *coinductive hypothesis rule* which states that during execution, if the current resolvent R contains a call C' that unifies with a call C encountered earlier, then the call C' succeeds; the new resolvent is $\theta(R')$ where θ is the most general unifier of C and C' with terms ranging over the infinitary Herbrand universe, and R' is obtained by deleting C' from R. With this extension a clause such as `p([1|T]) :- p(T)` and the query `p(Y)` will produce an infinite answer `Y = [1|Y]`.

Sometimes it is desirable to restrict consideration of a logical statement to finite objects, such as finite lists or finite trees. In addition, it may also be desirable to restrict proofs of a certain property to be finite in size. In other words, just as it is useful to have coinductive logic programming, it is also useful to have inductive logic programming. Even more importantly, there are cases when both forms of logic programming are simultaneously useful. For this reason, coinductive logic programming and inductive logic programming are combined in chapter 4, so that the programmer can annotate predicates as either inductive or coinductive, and inductive predicates can invoke coinductive predicates and *visa-versa*, with the only restriction being that no cycles through alternating

induction and coinduction are allowed. In other words, an inductive predicate and a coinductive predicate cannot be mutually recursive.

This dissertation refers to the aforementioned combination of traditional and coinductive logic programming as *co-logic programming*. Co-logic programming is a natural generalization of traditional logic programming and coinductive logic programming, which in turn generalizes other extensions of logic programming, such as rational trees, lazy predicates, and concurrent perpetual predicates. The declarative semantics for co-logic programming is defined as an alternating fixed-point model, and a corresponding top-down, goal-directed operational semantics is provided in terms of alternating *SLD* and co-*SLD* semantics. The restriction preventing cycles through induction and coinduction, here referred to as "the stratification restriction", is a syntactic restriction that is easily enforced via a compile-time static analysis of the source code, and the operational semantics is easily implemented using a top-down, hypothesis-first, left-most, depth-first backtracking search.

The practical applications of this new logic programming paradigm range from static analysis to artificial intelligence to web services. The applications of co-logic programming are discussed in chapters 5 through 7.

Co-logic programming can be further extended by applying the notion of tabling to programs' inductive predicates, so that their operational behavior more closely matches their declarative semantics. Tabling inductive predicates can be used to minimize recomputation, but more importantly, it is used to improve the termination properties of inductive predicates in a manner similar to the coinductive hypothesis rule used for coinductive predicates. This extension, called *tabled co-logic programming*, is mentioned in chapter 8, along with other future work regarding co-logic programming.

CHAPTER 2
BACKGROUND

We begin by covering the requisite background concepts of induction, coinduction, fixed-points, recursion, and corecursion in section 2.1.1. These generic mathematical concepts are used, explicitly and implicitly, throughout this entire dissertation, starting with the brief introduction to traditional logic programming in section 2.2. Finally, this chapter ends with a discussion of work related to the extension of logic programming with coinduction.

2.1 Mathematical Concepts

The dual form of induction, that is, coinduction, has been known to mathematicians for many years in the fields of universal algebra and category theory. However, the use of formal coinductive proof techniques in computer science is a relatively new trend starting with work on process algebras [11], concurrency [12], and programming language semantics [13].

Following the account given in *Types and Programming Languages* [14], we briefly review the set theoretic notions of induction and coinduction, which are defined in terms of monotonic functions on sets and least and greatest fixed-points. For the remaining discussion, it is assumed that all objects such as elements, sets, and functions are taken from the universe of hypersets with the axiom of plenitude. Details can be found in [2, 15, 16, 17].

2.1.1 Induction and Coinduction

A naive attempt to prove a property of the natural numbers involves demonstrating the property for 0, 1, 2, ..., ad infinitum. In order for such a proof to be comprehensive, it must be infinite. However, since nobody has the time to write an infinite proof, the principle of proof by induction can be used to represent such an infinite proof in a finite form. This is precisely what the operational semantics of coinductive logic programming does as well. That is, coinductive logic programming uses the principle of proof by coinduction for representing infinite proofs or executions in a finite form. The difference between induction and coinduction will be made more obvious later.

Definition 2.1 *A function* Γ *on sets is monotonic if* $S \subseteq T$ *implies* $\Gamma(S) \subseteq \Gamma(T)$. *Such functions are called generating functions.*

Generating functions can be thought of as a definition for creating objects, such as terms and proofs. The following example demonstrates one such definition.

Example 2.1 *Let* $\Gamma_{\mathcal{N}}$ *be a function on sets:* $\Gamma_{\mathcal{N}}(S) = \{0\} \cup \{succ(x) \mid x \in S\}$.

Obviously, $\Gamma_{\mathcal{N}}$ *is a monotonic function, and intuitively, it defines the set of natural numbers, as will be demonstrated below.*

Definition 2.2 *Let* S *be a set.*

1. S is Γ-closed if $\Gamma(S) \subseteq S$; S is Γ-justified if $S \subseteq \Gamma(S)$.

2. S is a fixed-point of Γ if S is both Γ-closed and justified.

S is Γ-closed when every object created by the generator Γ is already in S. Similarly, a set S is Γ-justified when every object in S is created or justified by the generator.

One of the purposes of mathematics is to provide unambiguous means for defining concepts. Theorem 2.1 shows that a generating function Γ can be used for giving a precise definition of a set of objects in terms of the least or greatest fixed-point of Γ, as these fixed-points are guaranteed to exist, and are unique.

Theorem 2.1 *(Knaster-Tarski) Let Γ be a generating function. The least fixed-point of Γ is the intersection of all Γ-closed sets. The greatest fixed-point of Γ is the union of all Γ-justified sets.*

Proof 2.1 *This is just a reformulation of the Knaster-Tarski theorem [18].*

Since these fixed-points always exist and are unique, it is customary to define unary operators μ and ν for manifesting either of these fixed-points.

Definition 2.3 *$\mu\Gamma$ denotes the least fixed-point of Γ, and $\nu\Gamma$ denotes the greatest fixed-point of Γ.*

Example 2.2 *Let $\Gamma_{\mathcal{N}}$ be defined as in example 2.1. The definition of the natural numbers N can now be unambiguously invoked via theorem 2.1, as $N = \mu\Gamma_{\mathcal{N}}$, which is guaranteed to exist and be unique. Note that this definition is equivalent to the standard "inductive" definition of the natural numbers, which is written: Let N be the smallest set such that $0 \in N$ and if $x \in N$, then $x + 1 \in N$.*

Hence what is sometimes referred to as an inductive definition, is subsumed by definition via least fixed-point. This is further generalized by creating the dual notion of a definition by greatest fixed-point, termed a coinductive definition.

Example 2.3 *$\Gamma_{\mathcal{N}}$ from example 2.1 also unambiguously defines another set, that is, $\mathcal{N}' = \nu\Gamma_{\mathcal{N}} = N \cup \{\omega\}$, where $\omega = succ(\omega)$, that is, $\omega = succ(succ(succ(...)))$ an infinite application of succ.*

Corollary 2.1 *The principle of induction states: if S is Γ-closed, then $\mu\Gamma \subseteq S$. The principle of coinduction states: if S is Γ-justified, then $S \subseteq \nu\Gamma$.*

Definition 2.4 *Let $Q(x)$ be a property. Proof by induction demonstrates that the characteristic set $S = \{x \mid Q(x)\}$ is Γ-closed, and then invokes the principle of induction to prove that every element x of $\mu\Gamma$ has the property $Q(x)$.*

Similarly, proof by coinduction demonstrates that the characteristic set S is Γ-justified, and then invokes the principle of coinduction to prove that every element x that has property $Q(x)$ is also an element of $\nu\Gamma$.

Example 2.4 *The familiar proof by induction can be instantiated with regards to the set $\mathcal{N}$ defined in the previous example. Let $Q(x)$ be some property, and let $S = \{x \mid Q(x)\}$. In order to show that every element x in $\mathcal{N}$ has property $Q(x)$, by induction it is sufficient to show that $\Gamma_{\mathcal{N}}(S) \subseteq S$, which is equivalent to showing that $0 \in S$, and if $x \in S$, then $succ(x) \in S$.*

Like proof by induction, proof by coinduction is used in many aspects of computer science, e.g., bisimilarity proofs for process algebras such as the π-calculus [19]. Section 3.2.4 and section 4.2.3 demonstrate other examples of proof by coinduction in the soundness proofs for the operational semantics of coinductive logic programming and co-logic programming respectively.

2.1.2 Recursion and Corecursion

In the previous section, we began with the analogy of a naive proof. Similarly, one can make a futile attempt to define a mapping of the natural numbers into another set (possibly itself) by defining the mapping for 0, 1, 2, ..., ad infinitum. However, the definition of the function corresponding to this mapping will always be incomplete, as the definition is infinite in size. Since nobody has the time to write an infinite definition, the notion of recursion can be used for mapping a possibly infinite least fixed-point into some other set, using a finite definition. It also turns out that recursion itself can be used as an incomplete definition of computation. Of course, this is nothing new to a computer scientist. However, the dual notion, corecursion, is not as familiar. While recursion is used to map a least fixed-point to some other set, corecursion is used to map from a set into a greatest fixed-point.

A formal account of recursion and corecursion is out of the scope of this dissertation, and therefore the reader is referred to the wonderfully detailed account of Barwise and Moss [2]. In short, recursion works by deconstructing an element of a least fixed-point, descending from top to bottom, while corecursion constructs an element of a greatest fixed-point by ascending on the result, creating a larger and larger object. Hence recursion deconstructs (i.e., analyzes) elements of a least fixed-point, while corecursion constructs an element of a greatest fixed-point. Furthermore, by definition, recursion must terminate, while corecursion need not terminate. Extending logic programming with coinduction, as described in this dissertation, allows for both recursion and corecursion in the logic programming paradigm.

The `bitstream` example from the introduction is an example of a corecursive predicate.

Definition	Proof	Mapping
least fixed-point	induction	recursion
greatest fixed-point	coinduction	corecursion

Table 2.1. Dual Mathematical Techniques

```
bitstream([H | T]) :- bit(H), bitstream(T).
```

Notice how it does not have a base case that is always found in the use of recursion? This predicate constructs an infinite list by ascending on the one and only argument to the predicate. Therefore it is necessary to extend logic programming with coinduction, if predicates such as `bitstream` are to have their intended meaning.

Table 2.1 summarizes the general mathematical concepts that will be used in this dissertation. Finite or infinite sets of objects can be formally and finitely defined as a least or greatest fixed-point, which are guaranteed to exist. Using the notion of a characteristic set, a property can also be defined via least or greatest fixed-point, as the set of objects that have said property. Given objects and properties defined in such a way, finite proofs that objects have certain properties can be given using proof by induction or coinduction. Finally, mappings between objects, for example, a mapping that takes a natural number and returns an infinite list, can be defined using recursion or corecursion.

2.2 Logic Programming

Declarative programming languages as opposed to imperative or procedural programming languages, take a decidedly high-level approach to programming language design [20]. The goal of an imperative language is to allow a programmer to specify what is wanted from the desired software system, such that the language's compiler or interpreter are left with the task of figuring out how to accomplish or realize the high-level specification. The

declarative approach to programming languages has many advantages. Declarative languages are high-level, allowing a typically direct specification of the problem to be solved by the computer, and programs written in a declarative language are more amenable to formal mathematical reasoning.

There are numerous forms of declarative programming [20]. For example, functional programming has specifications of programs given in terms of formal mathematical functions. Meanwhile, logic programming has specifications of programs given in terms of some kind of formal mathematical logic [21]. This dissertation restricts consideration to Prolog-style logic programming based on Horn logic, which is a fragment of classical logic that can be efficiently executed on a computer. The following is a brief introduction of the abstract syntax and semantics of traditional logic programming, that is Horn logic or definite logic.

2.2.1 Syntax

In the following, it is important to distinguish between an idealized class of objects and the syntactic restriction of said objects. Elements of syntax are necessarily finite, while many of the semantic objects used by the coinductive extensions of logic programming defined later are infinite. It is assumed that there is an enumerable set of variables, an enumerable set of constants, and for all natural numbers n, there are an enumerable set of function and predicate symbols of arity n.

In logic programming, objects or data structures are high-level entities known as terms. A term is simply an expression that denotes a tree, as the following definition makes apparent.

Definition 2.5 *The set of terms is* $\nu\Gamma$ *and the set of syntactic terms is* $\mu\Gamma$*, where* $t \in \Gamma(S)$ *whenever one of the following is true:*

1. t is a variable.

2. t is a constant.

3. $t = f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n \in S$ and f is a function symbol of arity n.

Note that terms can be finite or infinite in size, while syntactic terms must be finite.

Definition 2.6 *A ground term is a term that does not contain variables. A term is said to be ground if it is a ground term.*

Variables are typically written as words that start with a capital letter. For example, X and Y are variables. Constants and function symbols are simply mutually distinct tokens, representing an object. Constants are atomic objects, while functions of arity n are intended to be applied to n arguments. For example, 0 and 1 are constants. More generally, constants are written as words that start with a lower-case letter. Hence *hot* and *cold* can be considered constants too. Non-trivial terms are constructed by applying function symbols (which have a syntax identical to constants) to terms in a way that respects the arity of the function symbol. For example, $succ(0)$ is a term that applies a function symbol *succ* of arity one to the constant 0. Note that such a term is ground, while a term such as $succ(X)$ is not ground.

As will be formally defined later, syntactically distinct ground terms denote distinct objects. This allows for a direct means of specifying new objects as the following examples demonstrate.

Example 2.5 *Natural numbers can be constructed by using just one function symbol succ and one constant zero. Following Peano's Arithmetic, we can take* $0 = zero$, $1 = succ(zero)$, $2 = succ(succ(zero))$, *and so on.*

Example 2.6 *Lists can also be constructed using just one function symbol node and one constant nil. In a manner similar to the standard linked node representation, lists of the form* $[X_1, X_2, \ldots, X_n]$ *can be represented as* $node(X_1, node(X_2, \ldots node(X_n, nil)))$. *Combined with the previous example, the two element list* $[0, 1]$ *can be represented* $node(zero, node(succ(zero), nil))$.

Objects alone are not enough to specify a program, and therefore logic programming provides predicates. A predicate is defined by its inference rules, also known as definite clauses, while the invocation of a predicate is known as an atom.

Definition 2.7 *An atom is an expression of the form* $p(t_1, \ldots, t_n)$, *where* p *is a predicate symbol of arity* n *and* $t_1, \ldots, t_n$ *are terms.*

As will be formally defined later, an atom $p(t_1, \ldots, t_n)$ denotes the logical statement that the tuple $(t_1, \ldots, t_n)$ is in the mathematical relation p, that is, the property p holds for the objects $t_1, \ldots, t_n$.

Definition 2.8 *A clause is a logical inference rule of the form* $C \leftarrow D_1, \ldots, D_n$ *where* $C, D_1, \ldots, D_n$ *are atoms. Furthermore,* C *is called the head of the clause and* $D_1, \ldots, D_n$ *is called the body of the clause.*

Intuitively, a clause $C \leftarrow D_1, \ldots, D_n$ can be interpreted as an implication, where $D_1, \ldots,$ and D_n implies C. Throughout this dissertation, "clause" and "inference rule" will refer to the same thing.

Definition 2.9 *An atom or clause is said to be syntactic if it only contains syntactic terms. Similarly, an atom, clause, or program is said to be ground if it only contains*

ground terms. A term, atom, or clause is an instance of another term, atom, or clause respectively, when the instance is obtained from the original by substituting terms for variables in the original.

Definition 2.10 *A definite program is a finite set of syntactic definite clauses.*

Intuitively, a definite program can be interpreted as the disjunction of the interpretations of each individual clause. Note that a definite program is a finite object. The following definition of a traditional logic program marries syntax and semantics.

Definition 2.11 *A traditional logic program is a definite program with the declarative semantics defined in section 2.2.2.1.*

This section has defined the abstract or mathematical syntax of a logic program. However, logic programs, such as Prolog programs, are typically written in the form of ASCII text files. The details of the concrete syntax of logic programs varies in minor ways, which are unimportant to the material covered in this dissertation. However, code examples will be given using a concrete Prolog syntax compatible with the SICStus Prolog system [22]. Details regarding the distinction between abstract and concrete syntax for logic programming can be found in [23], and the details of the SICStus Prolog syntax can be found in the SICStus Prolog user manual [22]. A short incomplete description of the concrete syntax is the following:

1. Constants, function symbols, and predicate symbols are tokens beginning with a lowercase letter from the Roman alphabet.

2. Variables are tokens beginning with an uppercase letter.

3. A definite clause $C \leftarrow D_1, \ldots, D_n$ is written "`C :- D1, ...,Dn`". Furthermore, when $n = 0$, the clause can be written "`C.`"

Example 2.7 *A predicate can be defined for determining if a given term is a natural number, as follows:*

```
number( zero ).
number( succ( X ) ) :- number( X ).
```

Intuitively, this program reads: "Y is a number when $Y = zero$ or $Y = succ(X)$, such that X is a number.

Example 2.8 *Similarly, a predicate can be defined for determining if a given term is a list:*

```
list( nil ).
list( node( X, L ) ) :- list( L ).
```

This program states that a list is either empty, that is, nil, or a list consists of a node containing an element and a list.

2.2.2 Semantics

A programming language is a combination of syntax and semantics, and a declarative programming language typically has two kinds of semantics: declarative semantics and operational semantics. The declarative semantics is the denotational or mathematical description of the typically formal set theoretic objects that are described by expressions in the language. The operational semantics, on the other hand, formally describes how

expressions in the language can be executed. In other words, declarative semantics describe "what it is", while operational semantics describe "how to do it". So the declarative semantics can be considered the standard benchmark for determining the meaning of statements in the language, while the operational semantics is merely just a means to an end.

2.2.2.1 Declarative Semantics

A logic program is executed by presenting the logic programming interpreter with a given program and a query. A query is simply a conjunction of atoms. The intent is to have the interpreter determine if the query is satisfiable according to the given logic program. Hence the declarative semantics of logic programming is concerned with determining when an atom is true, as satisfiability can then be reduced to finding a substitution for the query that makes every atom in the query true.

The following is a brief account of the minimal Herbrand model semantics for inductive logic programming. Intuitively, this assigns meaning to a clause $C \leftarrow D_1, \ldots, D_n$ as an inference rule that can be used for deriving a conclusion C from premises $D_1, \ldots, D_n$. It is also possible to apply a formal denotational semantics to logic programming by interpreting the "$\leftarrow$" symbol as classical mathematical logic's implication and the "," symbol as classical mathematical logic's conjunction. Both approaches coincide for traditional logic programming, but in order to extend logic programming with coinduction, it is more natural to use the minimal Herbrand model approach to semantics as opposed to the classical logic approach. Details of both approaches to semantics can be found in classic texts of Lloyd [7] and Sterling et al. [23].

Definition 2.12 *Let P be a logic program. Let $A(P)$ be the set of constants in P, and let $F_n(P)$ denote the set of function symbols of arity n in P. The Herbrand universe of*

P, is denoted $U(P) = \mu\Phi_P$, where

$$\Phi_P(S) = A(P) \cup \{f(t_1, \ldots, t_n) \mid f \in F_n(P) \wedge t_1, \ldots, t_n \in S\}$$

For technical reasons, we assume that there is at least one constant and one function symbol. This causes the universe to be non-empty. The Herbrand universe is the set of ground terms that can be constructed from the constants and functions in the program. Since a least fixed-point is used to define the Herbrand universe, only finite terms are included.

Intuitively, the Herbrand universe assigns meaning to a ground term t as a mathematical tree consisting of a direct representation of the term t. Hence when two ground terms are syntactically distinct, they are also semantically distinct. This does not apply to terms that are not ground. A term t that is not ground, that is, which contains variables $X_1, \ldots, X_n$, can be thought of as a function $f(t_1, \ldots, t_n) = t[X_1 := t_1, \ldots, X_n := t_n]$, where $t_1, \ldots, t_n$ are ground and $t[X_1 := t_1, \ldots, X_n := t_n]$ is the term t with variables $X_1, \ldots, X_n$ replaced with terms $t_1, \ldots, t_n$ respectively.

Definition 2.13 *Let P be a traditional logic program. The Herbrand base, written $B(P)$, is the set of all ground atoms that can be formed from the predicate symbols in P and the elements of $U(P)$. Also, let $G(P)$ be the set of ground clauses $C \leftarrow D_1, \ldots, D_n$ that are a ground instance of some clause of P such that $C, D_1, \ldots, D_n \in B(P)$.*

Finally we can give the formal definition of a traditional logic program P's semantics as the minimal Herbrand model of P.

Definition 2.14 *A Herbrand model of a traditional logic program P is a fixed-point of*

$$T_P(S) = \{C \mid C \leftarrow D_1, \ldots, D_n \in G(P) \wedge D_1, \ldots, D_n \in S\}$$

The minimal Herbrand model of an inductive program P, denoted $M^{in}(P)$, is the least fixed-point of T_P, which exists and is unique according to theorem 2.1. Hence $M^{in}(P)$ is taken to be the declarative semantics of a traditional logic program P.

Truth is defined in terms of inclusion in the model.

Definition 2.15 *An atom A is true in a traditional logic program P if and only the set of all groundings of A, with substitutions ranging over the $U(P)$, is a subset of $M^{in}(P)$.*

Example 2.9 *Let P_1 be the following inductive logic program.*

```
from(N, [N|T]) :- from(s(N), T).
| ?- from(0, _).
```

Note that this example uses the shorthand notation for lists, such as $[H|T]$ and $[t_1, \ldots, t_n]$ [23]. The inductive semantics are derived as follows. The Herbrand Universe is $U(P_1) \supseteq N \cup L$ where $N = \{0, s(0), s(s(0)), \ldots\}$ is the set of natural numbers and L is the set of all finite lists of elements in N and L. However, the minimal Herbrand model $M^{in}(P_1) = \emptyset$ is the meaning of the program. Therefore the query $from(0, [0, s(0), s(s(0)), \ldots]) \notin M^{in}(P_1)$, and hence it is not true. We would like the predicate to denote that its second argument is a list that starts counting with the number denoted by its first argument. However, this necessitates that the second argument is an infinite list, which simply does not exist in the Herbrand universe.

Example 2.10 *Recall the bit stream example from the introduction.*

```
bit(0).
bit(1).
bitstream([H | T]) :- bit(H), bitstream(T).
```

The Herbrand universe only contains 0, 1, *finite lists of these two digits, and finite lists of elements from the universe itself. Because the* `bitstream` *predicate requires its argument to be an infinite list of binary digits, the minimal Herbrand model of this program only contains* $bit(0)$ *and* $bit(1)$*. Hence the only logic statements that are true, with regards to this program, are* $bit(0)$*,* $bit(1)$*, and* $bit(X)$ *for all variables* X*.*

2.2.2.2 Operational Semantics

The minimal Herbrand model of a traditional logic program tells us what the program means, but it doesn't tell us how to execute the program. Hence the language also needs an operational semantics. The following account of inductive logic programming's operational semantics is similar to that of Lloyd [7]. The semantics, called *SLD*, which stands for "**S**election function with **L**inear resolution for **D**efinite programs", can be thought of as a non-deterministic state transition system corresponding to the high-level state transitions of an execution of an inductive logic programming query.

All definitions in this section and the following section 2.2.2.3 are specific to *SLD*. Hence the definition of a state, a transition, etc, will be redefined in later sections for distinct operational semantics that extend *SLD*.

Definition 2.16 *A state is a multi-set of syntactic atoms.*

Intuitively, a state in *SLD* semantics is the set of logical statements that remain to be proven true. A transition from one state to another involves unifying the head of an instance of a clause with one of the atoms of the state.

Unification is a general problem involving solving equations between syntactic terms. A solution is a substitution of terms for variables, such that all equations are vacuously true. So in this case, the head of an inference rule is equated with an atom taken from the state, and unification is used to confirm that such an equation is solvable, as well as to yield a solution. So a transition is only valid if the two terms do in fact unify.

Unification was originally devised by Robinson [24, 25], though his first algorithm runs in exponential time in the worst case. Shortly there after, algorithms that run in quasi-linear time, which are based on Tarjan's union-find algorithm [26], were independently developed by Martelli et al. [27] and by Huet [28, 29]. However, the algorithm of Martelli et al. only works with acyclic or finite terms, while Huet's algorithm works with finite, infinite, acyclic, and cyclic terms. Paterson et al. later introduced an algorithm that can perform unification for acyclic terms in truly linear time [30, 31].

However, the differences in asymptotic complexities do not accurately reflect real world performance, especially considering the fact that the kinds of terms unified varies from problem to problem [32]. For this reason, logic programming implementations tend to use a unification method that is not even quasi-linear, as in the common case it performs better than algorithms such as Huet's, which are quasi-linear in the worst-case. Knight [33], Baader and Siekmann [34], and Baader and Snyder [35], provide further details regarding unification's history, theory, and algorithms.

Definition 2.17 *In inductive logic program P, let Θ be the most general unifier of A_n and A, such that $C = A \leftarrow B_1, \ldots, B_m$ is a clause in program P. Then in P's SLD state transition system, a state $\{A_1, \ldots, A_n\}$ transitions to a state $\Theta(\{A_1, \ldots, A_{n-1}, B_1, \ldots, B_m\})$ by a transition labelled (C, Θ), where $\Theta(S)$ is the multi-set obtained from S by applying the substitution Θ to every element of S. Furthermore, the most general unifier is restricted to range over the Herbrand universe.*

Definition 2.18 *A query Q is a multi-set of syntactic atoms. A query is said to be true in program P, whenever each $A \in Q$ is true in P.*

Definition 2.19 *In inductive logic program P, given query $A_1, \ldots, A_n$, the start state of P's SLD state transition system is the state $\{A_1, \ldots, A_n\}$. The accepting state is the state $\emptyset$.*

In other words, an execution starts with a query, then it proceeds by trying to prove each atomic logical statement in the current state. Finally, it successfully ends when no remaining outstanding logical statements still need to be proved.

Definition 2.20 *A derivation of a query in program P is a sequence of transitions $(C_1, \Theta_1), \ldots, (C_n, \Theta_n)$ in P's SLD state transition system that forms a path from the start state to some other state in the system. A derivation is successful if it terminates in the accepting state.*

A derivation is simply a formalization of a query's execution trace.

Definition 2.21 *A query Q is successful in program P, whenever Q has an accepting derivation $(C_1, \Theta_1), \ldots, (C_n, \Theta_n)$. When restricted to the variables occurring in Q, the composition of the substitutions $\Theta_1 \ldots \Theta_n$ is called the computed answer for Q in program P.*

So the operational semantics is quite straightforward. Given a query, execution proceeds by repeatedly trying each applicable inference rule until there are no unproven logical statements. The computed answer is simply the substitutions for variables in the original query.

As mentioned, determining if an inference rule is applicable to an atom involves unifying the head of the rule with the atomic logical statement. Calculating the most general unifier of two atoms can be accomplished in time linear in the size of the two atoms [30]. Hence the definition of *SLD* is a satisfactory account of the execution of a query. The only hidden component is the need to use a backtracking search, so as to be able to find a path from the start state to the accepting state, as not every sequence of rule applications is successful. Traditionally, this is accomplished using a left-most, depth-first search through the state transition system [36, 7, 23].

Example 2.11 *Consider the following logic program:*

```
number( zero ).
number( succ( X ) ) :- number( X ).

lessThanEq(0, Y) :- number(Y).
lessThanEq(succ(X), succ(Y)) :- lessThanEq(X, Y).
```

The number predicate is the same as in example 2.7. The `lessThanEq(X, Y)` *predicate defines the traditional* $X \leq Y$ *relation between natural numbers. It states that* 0 *is less than or equal to* Y*, if* Y *is a natural number, and* X *is less than or equal to* Y *implies* $X + 1$ *is less than or equal to* $Y + 1$*. Lets observe the* SLD *execution of the query corresponding to* $2 \leq 3$ *using a left-most, depth-first search strategy. We start with the state containing* `lessThanEq(succ(succ(0)), succ(succ(succ(0))))`*, and so we have no other choice but to try to apply a rule to this logical statement. The head of the first inference rule,* `number( zero )`*, does not unify with the statement, so we keep trying the next inference rule until a rule can be applied. It turns out that the last rule in the*

program is the only inference rule that can be applied to this statement, yielding the next state, which contains ***lessThanEq(succ(0), succ(succ(0)))****. This process repeats as demonstrated below:*

1. `{lessThanEq(succ(0), succ(succ(0)))}`
2. `{lessThanEq(0, succ(0))}`
3. `{number(succ(0))}`
4. `{number(0)}`
5. $\emptyset$

The final state is the empty multi-set, that is, the state with no outstanding unproven logical statements. So according to the program, $2 \leq 3$ is in fact true, which was the intent of the program. It is also important to see how *SLD* determines when a logical statement is false. So consider the execution of the query corresponding to $3 \leq 2$, that is, consider the execution of `lessThanEq(succ(succ(succ(0))), succ(succ(0)))`. This proceeds as follows:

1. `{lessThanEq(succ(succ(succ(0))), succ(succ(0)))}`
2. `{lessThanEq(succ(succ(0)), succ(0))}`
3. `{lessThanEq(succ(0), 0)}`

So far, at every step of the computation, there was no other choice but to apply the fourth inference rule, which has resulted in transitioning to the state containing `lessThanEq(succ(0), 0)`, which corresponds to the statement $1 \leq 0$. At this point,

SLD notices that there is a problem, because there are no applicable inference rules. Hence the logical statement `lessThanEq(succ(0), 0)` is false, and because there were no other options encountered during the execution of the original statement, it must be that the original statement is also false.

2.2.2.3 Correctness

A declarative programming language's canonical meaning comes from its declarative semantics, but its practical implementation is based on its operational semantics. This means that it is necessary to establish a correspondence between the declarative and operational semantics. Since the declarative semantics is considered the benchmark, this correspondence is described in terms of the soundness and completeness of the operational semantics. Soundness requires that the operational semantics only computes correct results, and completeness requires that for a satisfiable query, the operational semantics is capable of deriving that the query is satisfiable.

Theorem 2.2 *(Soundness) If query Q is successful in inductive logic program P with computed answer Θ, then $\Theta(Q)$ is true in P.*

Proof 2.2 *Let $(C_1, \Theta_1), \ldots, (C_n, \Theta_n)$ be the successful derivation of Q in program P, and let $\Theta' = \Theta_1 \ldots \Theta_n$. The proof proceeds by induction on the length of the derivation, that is, by induction on n.*

If $n = 0$, then $Q = \emptyset$, which is vacuously true. Now, consider the case when $n > 0$. So Q is of the form $\{A_1, \ldots, A_m\}$ and transitions to a state Q' of the form $\Theta_1(\{A_1, \ldots, A_{m-1}, B_1, \ldots, B_l\})$, where $C_1 = A \leftarrow B_1, \ldots, B_l$. By induction $[\Theta_2 \ldots \Theta_n](Q')$ is true in P. So all groundings of $[\Theta_2 \ldots \Theta_n](Q')$ are included in $M^{in}(P)$. This implies that all groundings of $\Theta(Q)$ are included in $T_P(M^{in}(P))$. The fact that $M^{in}(P)$ is the

least fixed-point of T_P implies $T_P(M^{in}(P)) = M^{in}(P)$. So it can be concluded that all groundings of $\Theta(Q)$ are included in $M^{in}(P)$. Hence $\Theta(Q)$ is true in program P.

Theorem 2.3 *(Completeness) If atom $A \in M^{in}(P)$, then A has a successful derivation in inductive logic program P.*

Proof 2.3 *The proof proceeds by set theoretic induction on $M^{in}(P)$, which is possible because $M^{in}(P) = \mu T_P$. Implicitly we are using the principle of induction to show that $\mu T_P \subseteq S$, where S is the set of all atoms with successful derivations in P. There are two cases to consider.*

In the base case, $A \leftarrow$ is in $G(P)$, the set of ground instances of the clauses of P. Obviously there is a derivation that starts with state $\{A\}$ and transitions to the state $\emptyset$ by applying the prototype clause $A' \leftarrow$ of which $A \leftarrow$ is a grounding. So A immediately has a successful derivation in P.

In the inductive case, $A \leftarrow D_1, \ldots, D_n \in G(P)$ and $D_1, \ldots, D_n \in M^{in}(P)$. By induction, $D_1, \ldots, D_n$ each have a successful derivation in P. Hence there is a successful derivation starting with state $\{D_1, \ldots, D_n\}$. Since state $\{A\}$ transitions to state $\{D'_1, \ldots, D'_n\}$ of which $\{D_1, \ldots, D_n\}$ is an instance, it is also true that A has a successful derivation in inductive logic program P.

2.3 Related Work

Most of the work regarding coinduction and logic programming has been focused on allowing for infinite data structures in logic programming, or it has dealt with mathematically describing infinite derivations. However, these stop short of providing both a declarative semantics as well as finite derivations for atoms that have infinite idealized proofs. Logic

programming with rational trees [37, 38, 39, 40, 41, 6, 42] allows for finite terms as well as infinite terms that are rational trees, that is, terms that have finitely many distinct subterms. Coinductive logic programming as defined in chapter 3, on the other hand, allows for finite terms, rational infinite terms, but unlike logic programming with rational trees, the declarative semantics of coinductive logic programming also allows for irrational infinite terms. Furthermore, the declarative semantics of logic programming with rational trees corresponds to the minimal co-Herbrand model. On the other hand, coinductive logic programming's declarative semantics is the maximal co-Herbrand model. Also, the operational semantics of logic programming with rational trees is simply *SLD* extended with rational term unification, while the operational semantics of coinductive logic programming corresponds to *SLD* only via the fact that both are implicitly defined in terms of state transition. Thus, logic programming with rational trees does not allow for finite derivations of atoms that have infinite idealized proofs, while coinductive logic programming does. Finally, logic programming with rational trees can *only* create infinite terms via unification (without occurs check), while coinductive logic programming can create infinite terms via unification (without occurs check) *as well as via user-defined corecursive predicates*, as demonstrated by the bit stream example in the introduction. On top of that, co-logic programming as defined in chapter 4 allows for the use of coinduction to be controlled by the programmer. Such flexibility simply isn't possible with the aforementioned previous work.

It is also well known that atoms with infinite *SLD* derivations are contained in the maximal model [43, 6, 44, 7]. However, the novel contribution of co-logic programming is its operational semantics' use of memoization for synthesizing a coinductive hypothesis, which allows for the invocation of the coinductive hypothesis rule for recognizing atoms that have an infinite idealized proof. For example, the work of [43, 6, 44, 7] doesn't

provide an effective means, i.e., an operational semantics, for answering the bit stream query in the introduction. In their operational semantics, such a query would simply not terminate, while in coinductive logic programming such a query terminates because it has a successful, finite co-*SLD* derivation.

Independently to the work introduced in this thesis, Jaffar et al. introduced a coinductive tabling proof method [45, 46] that uses coinduction as a means of proving infinitary properties in model checking. This is in contrast to using it in defining the semantics of a new declarative programming language, as is the case with coinductive logic programming and co-logic programming presented in this dissertation. Jaffar et al.'s coinductive tabling proof method itself is analogous to coinductive logic programming's co-*SLD* operational semantics described above in that both use the principle of coinduction to prove infinitary properties with some form of a finite derivation. However, Jaffar et al.'s coinductive tabling proof method is not assigned any declarative, model-theoretic semantics, as is the case with both coinductive logic programming and co-logic programming presented in this dissertation, which have a declarative semantics, operational semantics, and a correctness proof showing the correspondence between the two. Co-logic programming, when extended with constraints, can be used for the same applications as Jaffar et al.'s coinductive tabling proof method.

Like lazy functional programming [3], lazy functional *logic* programming (e.g., [47, 48, 49, 50, 51, 52]) also allows for infinite data structures, but it encodes predicates as Boolean functions, while in comparison, co-logic programming defines predicates via Horn clauses. The difference in semantics is even more pronounced. Predicates in lazy functional logic programming tend to have a mostly operational semantics in terms of lazy narrowing, which means that an instance of a predicate is true when the argument terms of the corresponding predicate can be instantiated in such a way that the function

evaluates to true. However, if the property is infinitary and has an infinite idealized proof, then the corresponding function will not evaluate to true because it will have an infinite evaluation. In co-logic programming, on the other hand, a coinductive predicate with an infinite idealized proof is defined as true, and the operational semantics allow for the finite derivation via the automated use of coinduction. Therefore, predicates in lazy functional logic programming are semantically different from those in coinductive logic programming. Of course, since functional logic programming languages typically allow for user programmable search strategies, it should be possible to implement coinductive logic programming in a functional logic programming language by effectively implementing a co-*SLD* search strategy. Similarly, a meta-interpreter for co-logic programming can be implemented on top of traditional logic programming.

The Horn μ-calculus of Charatonik et al. [53] and the alternation-free variant of Talbot [54] extend Horn logic with least and greatest fixed-points and provide declarative semantics in a manner similar to co-logic programming. The Horn μ-calculus does not have the stratification restriction found in co-logic programming, while the alternation-free restriction does. The Horn μ-calculus requires predicate symbols to be labeled with an integer priority, which is used in the definition of the language's semantics. Co-logic programming does not require predicates to have a specified priority, which is semantically unambiguous thanks to the stratification restriction. Aside from respectively defining the syntax and semantics for these variants of the Horn μ-calculus, both Charatonik et al. and Talbot only consider a non-Church-Turing-complete restriction to "uniform" programs. These uniform programs are then used to model reactive systems. Hence Charatonik et al. and Talbot don't provide a top-down, goal-direction operational semantics, let alone an efficient implementation for the full Horn μ-calculus or the full alternation-free Horn μ-calculus, as they only consider the restricted class of uniform

programs as a means of statically analyzing reactive systems, not as a general purpose programming language. Co-logic programming has a much more ambitious goal of unifying seemingly disparate logic programming concepts into a simple and efficient general purpose declarative programming language.

CHAPTER 3
COINDUCTIVE LOGIC PROGRAMMING

In chapters 1 and 2, we saw examples of logical relations that cannot be easily represented in traditional logic programming. These predicates deal with infinite objects and infinite properties. One of the contributions of this thesis is the creation of a new dual paradigm for logic programming, which allows for a natural approach to reasoning directly about infinite objects and infinite properties.

This development proceeds by redefining the declarative semantics of traditional logic programming. The declarative semantics for traditional logic programming has been given using the notions of Herbrand universe, Herbrand base, and minimal model [7]. Each is defined as a least fixed-point, and the set is manifested in traditional set theory. The declarative semantics of the language proposed here, called coinductive logic programming, takes the dual of each of these notions in hyperset theory with the *axiom of plenitude* [2].

This variation of the declarative semantics of a logic program has appeared before [43, 6, 7, 44] in order to describe rational trees and infinite, that is, non-terminating *SLD* derivations, while here it is used to describe finite derivations in a new operational semantics, which we call co-*SLD* in section 3.2.2. So the contribution of this thesis is the realization that the dual declarative semantics, under certain limitations discussed in section 3.2.4, describes a finitely computable behavior. That is, the contribution of this thesis is the development of an operational semantics for coinduction.

3.1 Syntax

A coinductive logic program P is syntactically identical to a traditional logic program. See section 2.2.1 for the formal definition of the syntax. The following defines a coinductive logic program as the combination of syntax and semantics, which assigns a definite program the semantics defined in the following section.

Definition 3.1 *A coinductive logic program is a definite program with the declarative semantics defined in section 3.2.1.*

3.2 Semantics

The semantics of coinductive logic programming follows that of traditional logic programming. However, both the declarative and operational semantics of traditional logic programming must be changed in order to capture the concepts of infinite objects and properties. Essentially, only the syntax remains the same. Hence at first glance, the two languages seem to be identical, though they are in fact quite different as the following sections demonstrate.

3.2.1 Declarative Semantics

The declarative semantics of coinductive logic programming is the "across the board" dual of the traditional minimal model Herbrand semantics [7, 36]. As demonstrated in [43, 6, 7, 44], this allows the universe of terms to contain infinite terms, in addition to the traditional finite terms; and, it also allows for the model to contain ground goals that have either finite or infinite idealized proofs. The difference here is that we define such goals as true, and in the next section we provide a new operational semantics that yields *finite* derivations for goals with an (rational) *infinite* idealized proof.

Terms in traditional logic programming are always finite, and meaning is assigned to them via definition by least fixed-point. Here terms can be finite as well as infinite, and therefore it is necessary to define the meaning of terms in coinductive logic programming as a greatest fixed-point.

Definition 3.2 *Let P be a coinductive logic program. Let $A(P)$ be the set of constants in P, and let $F_n(P)$ denote the set of function symbols of arity n in P. The co-Herbrand universe of P, denoted $U^{co}(P) = \nu\Phi_P$, where*

$$\Phi_P(S) = A(P) \cup \{f(t_1, \ldots, t_n) \mid f \in F_n(P) \wedge t_1, \ldots, t_n \in S\}$$

As was the case with $U(P)$ (defined in section 2.2.2.1), for technical reasons it is assumed that $U^{co}(P)$ is always not empty. Intuitively, $U^{co}(P)$ is the set of terms both finite and infinite that can be constructed from the constants and functions in the program. Also, note that the co-Herbrand universe of program P is the dual of the Herbrand universe of program P.

This seemingly simple variation on the Herbrand universe has important practical ramifications. Logic programming's operational semantics implicitly makes use of unification, and while unification can be implemented efficiently, in general, it can be even more efficiently implemented when occurs check is not necessary. The occurs check in a unification procedure involves checking for implied cyclic equality constraints. For example, the occurs check forbids the unification of X and $f(X)$, as no finite term can satisfy the equation $X = f(X)$, for function symbol f. This occurs check is computationally expensive, but necessary for the soundness of traditional logic programming's operational semantics. Coinductive logic programming, on the other hand, allows for solutions to such equations, that is, unification *without* occurs check has a greatest fixed-point interpretation, as infinite trees are included in the co-Herbrand universe. Hence the

unification of X and $f(X)$ has the solution for X being the infinite term $f(f(f(\ldots)))$, that is, the term consisting of an infinite application of the function f.

Definition 3.3 *Let P be a coinductive logic program. The co-Herbrand base (also known as the infinitary Herbrand base [6]), written $B^{co}(P)$, is the set of all ground atoms that can be formed from the predicate symbols in P and the elements of $U^{co}(P)$. Also, let $G^{co}(P)$ be the set of ground clauses $C \leftarrow D_1, \ldots, D_n$ that are a ground instance of some clause of P such that $C, D_1, \ldots, D_n \in B^{co}(P)$.*

As stated, the formal definition of coinductive logic programming's declarative semantics is the mirror image of traditional logic programming's minimal Herbrand model.

Definition 3.4 *A co-Herbrand model of a program P is a fixed-point of*

$$T_P(S) = \{C \mid C \leftarrow D_1, \ldots, D_n \in G^{co}(P) \wedge D_1, \ldots, D_n \in S\}$$

The maximal co-Herbrand model of a program P, denoted $M^{co}(P)$, is the greatest fixed-point of T_P, which exists and is unique according to theorem 2.1. $M^{co}(P)$ is taken to be the declarative semantics of a coinductive logic program P.

The traditional minimal model, on the other hand, is the restriction of the least fixed-point of T_P to the Herbrand base. Hence coinductive logic programming's declarative semantics is dual to traditional logic programming's semantics, "across the board": Herbrand universe, Herbrand base, and the minimal Herbrand model are all taken in "co-" form. Truth of a coinductive logic statement is defined in terms of inclusion in the co-Herbrand model.

Definition 3.5 *An atom A is true in a coinductive logic program P if and only if the set of all groundings of A, with substitutions ranging over the $U^{co}(P)$, is a subset of $M^{co}(P)$.*

Now we can review examples from 2.2.2.1 in order to contrast the stark difference between the syntactically identical inductive logic programming and coinductive logic programming.

Example 3.1 *Let P_1 be the following program.*

```
from(N, [N|T]) :- from(s(N), T).

| ?- from(0, _).
```

The coinductive semantics are derived as follows. The co-Herbrand universe is $U^{co}(P_1) = N \cup \Omega \cup L$ where $N = \{0, s(0), s(s(0)), \ldots\}$, $\Omega = \{s(s(s(\ldots)))\}$, and L is the set of all finite and infinite lists of elements in N, Ω, L. Therefore the maximal co-Herbrand model $M^{co}(P_1) = \{from(t, [t, s(t), s(s(t)), \ldots]) \mid t \in U^{co}(P_1)\}$, which is the meaning of the program and obviously not null, as was the case with traditional logic programming. Furthermore $from(0, [0, s(0), s(s(0)), \ldots]) \in M^{co}(P_1)$ implies that the query returns "yes".

The next example has a simpler semantics.

Example 3.2 *Let P_2 be the following program.*

```
p :- p.
```

The maximal co-Herbrand model $P_2 = \{\mathtt{p}\}$, which is not empty. Hence, the query `| ?- p.` *would terminate with a "yes".*

While this may seem counter to the intuition of Horn logic in terms of an encoding of clauses in terms of classical logic, a better intuition of the semantics is that clauses are merely inference rules for a proof system that allows finite and infinite proofs. Therefore, p is true because it has an infinite proof. Of course, it isn't always desirable to allow

infinite proofs for every predicate, which is addressed by the combination of inductive and coinductive logic programming in the next chapter. This combination results in the core contribution of this dissertation: co-logic programming. In co-logic programming the programmer annotates predicates as either "inductive" or "coinductive" [10, 5], effectively allowing the programmer to invoke least and greatest fixed-points at their discretion.

Example 3.3 *Recall the bit stream example from the introduction.*

```
bit(0).
bit(1).
bitstream([H | T]) :- bit(H), bitstream(T).
```

The co-Herbrand universe contains 0, 1, *finite* and infinite *lists of these two digits, and finite* as well as infinite *lists of elements from the universe itself. Because the* `bitstream/1` *predicate requires its second argument to be an infinite list of binary digits, the minimal Herbrand model of this program only contains* $bit(0)$ *and* $bit(1)$*. The co-Herbrand model, however, also contains* $bitstream(t)$*, where* t *is an infinite binary stream. Hence coinductive logic programming is required for this kind of a program, as the* `bitstream/1` *predicate has no meaning in traditional logic programming, while it effectively means "this is a binary stream" in coinductive logic programming.*

The model characterizes semantics in terms of truth, that is, the set of ground atoms that are true. This set is defined via a generator, and in section 3.2.4, we discuss the way in which the generator is applied in order to include an atom in the model. For example, the generator is only allowed to be applied a finite number of times for any given atom in the minimal model, while it can be applied an infinite number of times for an atom in the maximal co-Herbrand model. We characterize this by recording the application of

the generator in the elements of the fixed-point itself. We call these elements "idealized proofs".

Definition 3.6 *Let $node(A, L)$ be a constructor of a tree with root A and subtrees L, where A is an atom and L is a list of trees. The set of idealized proofs for program P is $\nu\Sigma_P$, where*

$$\begin{aligned}\Sigma_P(S) = \quad & \{node(C, [T_1, \ldots, T_n]) \mid \\ & C \leftarrow D_1, \ldots, D_n \in G^{co}(P) \ \wedge \ \textit{the root of } T_i \in S \textit{ is } D_i\}\end{aligned}$$

Again, this is nothing more than a reformulation of the maximal co-Herbrand model, which records the applications of the generator in the elements of the fixed-point itself, as the following theorem demonstrates.

Theorem 3.1 *Let $S = \{A \mid \exists T \in \nu\Sigma_P.A \textit{ is the root of } T\}$, then $S = M^{co}(P)$.*

Every element in the maximal co-Herbrand model has an idealized proof and anything that has an idealized proof is in the model. A similar theorem exists, equating the minimal model with the least fixed-point of Σ_P restricted to finite terms, i.e., the minimal model consists of all ground atoms that have a finite idealized proof. This recasting of the declarative semantics in terms of idealized proofs will be used in the completeness proof in section 3.2.4.

3.2.2 Operational Semantics

This section presents one of the main contributions of this dissertation: an operational semantics for coinduction. This is accomplished by noting that a tabling, that is, memoization mechanism can be used to dynamically synthesize a coinductive hypothesis while executing the program in a manner similar to traditional *SLD*. This coinductive hypothesis is then invoked, at will, by the language's virtual machine.

The operational semantics given for coinductive logic programming is defined in a manner similar to *SLD*, and is therefore called co-*SLD*. Where *SLD* uses sets of syntactic atoms and syntactic term substitutions for states, co-*SLD* uses finite trees of syntactic atoms along with systems of equations. Of course, the traditional goals of *SLD* can be extracted from these trees, as the goal of a tree is simply the set of leaves of the tree. Furthermore, where *SLD* only allows program clauses as state transition rules, co-*SLD* also allows an implicit coinductive hypothesis rule for providing atoms that have an infinite proof, with a finite derivation. As is the case with *SLD*, it is up to the underlying search strategy to find a sequence of transition rules that prove the original query.

The operational semantics for coinduction is defined in this section as a part of the operational semantics for coinductive logic programming. These definitions require some infinite tree theory. However, this section only states a few well known definitions and theorems without proof. Details can be found in classic work of Courcell [55].

Definition 3.7 *A tree is rational if the cardinality of the set of all its subtrees is finite. An object such as a term, atom, or idealized proof is said to be rational if it is modeled as a rational tree.*

Definition 3.8 *A substitution is a finite mapping of variables to terms. A substitution is syntactic if it only substitutes syntactic terms for variables. A substitution is said to be rational if it only substitutes rational terms for variables.*

Definition 3.9 *A term unification problem is a finite set of equations between terms. A unifier for a term unification problem is a substitution that satisfies every equation in the problem. σ is a most general unifier for a term unification problem, if any other solution σ' can be defined as the composition $\sigma'' \circ \sigma$.*

Note that terms are possibly infinite. So it is possible for a unification problem to lack a syntactic unifier, while at the same time the problem has a solution: a rational unifier. However, objects of an operational semantics should be finite. Hence we define a standard finite representation of rational substitutions called a system of equations.

Definition 3.10 *A system of equations E is a term unification problem where each equation is of the form $X = t$, s.t. X is a variable and t a syntactic term.*

Theorem 3.2 *(Courcelle) Every system of equations has a most general unifier that is rational.*

Theorem 3.3 *(Courcelle) For every rational substitution σ with domain V, there is a system of equations E, such that the most general unifier σ' of E is equal to σ when restricted to the domain V.*

Without loss of generality, the previous two theorems allow for a solution to a term unification problem to be simultaneously a substitution as well as a system of equations. Note that given a substitution specified as a system of equations E, and a term A, the term $E(A)$ denotes the result of applying the substitution E to A.

Now the operational semantics can be defined. The semantics implicitly defines a state transition system. Systems of equations are used to model part of the state of coinductive logic programming's semantics. They effectively denote the current state of unification of terms. The current state of the pending goals is modeled using a finite tree of atoms, as it is necessary to recognize cycles in the sequence of pending goals, that is, the ancestors of a goal are memo-ed in order to recognize a cycle in the proof.

Definition 3.11 *A state S is a pair (T, E), where T is a finite tree with nodes labeled with syntactic atoms, and E is a system of equations.*

Definition 3.12 *A transition rule R of a coinductive logic program P is an instance of a clause in P, with variables standardized apart, i.e., consistently renamed for freshness, or R is a coinductive hypothesis rule of the form $\nu(n)$, where n is a natural number.*

Obviously the state transition system may be nondeterministic, depending on the program. In other words, it is possible for states to have more than one outgoing transition as the following definition shows.

Definition 3.13 *A state (T, E) transitions to another state (T', E') by transition rule R of program P whenever:*

1. R is a definite clause of the form $p(t'_1, \ldots, t'_n) \leftarrow B_1, \ldots, B_m$ and E' is the most general unifier for $\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E$, and T' is obtained from T according to the following case analysis of m:

   (a) $m = 0$ implies T' is obtained from T by removing a leaf labeled $p(t_1, \ldots, t_n)$ and the maximum number of its ancestors, such that the result is still a tree.

   (b) $m > 0$ implies T' is obtained from T by adding children $B_1, \ldots, B_m$ to a leaf labeled with $p(t_1, \ldots, t_n)$.

2. R is of the form $\nu(m)$, a leaf N in T is labeled with $p(t_1, \ldots, t_n)$, the proper ancestor of N at depth m is labeled with $p(t'_1, \ldots, t'_n)$, E' is the most general unifier for $\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E$, then T' is obtained from T by removing N and the maximum number of its ancestors, such that the result is still a tree.

The part of the previous definition that removes a leaf and a maximum number of its ancestors can be thought of as a successful call returning and therefore deallocating memo-ed calls on the call stack. This involves successively removing ancestor nodes of the leaf until an ancestor is reached, which still has other children, and so removing any more ancestors would cause the result to no longer be a tree, as children would be orphaned. Hence the depth of the tree is bounded by the depth of the call stack.

Definition 3.14 *A transition sequence in program P consists of a sequence of states $S_1, S_2, \ldots$ and a sequence of transition rules $R_1, R_2 \ldots$, such that S_i transitions to S_{i+1} by rule R_i of program P.*

A transition sequence denotes the trace of an execution. Execution halts when it reaches a terminal state: either all goals have been proved or the execution path has reached a dead-end.

Definition 3.15 *The following are two distinguished terminal states:*

1. An accepting state is of the form $(\emptyset, E)$, where $\emptyset$ denotes an empty tree.
2. A failure state is a non-accepting state lacking any outgoing edges.

Without loss of generality, we restrict queries to be single syntactic atoms. A query containing multiple atoms can be modeled by adding a new predicate with one clause to the program. Finally we can define the execution of a query as a transition sequence through the state transition system induced by the input program, with the start state consisting of the initial query.

Definition 3.16 *A co-SLD derivation of a state (T, E) in program P is a state transition sequence with the first state equal to (T, E). A derivation is successful if it ends in an accepting state, and a derivation has failed if it reaches a failure state. We say that a syntactic atom A has a successful derivation in program P, if $(A, \emptyset)$ has a successful derivation in P.*

3.2.3 Examples

In addition to allowing infinite terms, the operational semantics allows for an execution to succeed when it encounters a goal that unifies with an ancestor goal. While this is somewhat similar to tabled logic programming [56] in that called atoms are recorded so as to avoid unnecessary redundant computation, the difference is that coinductive logic programming's memo-ed atoms represent a coinductive hypothesis, while tabled logic programming's table represents a list of results for each called goal in the traditional inductive semantics.

Hence we call the memo-ed atoms in a coinductive logic program the dynamic coinductive hypothesis. An example that demonstrates the distinction is the following program.

```
p :- p.

| ?- p.
```

Execution starts by checking the dynamic coinductive hypothesis for an atom that unifies with `p`, which does not exist, so `p` is added to the hypothesis. Next, the body of the goal is executed. Again, the hypothesis is checked for an atom that unifies with `p`, which is now already included in the hypothesis, so the most recent call succeeds and then since no remaining goals exist, the original query succeeds. Hence, according to the operational semantics of coinductive logic programming, the query has a successful

derivation, and hence returns "yes", while traditional (tabled) logic program returns "no".

Now for a more complicated example involving function symbols. Consider the execution of the following program, which defines a predicate that recognizes infinite streams of natural numbers and ω, that is, infinity.

```
stream([H | T]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).
| ?- stream([0, s(0), s(s(0)) | T ]).
```

The following is an execution trace, for the above query, of the memoization and unmemoization of calls by the operational semantics:

1. `MEMO: stream( [ 0, s(0), s(s(0)) | T ] )`
2. `MEMO: number( 0 )`
3. `UNMEMO: number( 0 )`
4. `MEMO: stream( [ s(0), s(s(0)) | T ] )`
5. `MEMO: number( s(0) )`
6. `MEMO: number( 0 )`
7. `UNMEMO: number( 0 )`
8. `UNMEMO: number( s(0) )`
9. `MEMO: stream( [ s(s(0)) | T ] )`

10. `MEMO: number( s(s(0)) )`
11. `MEMO: number( s(0) )`
12. `MEMO: number( 0 )`
13. `UNMEMO: number( 0 )`
14. `UNMEMO: number( s(0) )`
15. `UNMEMO: number( s(s(0)) )`

The next goal call is `stream( T )`, which unifies with memo-ed ancestor (1), and therefore immediately succeeds. Hence the original query succeeds with

```
T = [ 0, s(0), s(s(0)) | T ]
```

The user could force a failure here, which would cause the goal to be unified with the next matching memo-ed ancestor, if such an element exists, otherwise the goal is memo-ed and the process repeats—generating additional results (`T = [0, s(0), s(s(0)) | R]`, `R = [0 | R]`, etc.). Note that excluding the occurs check is necessary as such structures have a greatest fixed-point interpretation and are in the co-Herbrand Universe. Again, this is in fact one of the benefits of coinductive logic programming. Traditional logic programming's least Herbrand model semantics requires *SLD* resolution to unify with occurs check (or otherwise lack soundness), which adversely affects performance in the common case. Coinductive logic programming, on the other hand, has a declarative semantics that allows unification without doing occurs check in an efficient manner as seen in rational tree unification, and in addition, coinductive logic programming allows for programs to reason about rational terms generated by rational tree unification in a manner that is impossible in traditional logic programming, as traditional logic programming

would diverge into an infinite derivation, where coinductive logic programming would yield a finite derivation thanks to the dynamic synthesis of a coinductive hypothesis via memoization.

3.2.4 Correctness

We next prove the correctness of the operational semantics by demonstrating its correspondence with the declarative semantics via soundness and completeness theorems. Completeness, however, must be restricted to atoms that have a rational proof.

Lemma 3.1 *If (A, E_1) has a successful co-SLD derivation in program P, with final state $(\emptyset, E_2)$, then (A, E_3) has a successful co-SLD derivation in program P, where $E_2 \subseteq E_3$, with each state of the derivation of the form (T, E_3) for some tree of atoms T.*

Proof 3.1 *Let (A, E_1) have a successful co-*SLD *derivation in program P ending with state $(\emptyset, E_2)$. In the sequence of states, the system of equations monotonically increases, and so the monotonicity of unification with infinite terms implies (A, E_3) has a successful co-*SLD *derivation in program P, where $E_2 \subseteq E_3$, with each state of the derivation of the form (T, E_3) for some tree of atoms T.*

Lemma 3.2 *If A has a successful co-SLD derivation in program P, which first transitions to a state $(node(A, [B_1, \ldots, B_n]), E)$ by applying clause*
$A' \leftarrow B_1, \ldots, B_n$, such that $E(A) = E(A')$, then each (B_i, E) also has a successful derivation in program P.

Proof 3.2 *Let $(node(A, [B_1, \ldots, B_n]), E)$ have a successful co-*SLD *derivation in program P, which first transitions to a state $(node(A, [B_1, \ldots, B_n]), E)$ by applying clause $A' \leftarrow B_1, \ldots, B_n$, such that $E(A) = E(A')$. A derivation for (B_i, E) can be created by*

mimicking each transition that modifies the subtree rooted at B_i in the original derivation, except for the transitions which are of the form $\nu(n)$, which no longer are correct derivations because the parent A of B_i no longer exists. In the case that $n > 0$, instead apply the transition rule $\nu(n-1)$ to the corresponding leaf to which the original derivation would have applied $\nu(n)$. Otherwise, when $n = 0$, a coinductive transition rule cannot be applied to the corresponding leaf. Instead, mimic the transitions of the entire original derivation of A.

Lemma 3.3 *If (A, E) has a successful co-SLD derivation in programP, then $E'(E(A))$ is true in program P, where $(\emptyset, E')$ is the final state of the derivation.*

Proof 3.3 *Let Q be the set of all groundings ranging over the $U^{co}(P)$ of all such $E'(E(A))$. It is sufficient to prove that $Q \subseteq M^{co}(P)$. The proof proceeds by coinduction.*

Let $A' \in S$, then $A' = E_1(E_2(E_3(A)))$, where E_1 is a grounding substitution for $E_2(E_3(A))$ and (A, E_3) has a successful derivation ending in $(\emptyset, E_2)$. By lemma 4.1, (A, E) has a successful derivation, where $E = E_1 \cup E_2 \cup E_3$. This derivation must begin with an application of a program clause $A' \leftarrow B_1, \ldots, B_n$, resulting in the state $(node(A, [B_1, \ldots, B_n]), E)$, where $E(A) = E(A')$. By lemma 4.2 each state (B_i, E) has a successful derivation. Let E' be a grounding substitution for the clause $E(A' \leftarrow B_1, \ldots, B_n)$, such that $C = E'(E(A' \leftarrow B_1, \ldots, B_n)) \in G^{co}(P)$, then
$C = A \leftarrow E''(B_1), \ldots, E''(B_n)$, where $E'' = E' \cup E$. By lemma 4.1, each (B_i, E'') has a successful derivation, and hence $E''(B_i) \in Q$. Therefore, by the principle of coinduction, $Q \subseteq M^{co}(P)$.

Theorem 3.4 *(soundness) If atom A has a successful co-SLD derivation in program P, then $E(A)$ is true in program P, where E is the resulting variable bindings for the derivation.*

Proof 3.4 *This follows by a straightforward instantiation of lemma 4.3.*

As this theorem demonstrates, the soundness of co-*SLD* does not require the execution of an occurs check during unification, as the declarative semantics naturally allows for infinite terms. The soundness of *SLD*, with regards to the minimal Herbrand model, however, requires the occurs check, which has an adverse affect on performance.

Theorem 3.5 *(completeness) If $A \in M^{co}(P)$ has a rational idealized proof, then A has a successful co-SLD derivation in program P.*

Proof 3.5 *Let $A \in M^{co}(P)$ have a rational idealized proof T. The derivation is constructed by recursively applying the clause corresponding to each node encountered along a depth-first traversal of the idealized proof tree to the corresponding leaf in the current state. In order to ensure that the derivation is finite, the traversal stops at the root R of a subtree that is identical to a subtree rooted at a proper ancestor at depth n. Then the derivation applies a transition rule of the form $\nu(n)$ to the leaf corresponding to R in the current state, and finally the depth-first traversal continues traversing starting at a node in the idealized proof tree corresponding to some leaf in the current state of the derivation.*

The fact that the set of all subtrees of T is finite in cardinality implies that the maximum depth of the traversal in the idealized proof tree is finite, and the fact that all idealized proofs are finitely branching implies that the traversal always terminates. So the constructed derivation is finite.

It is still necessary prove that the final state of the constructed derivation is a success state. The traversal stops going deeper in the idealized proof tree due to two cases: the traversal reaches a leaf in the idealized proof tree or the traversal encountered a subtree

*identical to an ancestor subtree. In either case, the derivation removes the corresponding leaf in the current state, as well as the maximal number of ancestors of the corresponding leaf such that the result is still a tree. So a leaf only remains in the state when its corresponding node in the proof tree has yet to be traversed. Since every node in the idealized proof tree corresponding to a leaf in the state is traversed at some point, the final state's tree contains no leaves, and hence the final state has an empty tree, which is the definition of an accept state. Therefore, A has a successful co-*SLD *derivation in program P.*

3.3 Implementation

Just as it is possible to implement a compiler or interpreter for C++, in C++ itself, it is possible to implement an interpreter for logic programming in logic programming itself. Figure 3.1 is a source listing of "`sld.pro`", which is an interpreter for traditional logic programming, written in itself. The interpreter is used by including this file in your logic program source file. This allows for the rapid prototyping of new forms of logic programming.

This interpreter, sometimes called a meta-interpreter, works by directly encoding the formal operational semantics of logic programming. A query is passed to a predicate called `solve`. The first clause of this predicate tries to solve each outstanding atom in the current *SLD* state, while the third clause of `solve` actually solves an atomic logic statement by applying one of the clauses of the interpreted program. This is accomplished by invoking a built-in predicate `clause` that makes it possible to retrieve the body of any clause that has a head that unifies with the first argument of the predicate. The second `solve` clause is a special case for solving atoms that are built-in to the language. This interpreter must choose an order for applying the clauses from the interpreted program

```
query( Goal ) :- solve( Goal ).

solve( ( Goal1, Goal2 ) ) :-
  solve( Goal1 ),
  solve( Goal2 ).

solve( Atom ) :-
  builtin( Atom ),
  Atom.

solve( Atom ) :-
  Atom \= ( _, _ ),
  \+ builtin( Atom ),
  clause( Atom, Atoms ),
  solve( Atoms ).

builtin( _ = _  ).
builtin( true   ).
builtin( _ is _ ).
builtin( _ > _  ).
builtin( _ < _  ).
```

Figure 3.1. Logic Programming Interpreter "`sld.pro`"

to atoms in the current state, and so it uses the traditional left-most, depth-first search strategy that is implicit in the ordering of the clauses and goals in the "`sld.pro`" source file.

A traditional logic programming query is executed by loading the logic program source file as if it were a normal SICStus program. Then queries must begin and end, with "`query((`" and "`)).`" respectively in order to be executed.

Following the example of traditional logic programming, the interpreter for coinductive logic programming uses a left-most, depth-first search strategy for searching for a successful co-*SLD* derivation of a given query, within a given coinductive logic program. However, we must also decide when to apply the coinductive hypothesis rule. The main

intent of this rule is to yield a finite derivation of a query that would otherwise have an infinite derivation, and for this purpose, we use a "hypothesis-first" strategy, yielding the implementation's overall hypothesis-first, left-most, depth-first search strategy. The other way in which the coinductive logic programming interpreter differs from the interpreter for traditional logic programming is that the `solve` predicate carries a second argument called `Hypothesis` that represents the dynamically synthesized coinductive hypothesis. The `solve` predicate uses this argument in its third clause, which represents the coinductive hypothesis rule from the co-*SLD* semantics described in section 3.2.2.

Figure 3.2 shows the source code listing of "`cosld.pro`", which is an interpreter for coinductive logic programming that runs on SICStus Prolog. The interpreter is used by including this file in every coinductive logic program source file. Furthermore, all user defined predicates must be declared dynamic, as demonstrated in the example coinductive logic program "`example.cosld`" depicted in figure 4.2. It is also forbidden to redefine a built in predicate.

A query is executed by loading the coinductive logic program source file as if it were a normal SICStus program. Then queries must begin and end, with "`query((`" and "`)).`" respectively in order to be executed with co-*SLD* semantics using a coinductive hypothesis first, left-most, depth-first search strategy.

```
query( Goal ) :- solve( [], Goal    ).

solve( Hypothesis, ( Goal1, Goal2 ) ) :-
  solve( Hypothesis, Goal1 ),
  solve( Hypothesis, Goal2 ).

solve( _          , Atom            ) :-
  builtin( Atom ),
  Atom.

solve( Hypothesis, Atom             ) :-
  member( Atom, Hypothesis ).

solve( Hypothesis, Atom             ) :-
  Atom \= ( _, _ ),
  \+ builtin( Atom ),
  clause( Atom, Atoms ),
  solve( [ Atom | Hypothesis ], Atoms ).

builtin( _ = _  ).
builtin( true   ).
builtin( _ is _ ).
builtin( _ > _  ).
builtin( _ < _  ).

member( X, [ X | _ ] ).
member( X, [ _ | T ] ) :- member( X, T ).
```

Figure 3.2. Coinductive Logic Programming Interpreter "`cosld.pro`"

```
:- include( 'cosld.pro' ).


:- dynamic p/0.
:- dynamic bit/1.
:- dynamic bitstream/1.
:- dynamic equal/2.
:- dynamic num/1.
:- dynamic stream/1.
:- dynamic append1/3.

p :- p.

bit( 0 ).
bit( 1 ).

bitstream( [ H | T ] ) :- bit( H ), bitstream( T ).

equal( X, X ).

num( 0 ).
num( s( N ) ) :- num( N ).

stream( [ H | T ] ) :- num( H ), stream( T ).

append1( [], X, X ).
append1( [ H | T ], Y, [ H | Z ] ) :- append1( T, Y, Z ).
```

Figure 3.3. Coinductive Logic Programming Examples "`example.cosld`"

CHAPTER 4
CO-LOGIC PROGRAMMING

In the previous chapter, we noted that there are predicates that can be naturally described in traditional (inductive) logic programming, yet not in coinductive logic programming, and as demonstrated in chapters 1, 2, and 3, there are predicates such as `bitstream` that are easily coded in coinductive logic programming, yet they cannot be easily coded in traditional logic programming. The obvious conclusion is that it would be beneficial to have a logic programming language that combined the benefits of both paradigms. This combination gives rise to co-logic programming, which is a new logic programming paradigm and the main contribution of this dissertation.

There are many approaches to combining these two paradigms. One approach would be to have the compiler automatically decide when it is best to use the inductive semantics and when it is best to use the coinductive semantics. The problem with this approach is that deciding which semantics to use involves somehow deciding the intent of the programmer. Hence it would be best to not try to guess the programmer's intent. Assuming the programmer annotates predicates as "inductive" or "coinductive", the problem then posed is how to assign meaning to the program in a way that simultaneously reflects the semantics of both inductive and coinductive logic programming. We have already seen how to give a mathematical meaning to each kind of predicate in isolation, but when combining both kinds of predicates, both inductive and coinductive, in the same program, it must be determined what it means for an inductive predicate to call a coinductive predicate and visa versa. Even more problematic is the possibility of mutual recursion between inductive and coinductive predicates.

In the next section, the abstract syntax of co-logic programming is defined, and unlike coinductive logic programming, co-logic programming's syntax is not identical to traditional logic programming's syntax. Though it is very similar. After that, in section 4.2.1, the canonical mathematical semantics or declarative semantics of co-logic programming is defined. This semantics elegantly subsumes the declarative semantics of inductive and coinductive logic programming, though the formal definitions required are tedious. As usual with declarative programming languages, knowing the syntax and meaning of a program is not useful unless there is a means of efficiently executing said programs. So in section 4.2.2 the operational semantics for co-logic programming is defined, and it also turns out to be a conservative generalization of the operational semantics of inductive and coinductive logic programming. In fact, there is no additional overhead incurred compared to the operational semantics for coinductive logic programming. The standard correctness proofs are given in section 4.2.3, and the final section in this chapter discusses a concrete implementation of a programming language based on the co-logic programming paradigm.

4.1 Syntax

As discussed, co-logic programming extends the syntax of coinductive logic programming by allowing predicate symbols to be annotated as being either inductive or coinductive. However, there is one restriction, referred to as the "stratification restriction". Inductive and coinductive predicates are not allowed to be mutually recursive, that is, execution cannot loop between inductive and coinductive calls. The reason for this limitation in syntax is that the program would become semantically ambiguous if an inductive predicate was allowed to be mutually recursive with a coinductive predicate. While it would be possible to overcome this ambiguity with further programmer-supplied annotations

on predicates as seen in the Horn μ-calculus [53], the resulting syntax would no longer be intuitive and it would no longer resemble a straightforward and conservative extension of inductive and coinductive logic programming.

To this end, it is easier to first define an annotated logic program that does not have the stratification restriction, and then later a co-logic program is defined as such a program with the restriction.

Definition 4.1 *A pre-program is a definite program paired with a mapping of predicate symbols to the token* `coinductive` *or the token* `inductive`*. A predicate is said to be coinductive (resp. inductive) if the partial mapping maps the predicate to* `coinductive` *(resp.* `inductive`*). Similarly, an atom is said to be coinductive (resp. inductive) if the underlying predicate is coinductive (resp. inductive).*

Not every pre-program is a co-logic program. Co-logic programs do not allow for any pair of inductive and coinductive predicates to be mutually recursive, that is, programs must be stratified with regards to alternating induction and coinduction. An inductive predicate can call, directly or indirectly, a coinductive predicate and visa versa, but both such predicates cannot be mutually recursive. The following definitions formally define this restriction.

Definition 4.2 *In some pre-program P, we say that a predicate p depends on a predicate q if and only if $p = q$ or P contains a clause $C \leftarrow D_1, \ldots, D_n$ such that C contains p and some D_i contains q. The dependency graph of program P has the set of its predicates as vertices, and the graph has an edge from p to q if and only if p depends on q.*

Co-logic programs are simply pre-programs that obey the following restriction, which is called the stratification restriction.

Definition 4.3 *A co-logic program is a pre-program such that for any strongly connected component G in the dependency graph of the program, every predicate in G is either mapped to* `coinductive` *or every predicate in G is mapped to* `inductive`*.*

4.2 Semantics

As was the case with coinductive logic programming and traditional logic programming before it, the "canonical" semantics for co-logic programming is given in terms of its declarative semantics, and the practical semantics for co-logic programming is given in terms of its operational semantics. The former tells us what an expression in the language means, while the latter tells us how to execute an expression in the language.

4.2.1 Declarative Semantics

The declarative semantics of a co-logic program is a stratified interleaving of the minimal co-Herbrand model [36, 7] and the maximal co-Herbrand model semantics [5]. Hence co-logic program strictly contains logic programming with rational trees [6] as well as coinductive logic programming [8, 9, 4]. This allows the universe of terms to contain infinite terms, in addition to the traditional finite terms. Finally, co-logic programming also allows for the model to contain ground goals that have either finite or infinite idealized proofs.

The following definition is necessary for defining the model of a co-logic program. Intuitively, a reduced graph is derived from a dependency graph by collapsing the strongly connected components of the dependency graph into single nodes. The graph resulting from this process is acyclic.

Definition 4.4 *The reduced graph for a co-logic program has vertices consisting of the strongly connected components of the dependency graph of P. There is an edge from v_1*

to v_2 in the reduced graph if and only if some predicate in v_1 depends on some predicate in v_2. A vertex in a reduced graph is said to be coinductive (resp. inductive) if it contains only coinductive (resp. inductive) predicates.

A vertex in a reduced graph of a program P is called a stratum, as the set of predicates in P is stratified into a collection of mutually disjoint strata of predicates. The stratification restriction states that all vertices in the same stratum are of the same kind, i.e., every stratum is either inductive or coinductive. A stratum v depends on a stratum v', when there is an edge from v to v' in the reduced graph. When there is a path in the reduced graph from v to v', v is said to be higher than v' and v' is said to be lower than v, and the case when $v \neq v'$ is delineated by the modifier "strictly", as in "strictly higher" and "strictly lower". This restriction allows for the model of a stratum v to be defined in terms of the models of the strictly lower strata, upon which v depends.

The model of a stratum, i.e., the model of a vertex in the reduced graph of a co-logic program. The model of each stratum is defined using what is effectively the same T_P monotonic operator used in defining the minimal Herbrand model [36, 7], except that it is extended so that it can treat the atoms defined as true in lower strata as facts when proving atoms containing predicates in the current stratum. This is possible because co-logic programs are stratified such that the reduced graph of a program is always a DAG and every predicate in the same stratum is the same kind: inductive or coinductive.

Definition 4.5 *The model of a stratum v of P equals μT_P^v if v is inductive and νT_P^v if v is coinductive, such that R is the union of the models of the strata that are strictly lower than v and*

$$T_P^v(S) = R \cup \left\{ q\left(\widehat{t}\right) \mid q \in v \wedge [q\left(\widehat{t}\right) \leftarrow \widehat{D}] \in G^{co}(P) \wedge \widehat{D} \in S \right\}$$

Since any predicate resides in exactly one stratum, the definition of the model of a co-logic program is straightforward.

Definition 4.6 *The model of a co-logic program P, written $M(P)$, is the union of the model of every stratum of P.*

Obviously co-logic programming's semantics subsumes the minimal co-Herbrand model used as the semantics for logic programming with rational trees, as well as the maximal co-Herbrand model used as the semantics for coinductive logic programming. The alternating fixed-point model of a co-logic program is built by creating the inductive model for an inductive strata, using the lower strata as facts, and similarly for the coinductive model is made for a coinductive strata. This process is repeated from the lowest strata up to the topmost strata. Lowest and topmost strata exist because the program is stratified.

Definition 4.7 *An atom A is true in co-logic program P if and only if the set of all groundings of A with substitutions ranging over the $U^{co}(P)$, is a subset of $M(P)$.*

Example 4.1 *Let P_1 be the following program.*

```
coinductive(from, 2).
from(N, [N|T]) :- from(s(N), T).

| ?- from(0, _).
```

The model of the program, which is defined in terms of an alternating fixed-point is as follows. The co-Herbrand Universe is $U^{co}(P_1) = N \cup \Omega \cup L$ where $N = \{0, s(0), s(s(0)), \ldots\}$,

$\Omega = \{s(s(s(\ldots)))\}$, *and* L *is the set of all finite and infinite lists of elements in* N, Ω, *and even* L. *Therefore the model*

$$M(P_1) = \{from(t, [t, s(t), s(s(t)), \ldots]) \mid t \in U^{co}(P_1)\}$$

is the meaning of the program and is obviously not null, as was the case with traditional logic programming. Furthermore $from(0, [0, s(0), s(s(0)), \ldots]) \in M(P_1)$ *implies that the query returns "yes". On the other hand, if the directive on the first line of the program was removed, call the resulting program* P_1', *then the program's only predicate would by default be inductive, and* $M(P_1') = \emptyset$. *This corresponds to the traditional semantics of logic programming with infinite trees. Examples involving multiple strata of different kinds, i.e., mixing inductive and coinductive predicates, are given in section 4.2.2, chapter 5, and chapter 7.*

As was the case with the previous two chapters, the model of a co-logic program characterizes semantics in terms of truth, that is, the set of ground atoms that are true. This set is defined via a generator, and in section 4.2.3, we will need to talk about the manner in which the generator is applied in order to include an atom in the model. For example, the generator is only allowed to be applied a finite number of times for any given atom in a least fixed-point, while it can be applied an infinite number of times in the greatest fixed-point. For programs involving both induction and coinduction, this process alternates. As was done in the previous chapter, this process can be captured in the form of an idealized proof by recording the application of the generator in the elements of the fixed-point itself. In order to define the idealized proofs used in co-logic programming, it is first necessary to define some formalisms for trees.

Definition 4.8 *A path* π *is a finite sequence of positive integers* i. *The empty path is written* ϵ, *the singleton path is written* i *for some positive integer* i, *and the concatenation*

of two paths is written $\pi \cdot \pi'$. *A tree of* S, *also called an* S*-tree, is formally defined as a partial function from paths to elements of* S, *such that the domain is non-empty and prefix-closed. A node in a tree is unambiguously denoted by a path. So a tree* t *is described by the paths* π *from the root* $t(\epsilon)$ *to the nodes* $t(\pi)$ *of the tree, and the nodes of the tree are labeled with elements of* S.

Definition 4.9 *A child of node* π *in tree* t *is any path* $\pi \cdot i$ *that is in the domain of* t, *where* i *is some positive integer. If* π *is in the domain of* t, *then the subtree of* t *rooted at* π, *written* $t\backslash\pi$, *is the partial function* $t'(\pi') = t(\pi \cdot \pi')$. *Also,* $node(L, T_1, \ldots, T_n)$ *denotes a constructor of an* S*-tree with root labeled* L *and subtrees* T_i, *where* $L \in S$ *and each* T_i *is an* S*-tree, such that* $1 \leq i \leq n$, $node(L, T_1, \ldots, T_n)(\epsilon) = L$, *and* $node(L, T_1, \ldots, T_n)(i \cdot \pi) = T_i(\pi)$.

Idealized proofs are trees of ground atoms, such that a parent is deduced from the idealized proofs of its children according to the same process used to define the alternating fixed-point model.

Definition 4.10 *The set of idealized proofs of a stratum of* P *equals* $\mu\Sigma_P^v$ *if* v *is inductive and* $\nu\Sigma_P^v$ *if* v *is coinductive, such that* R *is the union of the sets of idealized proofs of the strata strictly lower than* v *and*

$$\begin{aligned}\Sigma_P^v(S) = \quad & R \cup \{node(q\left(\widehat{t}\right), T_1, \ldots, T_n) \mid q \in v \wedge T_i \in S \wedge \\ & [q\left(\widehat{t}\right) \leftarrow D_1, \ldots, D_n] \in G^{co}(P) \wedge T_i(\epsilon) = D_i\}\end{aligned}$$

Note that these definitions mirror the definitions defining models, with the exception that the elements of the sets record the application of the program clauses as a tree of atoms.

Definition 4.11 *The set of idealized proofs generated by a co-logic program P, written Σ_P, is the union of the sets of idealized proofs of every stratum of P.*

Again, this is nothing more than a reformulation of the alternating fixed-point model, which records the applications of the generator in the elements of the fixed points, as the following theorem demonstrates.

Theorem 4.1 *Let $S = \{A \mid \exists T \in \Sigma_P . A$ is the root of $T\}$, then $S = M(P)$.*

Hence any element in the model has an idealized proof and anything that has an idealized proof is in the model. This formulation of the declarative semantics in terms of idealized proofs will be used in section 4.2.3 in order to distinguish between the case when a query has a finite derivation, from the case when there are only infinite derivations of the query, in the operational semantics.

4.2.2 Operational Semantics

The operational semantics given for co-logic programming is defined as an interleaving of *SLD* [7] and co-*SLD* [8, 9, 10]. The semantics implicitly defines a state transition system. Systems of equations are used to model the part of the state involving unification. The current state of the pending goals is modeled using a forest of finite trees of atoms, as it is necessary to be able to recognize infinite proofs of coinductive predicates. However, an implementation that uses a policy of executing goals in the current resolvent from left to right (as in standard logic programming), only needs a single stack (see section 4.3).

Definition 4.12 *A state S is a pair (F, E), where F is a finite multi-set of finite trees, i.e., a forest of syntactic atoms, and E is a system of equations.*

Definition 4.13 *A transition rule R of a co-logic program P is an instance of a clause in P, with variables standardized apart, i.e., consistently renamed for freshness, or R is a coinductive hypothesis rule of the form $\nu(\pi, \pi')$, where π and π' are both paths, such that π is a proper prefix of π'.*

Before we can define how a transition rule affects a state, we must define how a tree in a state is modified when an atom is proved to be true. This is called the unmemo function, and it removes memo'ed atoms that are no longer necessary. Starting at a leaf of a tree, the unmemo function removes the leaf and the maximum number of its ancestors, such that the result is still a tree. This involves iteratively removing ancestor nodes of the leaf until an ancestor is reached, which still has other children, and so removing any more ancestors would cause the result to no longer be a tree, as children would be orphaned. When all nodes in a tree are removed, the tree itself is removed.

Definition 4.14 *The unmemo function δ takes a tree of atoms T, a path π in the domain of T, and returns a forest. Let $\rho(T, \pi \cdot i)$ be the partial function equal to T, except that it is undefined at $\pi \cdot i$. $\delta(T, \pi)$ is defined as follows:*

$$\begin{array}{lcll} \delta(T,\pi) & = & \{T\} & , \text{ if } \pi \text{ has children in } T \\ \delta(T,\epsilon) & = & \emptyset & , \text{ if } \epsilon \text{ is a leaf in } T \\ \delta(T,\pi) & = & \delta(\rho(T,\pi),\pi') & , \text{ if } \pi = \pi' \cdot i \text{ is a leaf in } T \end{array}$$

The intuitive explanation of the following definition is that (1) a state can be transformed by applying the coinductive hypothesis rule $\nu(\pi, \pi')$, whenever in some tree, π is a proper ancestor of π', such that the two atoms unify. Also, (2) a state can be transformed by applying an instance of a definite clause from the program. In either case, when a subgoal has been proved true, the forest is pruned so as to remove unneeded memos. Also, note that the body of an inductive clause is overwritten on top of the leaf of a tree, as an inductive call need not be memo'ed, since the coinductive hypothesis rule can never be

invoked on a memo'ed inductive predicate. When the leaf of the tree is also the root, this causes the old tree to be replaced with, one or more singleton trees. Coinductive subgoals, on the other hand, need to be memo'ed, in the form of a forest, so that infinite proofs can be recognized.

The state transition system may be nondeterministic, depending on the program, that is, it is possible for states to have more than one outgoing transition as the following definition shows (implementations typically use backtracking to realize non-deterministic execution; see section 4.3). We write $S - x$ to denote the multi-set obtained by removing an occurrence of x from S.

Definition 4.15 *Let $T \in F$. A state (F, E) transitions to another state $((F - T) \cup F', E')$ by transition rule R of program P whenever:*

1. R is an instance of the coinductive hypothesis rule of the form $\nu(\pi, \pi')$, p is a coinductive predicate, π is a proper prefix of π', which is a leaf in T, $T(\pi) = p(t'_1, \ldots, t'_n)$, $T(\pi') = p(t_1, \ldots, t_n)$, E' is the most general unifier for $\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E$, and $F' = \delta(T, \pi')$.

2. R is a definite clause of the form
   $p(t'_1, \ldots, t'_n) \leftarrow B_1, \ldots, B_m$, π is a leaf in T, $T(\pi) = p(t_1, \ldots, t_n)$, E' is the most general unifier for $\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E$, and the set of trees of atoms F' is obtained from T according to the following case analysis of m and p:

   (a) Case $m = 0$: $F' = \delta(T, \pi)$.

   (b) Case $m > 0$ and p is coinductive: $F' = \{T'\}$ where T' is equal to T except at $\pi \cdot i$, & $T'(\pi \cdot i) = B_i$, for $1 \leq i \leq m$.

(c) Case $m > 0$ and p is inductive: If $\pi = \epsilon$ then $F' = \{node(B_i) \mid 1 \leq i \leq m\}$. Otherwise, $\pi = \pi' \cdot j$ for some positive integer j. Let T'' be equal to T except at $\pi' \cdot k$ for all k, where T'' is undefined. Finally, $F' = \{T'\}$ where T' is equal to T'' except at $\pi' \cdot i$, where $T'(\pi' \cdot i) = B_i$, for $1 \leq i \leq m$, and $T'(\pi' \cdot (m + k)) = T(\pi' \cdot k)$, for $k \neq j$.

Definition 4.16 *A transition sequence in program P consists of a sequence of states $S_1 S_2, \ldots$ and a sequence of transition rules $R_1, R_2 \ldots$, such that S_i transitions to S_{i+1} by rule R_i of program P.*

A transition sequence denotes the trace of an execution. Execution halts when it reaches a terminal state: either all atoms have been proved or the execution path has reached a dead-end.

Definition 4.17 *The following are two distinguished terminal states:*

1. An accepting state is a state of the form $(\emptyset, E)$, where $\emptyset$ denotes the empty set.
2. A failure state is a non-accepting state lacking any outgoing transitions.

Finally we can define the execution of a query as a transition sequence through the state transition system induced by the input program, with the start state consisting of the initial query.

Definition 4.18 *A derivation of a state (F, E) in program P is a state transition sequence with the first state equal to (F, E). A derivation is successful if it ends in an accepting state, and a derivation has failed if it reaches a failure state. We say that a list of syntactic atoms $A_1, \ldots, A_n$, also called a goal or query, has a derivation in program P, if $(\{node(A_i) \mid 1 \leq i \leq n\}, \emptyset)$ has a derivation in P.*

An implementation will use backtracking search in order to find a successful derivation.

So the operational semantics for co-logic programming is an alternation of *SLD* and co-*SLD*. From one point of view, this new operational semantics can be seen as recording the coinductive subgoals that have resulted in the current state, while from the other point of view, the new operational semantics can be seen as a variation of co-*SLD* that does not include inductive predicates in the synthesized coinductive hypothesis. In other words, the operational semantics allows for the combination of both kinds of execution, as the following example demonstrates.

Example 4.2 *The following is a variation of a similar program from the previous chapter. Consider the execution of the following program, which defines a predicate that recognizes infinite streams of natural numbers. Note that only the* `stream/1` *predicate is coinductive, while the* `number/1` *predicate is inductive.*

```
coinductive(stream, 2).
stream([H | T]) :- number(H), stream(T).


inductive(number, 2)
number(0).
number(s(N)) :- number(N).


| ?- stream([0, s(0), s(s(0)) | T ]).
```

The following is an execution trace for the above query, of the memoization of calls by the operational semantics. Note that calls of `number/1` are not memo'ed because `number/1` is inductive. Compare this trace to the one from the previous chapter, where both `stream/1` and `number/1` were coinductive.

1. `MEMO: stream( [ 0, s(0), s(s(0)) | T ])`
2. `MEMO: stream( [ s(0), s(s(0)) | T ] )`
3. `MEMO: stream( [ s(s(0)) | T ] )`

The next goal call is `stream(T)`, which unifies with the first memo'ed ancestor, and therefore immediately succeeds. Hence the original query succeeds with

```
T = [ 0, s(0), s(s(0)) | T ]
```

`stream(T)` is true whenever `T` is some list of natural numbers. If `number/1` was also coinductive, as was the case in the previous chapter, then `stream(T)` would be true whenever `T` is a list containing either natural numbers or ω, i.e., infinity, which is represented as an infinite application of successor `s(s(s(...)))`. This difference in an inductive versus coinductive semantics for `number/1` was seen in the previous chapter. There the same program (sans the predicate annotation) also memo'ed the `number/1` predicate each time it was called, as all predicates are implicitly coinductive in coinductive logic programming. This caused an additional spurious ω element to appear in the infinite streams of numbers. The original intent of the `number/1` predicate was to denote the natural numbers, and since ω is not a natural number, it is undesirable to use a coinductive semantics for `number/1`. With coinductive logic programming and traditional logic programming, there is no choice. In coinductive logic programming, the "ω problem" occurs because `number/1` should have inductive semantics, but in traditional logic programming, the `stream/1` predicate has no meaning. So both kinds of logic programming are simultaneously needed for this program.

Example 4.3 *This example further illustrates that some predicates are naturally defined inductively, while other predicates are naturally defined coinductively. The* `member/2` *predicate is an example of an inherently inductive predicate.*

```
member( H, [ H | _ ] ).
member( H, [ _ | T ] ) :- member( H, T ).
```

If this predicate was declared to be coinductive, then `member( X, L )` *is true whenever* `X` *is in* `L` *or whenever* `L` *is an infinite list, even if* `X` *is not in* `L`*!*

Example 4.4 *The previous example, whether declared coinductive or not, states that the desired element is the last element of some prefix of the list, as the following equivalent reformulation of* `member/2`*, called* `membera/2` *demonstrates, where* `drop/3` *drops a prefix ending in the desired element and returns the resulting suffix.*

```
inductive(membera, 2).
membera( X, L ) :- drop( X, L, _ ).


inductive(drop, 3).
drop( H, [ H | T ], T ).
drop( H, [ _ | T ], T1 ) :- drop( H, T, T1 ).
```

When the `membera/2` *and* `drop/3` *predicates are inductive, the prefix ending in the desired element must be finite, but when the* `membera/2` *and* `drop/3` *predicates are declared coinductive, the prefix may be infinite. Since an infinite list has no last element, it is trivially true that the last element unifies with any other term. This explains why the above definition, when declared to be coinductive, is always true for infinite lists regardless of the presence of the desired element.*

In logic programming, the `member/2` predicate is used quite often, and so it would also be nice to be able to use it along with coinductive predicates. This is always possible in co-logic programming because the `member/2` predicate is recursively defined in terms of itself. Therefore it stands no chance of playing a role in violating the necessary stratification restriction.

Example 4.5 *A mixture of inductive and coinductive predicates can be used to define a variation of* `member/2`*, called* `comember/2`*, which is true if and only if the desired element occurs an infinite number of times in the list. Hence it is false when the element does not occur in the list or when the element only occurs a finite number of times in the list. On the other hand, if* `comember/2` *was declared inductive, then it would always be false. Hence coinduction is a necessary extension.*

```
coinductive(comember, 2).
comember( X, L ) :- drop( X, L, L1 ), comember( X, L1 ).

?- X = [ 1, 2, 3 | X ], comember( 2, X ).
Answer: yes.

?- X = [ 1, 2, 3, 1, 2, 3 ], comember( 2, X ).
Answer: no.

?- X = [ 1, 2, 3 | X ], comember( Y, X ).
Answer: Y = 1; Y = 2; Y = 3;
```

4.2.3 Correctness

This section proves the correctness of the operational semantics by demonstrating a correspondence between the declarative and operational semantics via soundness and completeness theorems. Completeness, however, must be restricted to atoms that have a rational proof, as termination cannot be guaranteed for atoms with only irrational proofs. Chapter 8 discusses an extension of the operational semantics, so as to improve its completeness.

Lemma 4.1 *If (A, E_1) has a successful derivation in program P, with final state $(\emptyset, E_2)$, then (A, E_3) has a successful derivation of the same length, in program P, where $E_2 \subseteq E_3$, with each state of the derivation of the form (F, E_3) for some forest F of finite trees of atoms.*

Proof 4.1 *Let (A, E_1) have a successful derivation in program P ending with state $(\emptyset, E_2)$. In the sequence of states, the system of equations monotonically increases, and so the monotonicity of unification with infinite terms implies (A, E_3) has a successful derivation in program P, where $E_2 \subseteq E_3$, with each state of the derivation of the form (F, E_3) for some forest F of finite trees of atoms.*

Lemma 4.2 *If A has a successful derivation in program P, which first transitions to the second state by applying clause $A' \leftarrow B_1, \ldots, B_n$, such that $E(A) = E(A')$, then each (B_i, E) also has a successful derivation in program P.*

Proof 4.2 *Let A have a successful derivation in program P, which first transitions to the next state by applying clause $A' \leftarrow B_1, \ldots, B_n$, such that $E(A) = E(A')$. A derivation for (B_i, E) can be created by mimicking each transition that modifies the (sub)tree rooted*

at B_i in the original derivation, except for the transitions which are of the form $\nu(\pi, \pi')$, which are no longer correct derivations because the parent of B_i, that is, A, no longer exists. In the case that $\pi = i \cdot \pi_0$ and $\pi' = i \cdot \pi_1$ for some number i and path π_0, instead apply the transition rule $\nu(\pi_0, \pi_1)$ to the corresponding leaf to which the original derivation would have applied $\nu(\pi, \pi')$. Otherwise, when $\pi = \epsilon$, a coinductive transition rule cannot be applied to the corresponding leaf. Instead, recursively mimic the transitions of the entire original derivation of A.

Lemma 4.3 *If (A, E) has a successful derivation in program P, then $E'(E(A))$ is true in program P, where $(\emptyset, E')$ is the final state of the derivation.*

Proof 4.3 *Since the model of P consists of the union of the models of the strata of P, it is sufficient to show that if (A, E) has a successful derivation in program P, then all groundings of $E'(E(A))$ are included in the model for the stratum in which A resides.*

The proof proceeds by induction on the height of the strata of P. Let Q_v be the set of all groundings ranging over the $U^{co}(P)$ of all such $E'(E(A))$ that are either in the same stratum v or some lower stratum. We prove by induction on the height of v that Q_v is contained in the model for v.

Consider the case when the stratum v is coinductive. We show that $Q_v \subseteq \nu T_P^v$. The proof proceeds by coinduction. Let $A' \in Q_v$, then $A' = E_1(E_2(E_3(A)))$, where E_1 is a grounding substitution for $E_2(E_3(A))$ and (A, E_3) has a successful derivation ending in $(\emptyset, E_2)$. By lemma 4.1, (A, E) has a successful derivation, where $E = E_1 \cup E_2 \cup E_3$. The case when $E(A)$ unifies with a fact is trivial, and the case when $E(A)$ is in a lower stratum than v follows by induction on the height of strata. Now, consider the case where the derivation begins with an application of a program clause $A'' \leftarrow B_1, \ldots, B_n$, resulting in the state $(node(A, [B_1, \ldots, B_n]), E)$, where $E(A) = E(A'')$. By lemma 4.2

each state (B_i, E) has a successful derivation. Let E' be a grounding substitution for the clause $E(A'' \leftarrow B_1, \ldots, B_n)$, such that $C = E'(E(A'' \leftarrow B_1, \ldots, B_n)) \in G^{co}(P)$, then $C = A' \leftarrow E''(B_1), \ldots, E''(B_n)$, where $E'' = E' \cup E$. By lemma 4.1, each (B_i, E'') has a successful derivation, and the stratification restriction implies that each $E''(B_i)$ is in a stratum equal to or lower than v. Hence $E''(B_i) \in Q_v$. Therefore, by the principle of coinduction, $Q_v \subseteq \nu T_P^v$.

Now consider the case when the stratum v is inductive. It is sufficient to prove that $Q_v \subseteq \mu T_P^v$. Let $A' \in Q_v$, then $A' = E_1(E_2(E_3(A)))$, where E_1 is a grounding substitution for $E_2(E_3(A))$ and (A, E_3) has a successful derivation ending in $(\emptyset, E_2)$. By lemma 4.1, (A, E) has a successful derivation, where $E = E_1 \cup E_2 \cup E_3$. The case when A' is in a lower stratum than v follows by induction. Now consider the case when A' does not occur in a stratum lower than v, that is, A' is an inductive atom. The proof proceeds by induction on the length of the derivation A'. When the derivation consists of just one transition, then A' unifies with a fact A in program P, and hence $A' \in \mu T_P^v$. Finally, consider the case when the derivation is of length $k > 1$. Then the derivation begins with an application of a program clause $A'' \leftarrow B_1, \ldots, B_n$, where $E(A) = E(A'')$. By lemma 4.2 each state (B_i, E) has a successful derivation. Let E' be a grounding substitution for the clause $E(A'' \leftarrow B_1, \ldots, B_n)$, such that $C = E'(E(A'' \leftarrow B_1, \ldots, B_n)) \in G^{co}(P)$, then $C = A' \leftarrow E''(B_1), \ldots, E''(B_n)$, where $E'' = E' \cup E$. By lemma 4.1, each (B_i, E'') has a successful derivation of length $k' < k$, and the stratification restriction implies that each $E''(B_i)$ is in a stratum equal to or lower than v. If $E''(B_i)$ is in a strictly lower stratum than v, then by induction on strata $E''(B_i) \in \mu T_P^v$, and if $E''(B_i)$ is not in a lower stratum, then since it has a derivation of length $k' < k$, by induction on the length of the derivation, $E''(B_i) \in \mu T_P^v$. Therefore, $A' \in \mu T_P^v$.

Theorem 4.2 *(soundness) If the query $A_1, \ldots, A_n$ has a successful derivation in program*

P, then $E(A_1, \ldots, A_n)$ is true in program P, where E is the resulting variable bindings for the derivation.

Proof 4.4 *If the goal $A_1, \ldots, A_n$ has a successful derivation in program P, with E as the resulting variable bindings for the derivation, then each $E(A_i)$ independently has a successful derivation in program P. By lemma 4.3, each $E(A_i)$ is true in program P.*

Theorem 4.3 *(completeness) Let $A_1, \ldots, A_n \in M(P)$. If each $A_1, \ldots, A_n$ has a rational idealized proof, then the query $A_1, \ldots, A_n$ has a successful derivation in program P.*

Proof 4.5 *Without loss of generality, we only consider the case when the query is a single atom, i.e., $n = 1$, as the case for $n = 0$ is trivial and the case for $n > 1$ follows by simply composing the individual derivations of each atom of the original query.*

Let $A \in M(P)$ have a rational idealized proof T. The derivation is constructed by recursively applying the clause corresponding to each node encountered along a depth-first traversal of the idealized proof tree to the corresponding leaf in the current state. In order to ensure that the derivation is finite, the traversal stops at the coinductive root $\pi = \pi' \cdot \pi''$ of a subtree that is identical to a subtree rooted at a proper coinductive ancestor π'. Then the derivation applies a transition rule of the form $\nu(\pi', \pi)$ to the leaf corresponding to R in the current state, and finally the depth-first traversal continues traversing starting at a node in the idealized proof tree corresponding to some leaf in the current state of the derivation.

The fact that T is rational implies that the set of all subtrees of T is finite in cardinality. Furthermore, the stratification restriction prevents a depth-first traversal from encountering the same subtree twice along the same path, such that the subtree has an inductive atom at its root. Only subtrees rooted at coinductive atoms can repeat in such

a fashion. So the maximum depth of the traversal in the idealized proof tree is finite. Combined with the fact that all idealized proofs are finitely branching, this implies that the traversal always terminates. So the constructed derivation is finite.

It remains to prove that the final state of the constructed derivation is a success state. The traversal stops going deeper in the idealized proof tree due to two cases: the traversal reaches a leaf in the idealized proof tree or the traversal encountered a subtree identical to an ancestor subtree. In either case, the derivation removes the corresponding leaf in the current state, as well as the maximal number of ancestors of the corresponding leaf such that the result is still a tree. So a leaf only remains in the state when its corresponding node in the proof tree has yet to be traversed. Since every node in the idealized proof tree corresponding to a leaf in the state is traversed at some point, the final state's tree contains no leaves, and hence the final state has an empty forest, which is the definition of an accept state. Therefore, A has a successful derivation in program P.

4.3 Implementation

Following the examples of both the interpreter for traditional logic programming and the interpreter for coinductive logic programming, the interpreter for co-logic programming uses a left-most, depth-first search strategy for searching for a successful alternating *SLD* and co-*SLD* derivation of a given query, within a given co-logic program. However, the interpreter must also decide when to it is appropriate and correct to apply the coinductive hypothesis rule. As was the case with coinductive logic programming, the purpose of this rule is to provide a finite derivation of an atom that would otherwise have an infinite derivation, but now that both inductive and coinductive predicates are mixed, it is only appropriate to invoke the coinductive hypothesis rule on coinductive atoms. So the interpreter uses a "hypothesis-first" strategy for coinductive atoms only, yielding

the hypothesis-first, left-most, depth-first search strategy. The other way in which the co-logic programming interpreter differs from the interpreter for coinductive logic programming is that the `solve` predicate only adds coinductive atoms to the `Hypothesis` argument.

Figure 4.1 shows the source code listing of "`colp.pro`", which is the interpreter for co-logic programming that runs on SICStus Prolog. The interpreter is used by including the "`colp.pro`" file in every co-logic program source file. As was the case for the coinductive logic programming interpreter presented in section 3.3, due to a limitation in the SICStus Prolog implementation of the `clause` predicate, all user defined predicates must be declared dynamic, as demonstrated in the example co-logic program "`example2.clp`" depicted in figure 4.2, and it is also forbidden to redefine a built in predicate. In addition, inductive and coinductive predicates must be declared so by including the clause `inductive( PREDICATE, ARITY )` and `coinductive( PREDICATE, ARITY )` respectively in the source program, where `PREDICATE` is the name of the predicate and `ARITY` is its arity.

A query is executed by loading the coinductive logic program source file as if it were a normal SICStus program. Then queries must begin and end, with `"query(("` and `"))."` respectively in order to be executed with alternating *SLD* and co-*SLD* semantics using a coinductive hypothesis first, left-most, depth-first search strategy.

```
query( Goal ) :- solve( [], Goal    ).

solve( Hypothesis, ( Goal1, Goal2 ) ) :-
  solve( Hypothesis, Goal1 ),
  solve( Hypothesis, Goal2 ).

solve( _          , Atom            ) :-
  builtin( Atom ),
  Atom.
solve( Hypothesis, Atom             ) :-
  coinductive( Atom ),
  member( Atom, Hypothesis ).
solve( Hypothesis, Atom             ) :-
  coinductive( Atom ),
  clause( Atom, Atoms ),
  solve( [ Atom | Hypothesis ], Atoms ).
solve( Hypothesis, Atom           ) :-
  inductive( Atom ),
  clause( Atom, Atoms ),
  solve( Hypothesis, Atoms ).

inductive( Atom ) :-
  Atom \= ( _, _ ),
  \+ builtin( Atom ),
  functor( Atom, Predicate, Arity ),
  inductive( Predicate, Arity ).

coinductive( Atom ) :-
  Atom \= ( _, _ ),
  \+ builtin( Atom ),
  functor( Atom, Predicate, Arity ),
  coinductive( Predicate, Arity ).
```

Figure 4.1. Co-Logic Programming Interpreter "`colp.pro`"

```
:- include( 'colp2.pro' ).

:- dynamic num/1.
:- dynamic stream/1.
:- dynamic append1/3.
:- dynamic member1/2.
:- dynamic member2/2.
:- dynamic drop/3.
:- dynamic comember/2.

inductive( num, 1 ).
num( 0 ).
num( s( N ) ) :- num( N ).

coinductive( stream, 1 ).
stream( [ H | T ] ) :- num( H ), stream( T ).

coinductive( append1, 1 ).
append1( [], X, X ).
append1( [ H | T ], Y, [ H | Z ] ) :- append1( T, Y, Z ).

coinductive( member1, 2 ).
member1( X, [ X | _ ] ).
member1( X, [ _ | T ] ) :- member1( X, T ).

inductive( member2, 2 ).
member2( X, [ X | _ ] ).
member2( X, [ _ | T ] ) :- member2( X, T ).

inductive( drop, 3 ).
drop( H, [ H | T ], T ).
drop( H, [ _ | T ], T1 ) :- drop( H, T, T1 ).

coinductive( comember, 2 ).
comember( X, L ) :- drop( X, L, L1 ), comember( X, L1 ).
```

Figure 4.2. Coinductive Logic Programming Examples "`example2.clp`"

CHAPTER 5
APPLICATION: MODEL CHECKING

Model checking is a formal verification method that can be applied to the verification of both hardware and software systems. Properties of a system are verified by first modelling the system as a Kripke structure (the model), and then a specific property is specified via some logical formalism, such as temporal logic. A Kripke structure is a nondeterministic finite state machine, with states labelled with formal properties that hold in the respective state. The property is verified by searching the model for a counterexample that demonstrates that the specified property is false [57]. Furthermore, most properties of models fall into one of two categories: (1) *safety* properties, which state that "nothing bad will happen", and (2) *liveness* properties, which state that "something good will happen".

One of the applications of co-logic programming is its ability to directly represent and verify properties of Kripke structures and ω-automata [4, 5], which are automata that accept infinite strings. It is well known that traditional logic programming can be used

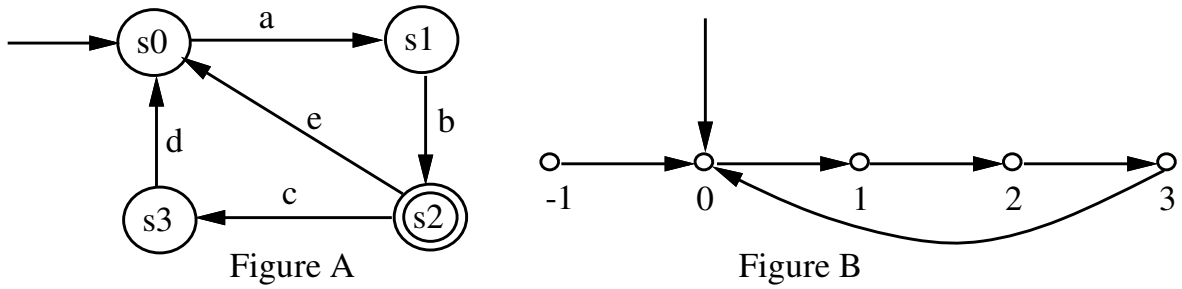

Figure 5.1. Example Automata

to directly reason about model checking in the presence of finite input strings [58], model checking ω-automata can be directly reasoned about using co-logic programming. For example, consider figure 5.1 (taken from [4]), which depicts an automata that accepts finite strings, and compare this to its direct encoding into a traditional logic program depicted in figure 5.2 (taken from [4]).

```
automata([X|T], St) :- trans(St, X, NewSt), automata(T, NewSt).
automata([], St) :- final(St).

trans(s0, a, s1).                          trans(s1, b, s2).
trans(s2, c, s3).                          trans(s3, d, s0).
trans(s2, e, s0).                          final(s2).
```

Figure 5.2. Example Encoding of an Automata

Using a traditional logic programming system, the query `automata(X, s0)` can be used to enumerate all finite strings accepted by the automata in figure 5.1. Now assume that we want to treat the automata in figure 5.1 as an ω-automata, that is, the automata accepts an infinite string if the final states are traversed an infinite number of times. The co-logic program that simulates this variation of the automata can be obtained by simply dropping the base case as depicted in figure 5.3.

```
coinductive(automata, 2).
automata([X|T], St) :- trans(St, X, NewSt), automata(T, NewSt).

trans(s0, a, s1).                          trans(s1, b, s2).
trans(s2, c, s3).                          trans(s3, d, s0).
trans(s2, e, s0).                          final(s2).
```

Figure 5.3. Example Encoding of an ω-automata

The query becomes "`automata(X, s0), comember(s2, X)`", where `comember/2` is

defined in section 4.2.2, and for the example it yields the solutions `X = [a, b, c, d | X]` and `X = [a, b, e | X]`.

5.1 Liveness Properties

The application of co-logic programming becomes apparent once it is noticed that safety and liveness are dual to each other.

Safety properties can be checked using reachability analysis. If a counterexample to the given property exists, then it can be found by enumerating every reachable state of the model, and since the model is assumed to have finitely many states, the counterexample will be found in finite time. Since the reachability set is naturally defined as a least fixed-point, it can be elegantly encoded into a traditional logic program [59].

Previous research shows that the inductive reachability technique described at the beginning of this chapter is not suitable for verifying the general class of liveness properties [60]. Furthermore, checking liveness properties can be reduced to verification of termination, under the assumption of fairness [61], and fairness properties can be defined in terms of alternating least and greatest fixed-point temporal logic formulas [62].

So the same thing cannot be said about encoding liveness properties into traditional logic programs. Counterexamples to liveness properties are represented as infinite paths through the model, and infinite paths are naturally defined in terms of greatest fixed-points, which are not directly contained in the semantics of traditional logic programming, which is based on least fixed-points. Traditional logic programming systems mitigate this problem via a transformation of the given logical formula denoting the property into a semantically equivalent least fixed-point formula, which can then be directly encoded in the form of a traditional logic program [59]. Since this transformation is quite complex and uses numerous nested negations, it introduces unnecessary semantic complexity and

execution overhead.

Co-logic programming, on the other hand, can directly encode model checking liveness properties because it can directly compute counterexamples that involve greatest fixed-point formulas, without requiring any transformations or new negations. A given state is said to be "not live", whenever it can be reached by transitioning through a loop of states. The reason for this is that in such a case, it is possible to never enter the given state by indefinitely traversing the loop. Hence liveness counterexamples can be found by using coinduction to enumerate every state that can be reached from an infinite loop and such that the state constitutes a valid counterexample to the liveness property being checked.

The method for encoding liveness properties into co-logic programs, described in this chapter, works by reducing the problem to verifying that the model satisfies the fairness constraint. This works by composing the co-logic program that encodes the original model with a co-logic program, which encodes the fairness constraint, and a co-logic program that encodes the negation of the liveness property. The resulting program is then queried to check if the initial state of the model is present in the alternating fixed-point semantics (as defined in section 4.2.1) of the co-logic program. If the alternating fixed-point contains this state, then there exists a counterexample that violates the specified liveness property. Otherwise, there is no counterexample, and therefore the model satisfies the given liveness property.

Take for example a modulo 4 counter depicted as "Figure B" in figure 5.1 (adapted from [63, 4]). Correctness of the counter dictates that along every path through the system, the state labelled with -1 is not reached. In other words, there is an infinite path through the system that never passes through the state labelled with -1. Since this property can be elegantly specified via a greatest fixed-point formula, it can be verified

coinductively. Figure 5.4, taken from [4], demonstrates how to encode this problem as a co-logic program. The negation of the property, that is, `N1 >= 0`, is composed with the counter program, and s0, s1, s2, s3, and sm1 denote the states labelled with 0, 1, 2, 3, and -1 respectively.

```
:- coinductive s0/2, s1/2, s2/2, s3/2, sm1/2.
sm1(N,[sm1|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
s0(N,[s0|T]) :- N1 is N+1 mod 4, s1(N1,T), N1>=0.
s1(N,[s1|T]) :- N1 is N+1 mod 4, s2(N1,T), N1>=0.
s2(N,[s2|T]) :- N1 is N+1 mod 4, s3(N1,T), N1>=0.
s3(N,[s3|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
```

Figure 5.4. Encoding a Liveness Property

When run, the query `sm1(-1,X), comember(sm1,X)` will fail, which implies that there is no counterexample to the original property. Therefore the original property holds for the model. As can be seen by the simplicity of the encoding, the benefit of using co-logic programming as opposed to traditional logic programming is that no complicated transformations of the model or property are necessary in order to work around the lack of coinduction in traditional logic programming. The transformation used to encode liveness properties into traditional logic programs has been reported to increase time and space complexity as much as six-fold [63].

5.2 Timed Automata

Timed automata extend ω-automata with real-time clocks or "stopwatches" [64]. Hence they can be encoded in a straightforward manner into co-logic programming extended with constraints over the reals, in a manner similar to the work of Gupta et al. [58]. Figure 5.5 (taken from [9]) contains a co-logic program with $CLP(\mathcal{R})$ constraints for modeling the classic train-gate-controller problem (see Gupta et al. [58]) depicted in

figure 5.6.

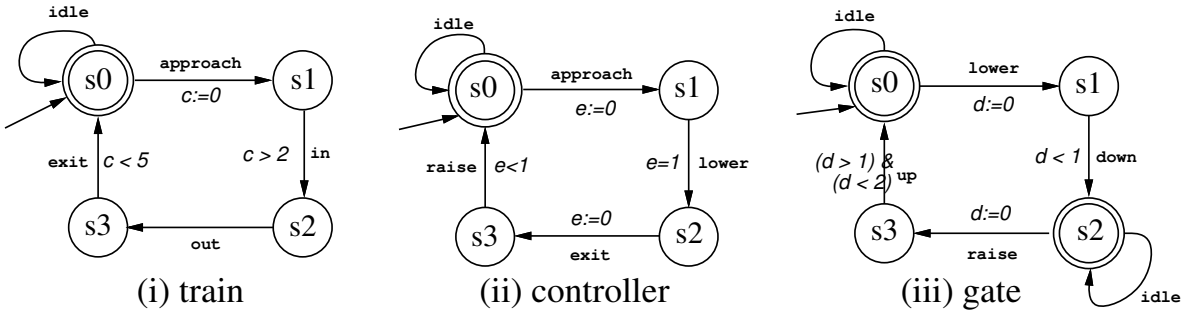

Figure 5.6. Train-Gate-Controller Timed Automata

The co-logic program can be queried for all infinite timed sequences of events that are accepted by the controller. Again, safety and liveness properties can be verified via the typical search for a counterexample. As demonstrated in [9, 4], when given the query in figure 5.7, the following results also depicted in the same figure are obtained, where `A`, `B`, ..., and `I` are the times when the corresponding events occur.

```
| ?- driver(s0, s0, s0, T, Ta, Tb, Tc, X, R).
R = [(approach,A),(lower,B),(down,C),(in,D),(out,E),
                (exit,F),(raise,G),(up,H)|R],
X = [approach,lower,down,in,out,exit,raise,up | X] ? ;

R= [(approach,A),(lower,B),(down,C),(in,D),(out,E),
                (exit,F),(raise,G),(approach,H),(up,I)|R],
X = [approach,lower,down,in,out,exit,raise,approach,up|X] ? ;
```

Figure 5.7. Train-Gate-Controller Query

A coinductive predicate `cosublist/2` is defined in figure 5.8 to check that the `down` event is always followed by the `in` event in the infinite event sequence. This predicate simply checks that its first argument occurs as a sublist infinitely often in its second

```
:- use_module(library(clpr)).
:- coinductive driver/9.

train(X,up,X,T1,T2,T2).
train(s0,approach,s1,T1,T2,T3) :-
        {T3 = T1}.
train(s1,in,s2,T1,T2,T3) :-
        {T1 - T2 > 2,
        T3 = T2}.
train(s2,out,s3,T1,T2,T2).
train(s3,exit,s0,T1,T2,T3) :-
        {T3 = T2,
        T1 - T2 < 5}.
train(X,lower,X,T1,T2,T2).
train(X,down,X,T1,T2,T2).
train(X,raise,X,T1,T2,T2).

gate(s0,lower,s1,T1,T2,T3) :-
                {T3 = T1}.
gate(s1,down,s2,T1,T2,T3) :-
                {T3 = T2,
                 T1 - T2 < 1}.
gate(s2,raise,s3,T1,T2,T3) :-
                {T3 = T1}.
gate(s3,up,s0,T1,T2,T3) :-
               {T3 = T2, T1 - T2 > 1,
                T1 - T2 < 2}.
gate(X,approach,X,T1,T2,T2).
gate(X,in,X,T1,T2,T2).
gate(X,out,X,T1,T2,T2).
gate(X,exit,X,T1,T2,T2).

contr(s0,approach,s1,T1,T2,T1).
contr(s1,lower,s2,T1,T2,T3) :- {T3 = T2, T1 - T2 = 1}.
contr(s2,exit,s3,T1,T2,T1).
contr(s3,raise,s0,T1,T2,T2) :- {T1-T2 < 1}.
contr(X,in,X,T1,T2,T2).
contr(X,up,X,T1,T2,T2).
contr(X,out,X,T1,T2,T2).
contr(X,down,X,T1,T2,T2).

driver(S0,S1,S2,T,T0,T1,T2,[X|Rest],[(X,T)|R]) :-
        train(S0,X,S00,T,T0,T00),
        contr(S1,X,S10,T,T1,T10) ,
        gate(S2,X,S20,T,T2,T20),
        {TA > T},
        driver(S00,S10,S20,TA,T00,T10,T20,Rest,R).
```

Figure 5.5. Train-Gate-Controller Program

argument, and hence `cosublist([down,in], X)` can be used to verify this property of the sequence of events `X`.

```
coinductive(cosublist, 2).
cosublist(X, Y) :-
  drop_until(X, Y, Remaining),
  cosublist(X, Remaining).

inductive(drop_prefix, 3).
drop_prefix([], Result, Result).
drop_prefix([H|T], [H|Y], Result) :- drop_prefix(T, Y, Result).

inductive(drop_until, 3).
drop_until(X, Y, Result) :- drop_prefix(X, Y, Result).
drop_until(X, [_|T], Result) :- drop_until(X, T, Result).
```

Figure 5.8. `cosublist`

5.3 Self Healing Systems

In this section we explore an application of co-logic programming to reasoning about self healing systems. A system is considered to be self healing or self correcting, when it is designed to recover itself from errors or failure states.

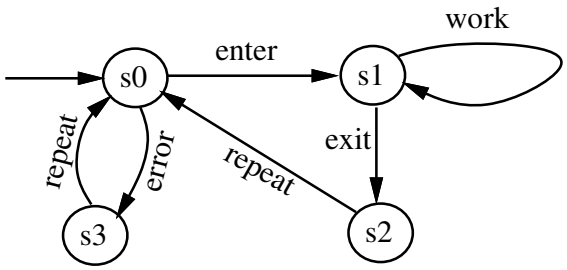

Figure 5.9. Automata Modeling a Self Correcting System

The following example, taken from [5], illustrates the co-logic programming approach.

```
:- coinductive state/2.
state(s0,[s0,is1|T]) :- enter, work, state(s1,T).
state(s1,[s1|T]) :- exit, state(s2,T).
state(s2,[s2|T]) :- repeat, state(s0,T).
state(s0,[s0|T]) :- error, state(s3,T).
state(s3,[s3|T]) :- repeat, state(s0,T).
work :- state(is1), enter.
exit.
repeat.
error.
state(is1) :- state(is1).
state(is1).
```

Figure 5.10. Encoding of the Self Correcting System

The model shown in figure 5.9 consists of four states `s0`, `s1`, `s2` and `s3`. The system starts in state `s0`, enters state `s1`, performs a finite amount of work in state `s1` and then exits to state `s2`. From state `s2` the system transitions back to state `s0`, and repeats the entire loop again, an infinite number of times. However, when the system encounters an error, possibly due to a hardware failure, the system transitions into the error recovery state `s3`, where corrective action is taken returning the system to the initial state `s0`. This system is modeled by the following co-logic programming code.

This example illustrates the necessity of both inductive and coinductive predicates, when encoding state transitions at the level of predicate calls. The state transition system in figure 5.9 uses two different kinds of loops: an outermost infinite loop and an inner finite loop. When encoding these transitions at the level of predicate calls, infinite loops are represented by coinductive predicates (in this case the coinductive predicate `state/2`) and finite loops are represented by inductive predicates (such as this examples inductive `state/1` predicate). The fact that the finite loop is strictly nested inside the infinite loop results in a stratified program. Hence the inner loop's semantics is given by a least fixed-point, while the outer loop's semantics is given by a greatest fixed-point.

One example property that can be automatically verified is that the system never stops working. This is a desirable property for life critical embedded systems. The property can be reduced to checking for the fairness property that every enter event is eventually followed by an exit event, and the system must traverse through the work state `s2` infinitely often. Again, this is accomplished by querying for a counterexample, so `comember/2` is negated using the negation as failure operator `\+`. So along with the program in figure 5.10, the user will pose the query:

```
| ?- state(s0,X), \+ comember(s2,X).
```

The co-logic programming system will respond with the answer: `X = [s0, s3 | X]`, which is a counterexample that states that there is an infinite path not containing `s2`. This means that it is possible for the system to fall into a state where it no longer accomplishes any work. Therefore even though the system can heal itself, there is no guarantee that a failure will not occur over and over again, preventing any work from being accomplished as the system spends all of its time executing error recovery routines.

CHAPTER 6
APPLICATION: ACTION DESCRIPTION LANGUAGES

Another practical application of co-logic programming to is to field of action description languages, which is closely related to model checking in that both use automata as their core semantic foundation. The following application to action description languages was originally published in the work of Simon et al. [65], and it currently only utilizes the inductive part of co-logic programming. However, as is the case with model checking, it should be possible to generalize the results of [65] in order to reason about perpetual sequences of actions.

Action description languages are high-level languages used to systematically reason about actions and state change in dynamic environments. These languages have proved to be a useful tool for solving various aspects of planning problems, such as plan specification and verification, planning with domain specific constraints and plan diagnosis and explanation [66, 67]. Given a partial description of the state of the world, and a sequence of actions and their properties, it is possible to deduce the state induced by the action sequence. Also, given a state that results from a sequence of actions, it is required to deduce information about past states. In the next section, we describe the action description language $\mathcal{A}$ [68], with a discussion of its syntax and semantics. The language has a simple syntax that is used to specify properties of actions and an automata-theoretic semantics to reason about sequences of actions.

Real world applications of action description languages involve systems that have real-time constraints. The occurrence of an action is just as important as the time at which the action occurs. In order to be able to model such real-time systems, the

action description language $\mathcal{A}$ is extended with real-time clocks and constraints. The formal syntax and semantics of the extended language are defined, and the use of logic programming as a means to an implementation of real-time $\mathcal{A}$ is discussed.

6.1 Introduction

Non-monotonic reasoning has been an area of intense study in the recent past [67]. Within non-monotonic reasoning, considerable attention has been paid to reasoning with action and change [68]. Actions induce non-monotonic behavior since they cause a change in the state of the world. Research in this area has included the design of *action description languages (ADLs)*: high level languages that allow systematic reasoning about actions and state change in dynamic environments [67]. An example of such a language is the language $\mathcal{A}$ designed by Gelfond and Lifschitz [68]. The language $\mathcal{A}$ has been used to elegantly specify and reason about a number of classical problems such as the Yale shooting problem [68], and it has also been applied to a number of practical situations [66].

Action description languages describe the effect of actions on the truth value of logical propositions. Given a system description in $\mathcal{A}$, one can reason to find out the state(s) that results from a sequence of actions, or given a resultant state, deduce the sequence of actions that will lead us there. These actions are assumed to occur in a sequential (i.e., non-concurrent) manner, with the time intervals between two consecutive actions being arbitrary. Thus, the exact time at which the actions occur, or the elapsed time interval between two actions is of no consequence in such a language.

In practice, however, one has to reason about actions and change in a time-bound world, where actions may have to be performed within a certain time, or actions may have to be performed after a certain amount of time has elapsed. For instance, if we

consider the classical Yale shooting problem, one may wish to model the fact that if the shooting does not take place within 30 seconds then the person being shot may go out of range. A difficulty with modeling such time-dependent actions is that real-time is continuous, making its modeling and reasoning hard. Recently, however, constraint logic programming over the domain of real numbers has been shown to be suitable for modeling and reasoning with such continuous time [58].

In this chapter, we show how action description languages can be extended to *real-time action description languages*. The notion of *timed actions* - an action with time constraints attached - is introduced. Timed ADLs can be used to systematically reason about actions and state-change in dynamic environments in the presence of real-time constraints. Thus, if the action of dropping a glass causes it to be broken, then with the added ability to reason with real-time, one can reason that dropping a glass causes it to be broken, unless it is caught within half a second (i.e., before it hits the floor).

In the rest of the chapter we describe how we've extended the language $\mathcal{A}$ with real-time to obtain the language $\mathcal{A}_T$. The complete syntax and semantics of $\mathcal{A}_T$ is given, along with a description of its prototype implementation that we have recently completed. The model-theoretic semantics of action description languages is given in terms of a labeled transition system [68]. Similarly, the model-theoretic semantics of real-time systems is given in terms of timed automata [69]. We combine these two notions and give the semantics of $\mathcal{A}_T$ in terms of *timed transition systems.* Next, we render this semantics executable by denotationally mapping it to Constraint Logic Programming over reals (CLP($\mathcal{R}$)) [70]. This executable semantics serves as an implementation of $\mathcal{A}_T$.

This chapter makes a number of contributions: (i) it presents the Real-timed Action Description Language $\mathcal{A}_T$ that can be used to elegantly model actions and change in the presence of real-time constraints; (ii) it presents the semantics of this new language, along

with its implementation, based on constraint logic programming over reals; and, (iii) it paves the way for further constraint-based extensions of action description languages (such as action description languages where actions are constrained by the amount of resource, not just the presence or absence of a resource), and the incorporation of continuous real-time in planning applications.

We assume that the reader is familiar with CLP($\mathcal{R}$) as well as with timed automata. A detailed exposition can be found in [70] and [69] respectively.

6.1.1 The Action Description Language $\mathcal{A}$

The action description language $\mathcal{A}$ provides a mechanism for describing action domains. Before we delve into the details of the language, we will introduce the notion of *fluents*. Intuitively, a fluent is something whose value depends upon the state, for example, the position of the ball on a soccer field. In this chapter, we will use *propositional* fluents that take on the truth values *true* and *false* according to the state of the world. The language provides two different kinds of propositions: (i) a *value proposition*, that describes the truth value of a fluent in a particular state where the state can either be an initial state or a state resulting from a sequence of actions; (ii) an *effect proposition*, that describes the effect a given action has on a fluent.

The language $\mathcal{A}$ provides two different sets of symbols, *fluent names* and *action names*. Fluents that might be optionally preceded by a $\neg$ are called *fluent expressions*. A *value proposition* has the following syntax

$$F \textbf{ after } A_1; \ldots; A_m,$$

where F is a fluent expression and $A_1, \ldots, A_m$ $(m \geq 0)$ are action names. If $m = 0$, the above value proposition is written as

initially F.

An *effect proposition* has the syntax

$$A \textbf{ causes } F \textbf{ if } P_1, \ldots, P_n,$$

where A is an action name, and each of $F, P_1, \ldots, P_n$ $(n \geq 0)$ is a fluent expression. The effect proposition describes the effect that the action A has on the fluent F, subject to the *preconditions* $P_1, \ldots, P_n$. If $n = 0$, the above effect proposition is written as

$$A \textbf{ causes } F$$

We say that a *domain* consists of a possibly infinite set of value propositions and a finite set of effect propositions.

The Yale Shooting domain [68], consists of the fluents *Loaded* and *Alive*. The action names are *Load*, *Shoot* and *Wait*. The propositions constituting the domain are

initially $\neg$*Loaded*,
initially *Alive*,
Load **causes** *Loaded*,
Shoot **causes** $\neg$*Alive* **if** *Loaded*,
Shoot **causes** $\neg$*Loaded*.

A state consists of a set of fluents. A fluent name F holds in state σ if $F \in \sigma$ and $\neg F$ holds in σ if $F \notin \sigma$. A *transition function* is a mapping Φ from the set of pairs (A, σ) to states. A *structure* is a tuple (σ_0, Φ), where σ_0 is called the *initial state* and Φ is a transition function.

A structure (σ_0, Φ) is a *model* of a domain D if every value proposition in D is true in the structure, and for every action name A, every fluent name F and every state σ, the following hold [68]:

1. if D includes an effect proposition describing the effect of A on F, whose preconditions are valid in σ, then $F \in \Phi(A, \sigma)$.

2. if D includes an effect proposition describing the effect of A on $\neg F$ whose preconditions are valid in σ, then $F \notin \Phi(A, \sigma)$.

3. if D does not include such effect propositions, then $F \in \Phi(A, \sigma)$ if and only if $F \in \sigma$.

6.1.2 Shortcomings of $\mathcal{A}$

The language $\mathcal{A}$ allows one to reason about properties of temporal sequences of actions. In other words, time is dealt with in a qualitative manner. On the other hand, for real-time domains, it is essential to reason about time in a quantitative manner, i.e., in addition to reasoning about sequences of actions, it is also essential to reason about the deadlines that these actions have to meet. There are many situations where this capability is needed. For example, if we consider the Yale shooting problem, we may want to reason that if a loaded gun is shot, then $\neg Alive$ will become true only if the shot is fired within 30 seconds of loading the gun (otherwise the person will get away, or the ammunition will not work). Similarly, we may want to reason that the drop action will cause a breakable object to shatter, unless it is caught within 0.5 seconds, thus preventing it from hitting the ground and breaking.

Action description languages can be used for specifying controllers and developing plans for machines, plants, and robots [66]. In these real-life situations, most actions

will have severe time constraints attached. One can argue that an action description language, augmented with the capability to reason with time, will have significantly more applications; for example, in safety-critical systems. We next propose an extension to the action description language $\mathcal{A}$, which provides the machinery to specify and reason about real-time actions.

6.2 The Timed Action Description Language $\mathcal{A}_T$

We would like to be able to apply action description languages such as language $\mathcal{A}$ to real-time systems. Extending action description language $\mathcal{A}$ with real-time involves augmenting actions with clock constraints describing when the action occurs, and effect propositions must be augmented with preconditions on clocks and the ability to mutate clocks. These extensions give rise to a language we call $\mathcal{A}_T$, which is a conservative extension of language $\mathcal{A}$ in the sense that language $\mathcal{A}$ is a syntactic and semantic subset of $\mathcal{A}_T$. The following subsections cover $\mathcal{A}_T$ syntax and semantics.

6.2.1 Syntax

A real-time action α is defined as the pairing of an action name with a list of its clock constraints. In $\mathcal{A}_T$, this is written as

$$A \textbf{ at } T_1, \ldots, T_n$$

where $T_1 \ldots, T_n$ $(n \geq 0)$ are clock constraints of the form $C \leq E$, $C \geq E$, $C < E$, and $C > E$, where C is a clock name and E is a clock name or a clock name plus or minus a real valued constant, and when $n = 0$ the **at** clause can be dropped.

Now that we can explicitly state when an action occurs, value propositions are extended in a straightforward manner, given fluent expressions $F_1, \ldots, F_m$ $(m > 0)$ and

real-time actions $\alpha_1, \ldots, \alpha_n$ $(n \geq 0)$ then a real-time value proposition is of the form:

$$F_1, \ldots, F_m \textbf{ after } \alpha_1; \ldots; \alpha_n$$

Note how inconsistent descriptions can arise from a real-time value proposition including a sequence of actions occurring at inconsistent times. However, even a language as simple as language $\mathcal{A}$ allowed for inconsistent descriptions, so clock constraints are simply another source of inconsistency. Furthermore, the typical abbreviation when the sequence of actions is empty, i.e., $n = 0$ is still written

$$\textbf{initially } F_1, \ldots, F_m$$

These degenerate forms of real-time value propositions simply serve as a means to describe the start state of a real-time system by asserting which fluents are true or false in the start state. Hence these degenerate real-time value propositions serve the exact same purpose as in language $\mathcal{A}$. As will be discussed in section 6.2.3, all clocks are assumed to be reset when initially entering the start state of a real-time system.

The most significant extension occurs with the effect proposition. Real-time effect propositions, also sometimes referred to as action rules, must be able to describe the fluent preconditions as well as the clock preconditions for the rule to apply. Moreover, in addition to describing how fluents are mutated, real-time effect propositions must also be able to describe how clocks are changed, by reseting some subset of them. So real-time effect propositions are of the form

$$A \textbf{ causes } F_1, \ldots, F_m \textbf{ resets } C_1, \ldots C_n \textbf{ when } T_1, \ldots, T_k \textbf{ if } P_1, \ldots, P_i$$

for action name A, fluent expressions $F_1 \ldots, F_m$, $P_1, \ldots, P_i$ $(m, i \geq 0)$, clock names $C_1, \ldots, C_n$ $(n \geq 0)$, and clock constraints $T_1, \ldots, T_k$ $(k \geq 0)$, where $m+n+k+i > 0$. As

usual, when m, n, k, or i is zero the keywords **causes**, **resets**, **when**, or **if** respectively, can be dropped. The **resets** clause denotes the clocks that are to be reset assuming the fluent preconditions and **when** clause are satisfied. Clocks that are not reset continue to advance.

One last extension is needed [68]: A special action name **wait** denotes the action of waiting for time to elapse. Therefore it acts as a sort of wild-card that matches all other action names. This is demonstrated in the following examples.

6.2.2 Examples

The Real-time Falling Object domain, a modification of an example from [68] with the notion that a dropped object can be caught before it hits the ground assuming the object takes 1 second to hit the ground.

> $Drop$ **causes** $\neg Holding$, $Falling$ **resets** $Clock$ **if** $Holding$, $\neg Falling$
> $Catch$ **causes** $Holding$, $\neg Falling$ **when** $Clock \leq 1$ **if** $\neg Holding$, $Falling$
> **wait causes** $Broken$, $\neg Falling$ **when** $Clock > 1$ **if** $\neg Holding$, $Falling$

Firstly, note that the assumption that units are in seconds is merely a convention used in this example. As far as the language $\mathcal{A}_T$ is concerned, all clocks are simply real valued variables. Furthermore, as is the case with language $\mathcal{A}$, the language $\mathcal{A}_T$ possibly describes many possible worlds. In one of these worlds
initially $Holding, \neg Falling, \neg Broken$ is true, and therefore $Broken$ **after** $Drop$; **wait at** $Clock = 2$ also holds as the object is dropped and then allowed to fall to the ground. Similarly, in that same world, if one takes too long to catch the object, the object still shatters on the ground. Hence in the aforementioned world $Broken$ **after** $Drop$; $Catch$ **at** $Clock = 2$ is also true. However, if the object is dropped and then is successfully caught, say half a second after dropping and therefore before it hits the ground, then as

expected, the object is not broken by the sequence of events, i.e., $\neg Broken$ **after** $Drop$; $Catch$ **at** $Clock = 0.5$ is true.

Other possible worlds include the object starting out already in a falling state, while another world could even have the object already broken. The more information given in a description, the fewer possible worlds exist that satisfy the description. For example, assume that in addition to the original Real-time Falling Object domain description, it is also given that $Broken$ **after** $Drop$; $Catch$ **at** $Clock = 0.5$ is true, then it can safely be deduced that the object was broken to begin with, i.e., according to this new description it is true that **initially** $Broken$.

Similarly, the real-time Soccer Playing domain is a modification of a domain described in [71]. It has the following domain description:

$ShotTaken$ **causes** $\neg HasBall$, $\neg ClearShot$, $Goal$ **when** $Clock \leq 0.5$
if $HasBall$, $ClearShot$, $\neg Goal$
$PassBall$ **causes** $ClearShot$ **resets Clock when** $Clock \leq 1$
if $\neg ClearShot$
wait causes $\neg HasBall$, $\neg ClearShot$ **when** $Clock > 0.5$
if $HasBall$, $ClearShot$
wait causes $\neg HasBall$, **when** $Clock > 1$ **if** $HasBall$

In the real-time Soccer Playing domain, a player has the ball and has a clear shot at the goal, then it is assumed that a goal is scored if the player can take a shot within 0.5 time units. If the player does not take the shot within this time, the ball gets stripped by an opponent. Also, a player who has possession of the ball, but no clear shot at the goal can pass the ball to a team-mate who has a clear shot. The pass has to be completed within 1 time unit. Failure to do so results in the ball getting intercepted by an opponent.

In one possible world, **initially** $HasBall$, $ClearShot$, $\neg Goal$ holds and therefore $Goal$ **after** $ShotTaken$ **at** $Clock = 0.2$ is also true. However, in an alternative world, if the player does not have a clear shot, and the ball is passed to a teammate who has a clear

shot, then if the teammate does not take the shot within 0.5 time units, possession of the ball is lost. Therefore the statement $\neg HasBall$ **after** $PassBall$; $wait$ **at** $Clock = 1$ is true.

6.2.3 Semantics of $\mathcal{A}_T$

Now that the language $\mathcal{A}_T$ has been informally introduced, we can more formally specify its semantics. This should not only aid in the understanding of the language, but should also serve as a measure of the correctness of its implementations. As is the case for action language $\mathcal{A}$, the semantics of $\mathcal{A}_T$ is given in terms of a transition system. However, $\mathcal{A}_T$'s transition system is timed and therefore is technically a timed automaton with a finite region graph [69].

The semantics of $\mathcal{A}_T$ is an extension of the semantics for language $\mathcal{A}$. A state σ is a pair (Φ, Θ) where Φ is a subset of fluents and Θ is a function assigning to each clock name a non-negative real value. Let F be a fluent, then F holds in Φ if $F \in \Phi$, and $\neg F$ holds in Φ if $F \notin \Phi$. This truth valuation can be extended to sets of fluent expressions S as follows. S holds in Φ if every $F \in S$ holds in Φ. A clock valuation Θ satisfies a set of time constraints Ω (see section 6.2.1), written $S(\Theta, \Omega)$ if and only if replacing every clock name C in Ω with $\Theta(C)$ results in a consistent set. This can be extended to states in a straightforward manner, $S((\Phi, \Theta), \Omega) \equiv S(\Theta, \Omega)$.

A real-time action α is simply a pairing (A, Ω) of an action name A with a set of time constraints Ω, which denotes the time constraints on the occurrence of a specific instantiation of the action named A. In the definition of a model we will also need to enforce the notion that the clocks monotonically increase during a state transition unless they are explicitly reset, and so we say that one clock valuation Θ is less than another valuation Θ' except for the reset clocks Π, written $\Theta \leq_\Pi \Theta'$ whenever $\forall C.\Theta(C) \leq_\Pi \Theta'(C)$.

This can be extended to states such that $(\Phi, \Theta) \leq_\Pi (\Phi, \Theta')$ if and only if $\Theta \leq_\Pi \Theta'$.

Let $\longrightarrow$ be a ternary relation between egress states, actions, and ingress states such that $\sigma \xrightarrow{\alpha} \sigma'$ if and only if $\sigma \leq_\emptyset \sigma''$, $\alpha = (A, \Omega)$, and $S(\sigma'', \Omega)$, then $\longrightarrow$ is called a transition relation. Informally, $\sigma \xrightarrow{\alpha} \sigma'$ means that in state σ executing action α causes the current state to mutate into σ'. Given a start state $\sigma_0 = (\Phi_0, \Theta_0)$ for some set of fluents Φ_0 and clock valuation function Θ_0 such that $\forall C.\Theta_0(C) = 0$, i.e., all clocks are initially reset, a transition relation $\longrightarrow$ determines a system $M = (\sigma_0, \longrightarrow)$. Let $M^{\alpha_1;\ldots;\alpha_n}$ denote the possible set of states that a system could be in after executing the sequence of actions $\alpha_1; \ldots; \alpha_n$ in system M. The set of states $s = M^{\alpha_1;\ldots;\alpha_n}$ where $M = (\sigma_0, \longrightarrow)$ is recursively defined as

$$\begin{array}{ll} \{\sigma_0\} & \text{if } n = 0 \\ \{\sigma' \mid \sigma \in M^{\alpha_1;\ldots;\alpha_{n-1}} \wedge \sigma \xrightarrow{\alpha_n} \sigma'\} & \text{otherwise} \end{array}$$

Let $s = M^{\alpha_1;\ldots;\alpha_n}$, then if $M^{\alpha_1;\ldots;\alpha_n}$ is empty then the sequence of actions is said to be inconsistent. Otherwise if s is nonempty, then real-time value proposition

$$F_1, \ldots, F_m \textbf{ after } \alpha_1; \ldots; \alpha_n$$

is true (false) in a system M, if for all $(\Phi, \Theta) \in s$, $\{F_1, \ldots, F_m\}$ holds (does not hold) in Φ. Otherwise the truth value of such a proposition is unknown, written $\perp$, as in some possible states the system fluents hold and in others they do not hold. We write $V_M(P)$ to denote this truth valuation of real-time value propositions P in system M. The truth valuation can be extended to sets of systems Γ, also known as "possible worlds", in the following manner. Firstly, let Γ be a set of systems, then $\Gamma^{\alpha_1;\ldots;\alpha_n} = \{M^{\alpha_1;\ldots;\alpha_n} \mid M \in \Gamma\}$. Given a set of systems Γ, then a real-time value proposition $P \equiv F_1, \ldots, F_m$ **after** $\alpha_1; \ldots; \alpha_n$ is assigned a truth value $V_\Gamma(P)$ as follows

$$\begin{array}{ll} \mathit{inconsistent} & \text{, if } \Gamma = \emptyset \text{ or } \emptyset \in \Gamma^{\alpha_1;\ldots;\alpha_n} \\ \mathit{true} & \text{, otherwise if } \forall M \in \Gamma.V_M(P) = \mathit{true} \\ \mathit{false} & \text{, otherwise if } \forall M \in \Gamma.V_M(P) = \mathit{false} \\ \perp & \text{, otherwise.} \end{array}$$

Again, inconsistency arises when there are no possible worlds corresponding to the proposition and $\perp$ arises when the proposition holds in some worlds but does not hold in other worlds, as is the case in [68].

Before we can define the models of a domain description, we need the following additional nomenclature. Let $reset(\Phi, \Theta, A)$ be the set of all clocks reset by effects propositions in D with preconditions satisfied in state σ. Furthermore, we say that an action A in state σ causes fluent expression F whenever there exists an effects proposition P in D with preconditions satisfied in state σ such that F occurs in the **causes** clause of P.

Now we can define the models of a real-time domain description D. A system M is said to be a model for D when every real-time value proposition P in D is true in M, i.e., $V_M(P) = true$. Furthermore, the transitions in a model must also satisfy the constraints imposed by the domain description's effect propositions. Hence $(\Phi_1, \Theta_1) \xrightarrow{(A,\Omega)} (\Phi_2, \Theta_2)$ in M if and only if there exists a Θ' such that $S(\Theta', \Omega)$, $\Theta_1 \leq_\emptyset \Theta' \leq_{reset(\Phi_1,\Theta',A)} \Theta_2$, and according to the domain description D one of the following holds:

1. action A in state (Φ_1, Θ') causes F and $F \in \Phi_2$
2. action A in state (Φ_1, Θ') causes $\neg F$ and $F \notin \Phi_2$
3. action A in state (Φ_1, Θ') does not cause F or $\neg F$, and $F \in \Phi_2$ if and only if $F \in \Phi_1$

Now we can define entailment. Let Γ be the set of all models of D, then a real-time domain description D entails a real-time value proposition P, if $V_\Gamma(P) = true$, D does not entail P, if $V_\Gamma(P) = false$, and it is unknown if D entails P, if $V_\Gamma(P) = \perp$.

Discussion The semantics do not prevent different clocks from advancing at different rates, as is the case in the real world. However, it is up to the specific domain description whether or not clocks are further constrained to be synchronized. These semantics are general enough to be applied to other hybrid planning domains, not necessarily involving time, e.g., continuously consumed resources such as battery power or fuel, where it is even more important to be able to model various resources that are consumed at different rates.

6.3 Implementation

The language $\mathcal{A}_T$ can easily be implemented using co-logic programming extended with CLP($\mathcal{R}$). The implementation has been done using the SICStus Prolog system [22]. A top-level driver is used to parse an input file and then provide an interactive prompt for the user to submit queries against the description in the form of real-time value propositions. The design pattern used is to directly model both the syntax using a Definite Clause Grammar (DCG) and the denotational semantics, using syntax-directed valuation functions written as Horn clauses with real constraints that map $\mathcal{A}_T$ description parse trees to their denotations. As the predicates that implement these valuation functions are more or less a straightforward encoding of the formal semantics of $\mathcal{A}_T$ into CLP($\mathcal{R}$), proofs of the soundness and completeness of the implementation are also straightforward, yet tedious. Therefore due to lack of space such proofs are omitted.

If `Fs` is a list of fluent expressions and `As` is a sequence of real-time actions, then an evaluation of a query of the form

$$Fs \textbf{ after } As$$

is implemented by the following predicate

```
after( PT, Fs, As, V ) :-
   PT = parseTree( VPs, EPs, GCs, GFs ),
   setof( W, execute( VPs, EPs, GCs, GFs, As, W ), Ws ),
   val( Fs, Ws, V ).
```

where `PT` is the parse tree of the $\mathcal{A}_T$ domain description that the query is against and `V` is the response to the query, with values: *yes* when the query is entailed by the description, *no* when the query is not entailed, *unknown* when the description describes at least one world, i.e., model, in which the query is *true* and another in which it is *false*. The implementation is also capable of recognizing when a description or query is inconsistent and can also respond to such queries as being *inconsistent*. The higher-order *setof* is used to get a set of possible worlds `Ws` corresponding to the domain description and queried action sequence. The value `V` is calculated from these possible worlds using the `val` predicate, which mirrors the denotational semantics valuation function $V_\Gamma(P)$ defined in section 6.2.3. However, in the implementation, the set of worlds is actually a set of possible residual states, i.e., the possible resultant states that could arise according to the domain description P and action sequence As.

A residual state is defined by the predicate `execute` which takes the query's sequence of actions `As` and generates a possible residual state `W` according to the pertinent elements of the domain description: the list of value propositions `VPs`, the list of effects propositions `EPs`, the list of global clock names `GCs`, and the list of global fluent names `GFs`.

```
execute( VPs, EPs, GCs, GFs, As, W ) :-
        applyAfterConstraints( VPs, EPs, GCs, GFs, SS ),
        transitionClosure( EPs, GCs, GFs, SS, As, W ),
        validState( GFs, W ).
```

The predicate `execute` first determines the constraints on a possible start state `SS` from the domain description's value propositions and effects propositions, using the `applyAfterConstraints` predicate. Then the
`transitionClosure` predicate is used to determine the constraints on the state reached via a path determined by the sequence of actions $\widehat{A}$s and the domain's effect propositions. Finally, the term representing the residual state is grounded using the `validState` predicate.

The predicate `transitionClosure` implements a transitive closure of the `transition` predicate, which implements the transition relation $\longrightarrow$ defined in section 6.2.3.

```
transition( EPs, GCs, GFs, S1, C1, A, TCs, S2, C2 ) :-
     satisfiesAllTimeConstraints( GCs, TCs, C1 ),
     applyEffectsRules(GCs,GFs,EPs,S1,C1,A,S2,C2,FFs,RCs),
     inertia( GFs, S1, S2, FFs ),
     increasingTime( GCs, C1, C2, RCs ).
```

The egress state of the transition is represented by two lists `S1` and `C1` of fluent values (*true* and *false*) and clock values respectively, while the ingress state is similarly represented by `S2` and `C2`. The list of fluent values represents the truth value table for the state, which is more efficient than a set representation due to the multiple physical representations of a set, which unnecessarily complicates search. The clock values are similarly implemented as a value table using CLP($\mathcal{R}$) variables, which allows for a declarative implementation, yet efficient constraint solving. The predicate is implemented by first checking that all time constraints on the time of occurrence of the action $\widehat{A}$ can be satisfied. Then the set of effects propositions which can apply in such a situation are used to determine

some of the fluent constraints (i.e., forced fluents `FFs`) on the ingress state in the main goal `applyEffectsRules`. The remaining fluent constraints are determined by the set difference of global fluent names `GFs` and forced fluents `FFs` in the `inertia` predicate. Finally, the `increasingTime` predicate uses CLP($\mathcal{R}$) constraints to force the clocks in the egress state `C1` to be less than or equal to the respective clocks in the ingress state `C2`, with exceptions for the clocks that are listed in `RCs` as being reset.

The main goal `applyEffectsRules` enforces that every effects proposition either applies or does not apply by traversing through the list of effects propositions. Because of the issues involved in using negation alongside CLP($\mathcal{R}$), negation is implicitly used by defining a predicate and its dual as follows

```
effectRuleApplies( GCs, GFs, EP, S1, C1, A1, S2 ) :-
     EP = causes( A2, _, TCs, EFs, CFs ),
     equalNames( A1, A2 ),
     satisfiesAllTimeConstraints( GCs, TCs, C1 ),
     satisfiesAllFluents( GFs, S1, CFs ),
     satisfiesAllFluents( GFs, S2, EFs ).


effectRuleDoesNotApply( _, _, EP, _, _, A1 ) :-
     EP = causes( A2, _, _, _, _ ),
     not( equalNames( A1, A2 ) ).
effectRuleDoesNotApply( GCs, _, EP, _, C1, A1 ) :-
     EP = causes( A2, _, TCs, _, _ ),
     equalNames( A1, A2 ),
     notSatisfiesAllTimeConstraints( GCa, TCs, C1 ).
effectRuleDoesNotApply( _, GFs, EPs, S1, _, A1 ) :-
```

```
        EP = causes( A2, _, _, _, CFs ),
        equalNames( A1, A2 ),
        notSatisfiesAllFluents( GFs, S1, CFs ).
```

Hence while `effectRuleApplies` is true when the given effects proposition `EP`'s preconditions are satisfied by egress state `S1` and the action named `A1`, the dual predicate `effectRuleDoesNotApply` is true when some precondition is not satisfied. Also note that the explicit use of negation of `equalNames` is not necessary, but since `A1` and `A2` are always ground and the predicate's definition does not make use of CLP($\mathcal{R}$), this limited use of negation by failure simplifies the implementation. Effects propositions are represented by terms of the form

```
causes( A, RCs, TCs, EFs, CFs )
```

where `A` is the action name, `RCs` is the list of reset clocks, `TCs` is the list of time constraint preconditions, `EFs` is the list of effected fluent expressions, and `CFs` is the list of fluent preconditions.

The implementation of the condition satisfaction predicates and their duals are relatively straightforward, where the predicates involving real-time are implemented using CLP($\mathcal{R}$) constraints and the other predicates are implemented as pure Horn clauses, with the exception of a limited use of negation as failure on ground goals.

```
satisfiesAllFluents( [], [], _ ).
satisfiesAllFluents( [ N | Ns ], [ _ | Fs ], CFs ) :-
      not( member( N, CFs ) ),
      not( member( -N, CFs ) ),
      satisfiesAllFluents( Ns, Fs, CFs ).
```

```
satisfiesAllFluents( [ N | Ns ], [ true | Fs ], CFs ) :-
      member( N, CFs ),
      satisfiesAllFluents( Ns, Fs, CFs ).
satisfiesAllFluents( [ N | Ns ], [ false | Fs ], CFs ) :-
      member( -N, CFs ),
      satisfiesAllFluents( Ns, Fs, CFs ).
```

Again, while this use of negation as failure could be eliminated via explicit definition of a dual predicate, doing so is unnecessary as the negated predicate is ground and does not involve the use of CLP($\mathcal{R}$) or negation in its definition. `satisfiesAllFluents` simultaneously recurses through a list of global fluent names `Ns` and a given state's fluent value table `Fs`, verifying that the fluents are either not mentioned in the preconditions `CFs` or that their occurrence in `CFs` is satisfiable.

The definition of `satisfiesAllTimeConstraints` and its dual is more complicated, as it involves the manifestation of CLP($\mathcal{R}$) constraints, which are then applied to the clocks in question. The predicate `satisfiesAllTimeConstraints` recurses through the list of time constraints on clocks, and asserts them. Their satisfiability is determined by the CLP($\mathcal{R}$) engine.

```
satisfiesAllTimeConstraints( _, [], _  ).
satisfiesAllTimeConstraints( GCs, [ T | Ts ], Cs ) :-
     satisfiesTimeConstraint( GCs, T, Cs ),
     satisfiesAllTimeConstraints( GCs, Ts, Cs ).
```

In other words, the top-level implementation traverses through the list of time constraints `Ts`, so that each time constraint can be individually applied to the given set of clocks `Cs`.

Each individual time constraint is represented as a term and only manifested into actual CLP($\mathcal{R}$) constraints when needed, so that subsequent applications of the constraints do not alter the original definition of effects propositions. Note that w.r.t. checking for satisfiability, the exact time at which a particular action happened is unimportant, what matters is that the accumulated constraints are consistent.

6.4 Related Work

Several frameworks have been proposed to reason about the real-time aspects of actions. Most of them are extensions of the Situation Calculus or the Event Calculus, with features like occurrences and narratives, and some representation of real-time. Though these techniques provide a powerful formal mechanism for reasoning about real-time actions, there is a dearth of tools implementing them. This is because these techniques are usually axiomatized in terms of first-order logic and therefore do not allow for a tractable implementation.

Logic programming has been extensively studied in the context of implementation of the Event Calculus and its extensions. It has also been demonstrated that logic programming is a viable means for implementing action description languages, which are fragments of the Situation Calculus. In this chapter, we show how CLP($\mathcal{R}$) can be used as an elegant framework for implementing reasoning tools based on the Situation and Event Calculi.

The works of Reiter and Pinto [72, 73] provide a method for reasoning about concurrent, real-time actions in the Situation Calculus, using a solution to the frame problem. Miller et al. [74] describes methods to reason about narratives with real-time in the Situation Calculus. Pinto [75] generalizes the approach in his earlier work [73] to the Situation Calculus with narratives and occurrences. All these formalisms are based on axiomatic

reasoning using first-order logic. Implementing a reasonable model of continuous real-time, which is essential to develop a practical reasoning tool becomes difficult in these frameworks due to decidability issues. As demonstrated in this chapter, CLP($\mathcal{R}$) can be used to realize a simple and elegant model of real-time, which allows us to develop a working implementation of a reasoning tool for real-time actions.

Other techniques for reasoning with real-time include [76], which models time as being discrete, whereas the method described in this chapter provides a more general continuous model of time. [77] describes a method for reasoning about time in a temporal rather than a numerical manner. [78] presents real-time extensions of the Event Calculus, but does not provide an implementable model of real-time. Though the technique described in this chapter has been developed in the context of action description languages, it can also be extended to the Situation Calculus, the Event Calculus and their various extensions.

Action description languages have traditionally been used for reasoning about the effect of actions and change of state in various domains in [79, 67, 80, 81]. However these languages do not provide the ability to reason about actions in real-time domains.

CHAPTER 7
OTHER APPLICATIONS

The previous chapters discussed two significant applications of co-logic programming to systems verification and planning. Since co-logic programming is Church-Turing-complete, its applications are as unlimited as any other general purpose high-level programming language. This chapter briefly mentions a few of the other applications of co-logic programming. As previously mentioned, co-logic programming generalizes traditional logic programming with rational trees as discussed in section 7.1, as well as lazy evaluation of predicates as discussed in section 7.2, and even concurrent logic programming as discussed in section 7.3. Because co-logic programming allows the programmer to reason at the high level of formal logic, it can also be used as an inference engine for automating reasoning about web services, as discussed in section 7.4.

7.1 Infinite Terms and Properties

As previously stated, co-logic programming subsumes traditional logic programming with rational trees of Jaffar et al [6] and Colmerauer [37]. However, because traditional logic programming with rational trees has semantics ascribed by the minimal co-Herbrand model, applying predicates to infinite trees is rather limited. Doing so typically results in nontermination, as true atoms cannot have infinite idealized proofs. Co-logic programming removes this limitation by ascribing the semantics in terms of the *maximal* co-Herbrand model and it provides an operational semantics that provides finite derivations for atoms with infinite idealized proofs. Hence true atoms can have finite or infinite idealized proofs. This is demonstrated by the traditional definition of `append`, which,

when executed with co-logic programming's semantics, allows for calling the predicate with infinite arguments. This is illustrated below. As an aside, note that irrational lists also make it possible to directly represent an infinite precision irrational real number as an infinite list of natural numbers, i.e., representing the number as a decimal expansion.

```
append( [], X, X ).
append( [H|T], Y, [H|Z] ) :- append( T, Y, Z ).
```

Not only can the above definition append two finite input lists, as well as split a finite list into two lists in the reverse direction, it can also append infinite lists under coinductive execution. It can even split an infinite list into two lists that when appended, equal the original infinite list. For example:

```
| ?- Y = [4, 5, 6, | Y], append([1, 2, 3], Y, Z).
    Answer: Z = [1, 2, 3 | Y], Y = [4, 5, 6, | Y]
```

If we also allow the possibility of expanding the variant call using its definition (and apply the coinductive hypothesis rule in the variant that will arise subsequently), then we will enumerate more values for `Y`:

```
Y = [4, 5, 6, 4, 5, 6, | Y]
Y = [4, 5, 6, 4, 5, 6, 4, 5, 6, | Y]
...
```

More generally, the coinductive append has interesting algebraic properties. When the first argument is infinite, it doesn't matter what the value of the second argument is, as the third argument is always equal to the first. However, when the second argument

is infinite, the value of the third argument still depends on the value of the first. This is illustrated below:

```
| ?- X = [1, 2, 3, | X], Y = [3, 4 | Y], append(X, Y, Z).
    Answer: Z = [1, 2, 3 | Z]


| ?- Z = [1, 2 | Z], append(X, Y, Z).
    Answers: X = [], Y = [1, 2 | Z];
             X = [1], Y = [2 | Z];
             X = [1, 2], Y = Z
```

As noted earlier, more solutions (e.g., `X = [], Y = [1,2,1,2|Y]` for the second query) can be generated by expanding the variant (coinductive) call to `append` using its definition, and applying the coinductive hypothesis rule to the subsequent variant calls.

All of these example queries would cause a traditional logic programming system that lacks rational trees, to return "`no`", signaling that the given query is not satisfiable. The reason for this is that the only solutions that satisfy the queries involve infinite lists, which simply do not exist in traditional logic programming. It is also interesting to note that in a traditional logic programming system with rational trees, these queries would diverge into an infinite loop, and no results would be generated. The obvious intent of these queries is for them to be satisfiable, which isn't possible in either previous approach. Traditional logic programming lacks both the infinite terms and infinite proofs necessary to compute the correct answers, and logic programming with rational trees lacks the infinite proofs necessary to compute the correct answers. This demonstrates that it is

necessary to extend traditional logic programming with both infinite terms and infinite proofs, as logic programming with rational trees is only a partial solution to the problem.

7.2 Lazy Evaluation of Logic Programs

Co-logic programming also allows for lazy evaluation to be elegantly incorporated into traditional logic programming. Lazy evaluation allows for manipulation of, and reasoning about, cyclic and infinite data structures and properties. In co-logic programming, if the infinite terms involved are rational, then given the goal `p(X),q(X)` with coinductive predicates `p/1` and `q/1`, then `p(X)` can coinductively succeed and terminate, and then pass the resulting `X` to `q(X)`. If `X` is bound to an infinite irrational term during the computation, then `p` and `q` must be executed in a coroutined manner to produce answers. That is, one of the goals must be declared the producer of `X` and the other the consumer of `X`, and the consumer goal must not be allowed to bind `X`. Consider the (coinductive) lazy logic program for the sieve of Eratosthenes, taken from [4]:

```
:- coinductive sieve/2, filter/3, member/2.
primes(X) :- generate_infinite_list(I), sieve(I,L), member(X, L).
sieve([H|T], [H|R]) :- filter(H,T,F), sieve(F,R).
filter(H,[],[]).
filter(H,[K|T],[K|T1]) :- R is K mod H, R > 0,filter(H,T,T1).
filter(H,[K|T],T1) :- 0 is K mod H, filter(H,T,T1).
```

In the above program `filter/3` removes all multiples of the first element in the list, and then passes the filtered list recursively to `sieve/2`. If the predicate `generate_infinite_list(I)` binds `I` to a rational list (e.g., `X = [2, .., 20 | X]`),

then filter can be *completely* processed in each call to `sieve/2`. However, in contrast, if `I` is bound to an irrational infinite list as in:

```
:- coinductive int/2.
int(X, [X|Y]) :- X1 is X+1, int(X1, Y).
generate_infinite_list(I) :- int(2,I).
```

then in the `primes/1` predicate, the calls `generate_infinite_list/1`, `sieve/2` and `member/2` should be coroutined, and, likewise, in the `sieve/2` predicate, the call `filter/3` and the recursive call `sieve/2` must be coroutined. Since co-logic programming allows for both inductive and coinductive predicates, it can support both traditional evaluation of inductive atoms as well as lazy evaluation of coinductive atoms.

7.3 Concurrent Logic Programming and Perpetual Processes

From the discussion on lazy logic programming, one can also observe that co-logic programming can be the basis of providing elegant declarative semantics to concurrent logic programming, which involve perpetual processes. As stated in section 2.3, it is well known that atoms (and queries) with infinite *SLD* derivations are contained in the maximal model [43, 6, 44, 7]. Since the declarative semantics for coinductive predicates is defined in terms of a greatest fixed-point, they too can have infinite idealized proofs and hence infinite *SLD* derivations. While the current implementation described in section 4.3 does not allow for concurrency, an industrial strength implementation is being developed that supports co-logic programming, concurrency, and many other advanced logic programming features [9]. The important thing to note is that the declarative semantics of traditional logic programming do not allow for atoms to have infinite idealized proofs, and therefore perpetual processes do not fit nicely into the framework of traditional logic

programming. A more general framework involving alternating fixed-points is needed, and this is exactly what co-logic programming provides.

7.4 Web Services

The next milestone in the Web's evolution is making *services* ubiquitously available. A web service is a program available on a web-site that effects some action or change in the world (i.e., causes a side-effect). Examples of such side-effects include a web-base being updated because of a plane reservation made over the Internet, a device being controlled, etc.

As automation increases, these web-services will be accessed directly by the applications themselves rather than by humans. In this context, a web-service can be regarded as a programmatic interface that makes application to application communication possible. For web-services to become practical, an infrastructure needs to be supported that allows users to discover, deploy, compose, and synthesize services automatically. Such an infrastructure *must* be semantics based so that applications can reason about a service's capability to a level of detail that permits their discovery, deployment, composition and synthesis.

Several efforts are underway to build this infrastructure. These efforts include approaches based on the semantic web (such as OWL-S [82]) as well as those based on XML, such as Web Services Description Language (WSDL) [83]. Approaches such as WSDL are purely syntactic in nature, that is, they merely specify the format of the service. In this section we describe an approach that is based on semantics. The approach of Simon et al. [84, 85, 86] can be regarded as providing semantics to WSDL statements.

The work of Simon et al. presents the design of a language called the Universal Services Description Language, USDL for short, which service developers can use to specify

formal semantics of web-services. Thus, if WSDL can be regarded as a language for formally specifying the syntax of web services, USDL can be regarded as a language for formally specifying their semantics. USDL can be thought of as *formal program documentation* that will allow sophisticated conceptual modeling and searching of available web services, automated composition, and other forms of automated service integration. For example, the WSDL syntax and USDL semantics of web services can be published in a directory which applications can access to automatically discover services. That is, given a formal description of the context in which a service is needed, the service(s) that will precisely fulfill that need can be automatically determined by an inference engine based on co-logic programming. The directory can then be searched to check if this exact service is available, and if not available, then whether it can be synthesized by composing two or more services listed in this (or another) directory.

To provide formal semantics, a common denominator must be agreed upon that everybody can use as a basis of understanding the meaning of services. This common conceptual ground must also be somewhat coarse-grained so as to be tractable for use by both engineers and computers. That is, semantics of services should not be given in terms of low-level concepts such as Turing machines, or from first principals using first-order logic, since service description, discovery, and synthesis then become tasks that are practically intractable and theoretically undecidable. Additionally, the semantics should be given at a conceptual level that captures common real world concepts. Furthermore, it is too impractical to expect disparate companies to standardize on application (or domain) specific ontologies to formally define semantics of web-services, and instead a common universal ontology must be agreed upon with additional constructors. Also, application specific ontologies will be an impediment to automatic discovery of services since the application developer will have to be aware of the specific ontology that has been used to

describe the semantics of the service in order to frame the query that will search for the service. The danger is that the service may not be defined using the particular domain specific ontology that the application developer uses to frame the query, however, it may be defined using some other domain specific ontology, and so the application developer will be prevented from discovering the service even though it exists. These reasons make an ontology based on WordNet OWL [87, 88] a suitable candidate for a universal ontology of atomic concepts upon which arbitrary meets and joins can be added in order to gain tractable flexibility.

Like WSDL, USDL describes a service in terms of ports and messages [84]. The semantics of the service is given in terms of the WordNet OWL ontology [87, 88]. USDL maps ports (operations provided by the service) and messages (operation parameters) to disjunctions of conjunctions of (possibly negated) concepts in the WordNet OWL ontology. The semantics is given in terms of how a service *affects* the external world. The present design of USDL assumes that each side-effect is one of the following operations: *create, update, delete,* or *find*, but also allows for a generic side-effect when none of the others apply. An application that wishes to make use of a service automatically should be able to reason with WordNet atoms using the WordNet OWL ontology.

USDL is perhaps the first language that attempts to capture the semantics of web-services in a universal, yet computationally tractable manner. It is quite distinct from previous approaches such as WSDL [83] and OWL-S [82]. As mentioned earlier, WSDL only defines syntax of the service; USDL can be thought of as providing the missing semantic component. USDL can be thought of as a formal language for program documentation. Thus instead of documenting the function of a service as comments in English, one writes USDL statements that describe the function of that service. USDL is quite distinct from OWL-S, which is designed for a similar purpose, and as we shall

see the two are in fact complimentary. OWL-S primarily describes the states that exists before and after the service and how a service is composed of other smaller sub-services (if any). Description of atomic services is left underspecified in OWL-S. They have to be specified using domain specific ontologies; in contrast atomic services are completely specified in USDL since USDL relies on a universal ontology (OWL WordNet Ontology) [87, 88]. USDL and OWL-S are complimentary in the sense that OWL-S's strength lies in describing the structure of composite services, i.e., how various atomic services are algorithmically combined to produce a new service, while USDL is good for fully describing atomic services. Thus, OWL-S can be used for describing the structure of composite services that combine atomic services that are described using USDL.

The formal semantics of USDL maps the syntactic terms describing ports and messages to disjunctions and conjunctions of (possibly negated) OWL WordNet ontological terms. These disjunctions and conjunctions are represented by points in the lattice obtained from the WordNet ontology with regards to the OWL subsumption relation. A service is then formally defined as a function, labeled with zero or more side-effects, between points in this lattice.

The design of USDL rests on two formal languages: Web Services Description Language (WSDL) [83] and Web Ontology Language (OWL) [89]. The Web Services Description Language (WSDL) [83], is used to give a syntactic description of the name and parameters of a service. The description is syntactic in the sense that it describes the formatting of services on a syntactic level of method signatures, but is incapable of describing what concepts are involved in a service and what a service actually does, i.e. the conceptual semantics of the service. Likewise, the Web Ontology Language (OWL) [89], was developed as an extension to the Resource Description Framework (RDF) [90], both standards are designed to allow formal conceptual modeling via logical ontologies, and

these languages also allow for the markup of existing web resources with semantic information from the conceptual models. USDL employs WSDL and OWL in order to describe the syntax and semantics of web services. WSDL is used to describe message formats, types, and method prototypes, while a specialized universal OWL ontology is used to formally describe what these messages and methods mean, on a conceptual level.

Since USDL descriptions are essentially type annotations for web services, with atomic types being taken from WordNet combined using set-theoretic type constructors and effects [85], numerous tasks involving USDL descriptions require the ability to check for type subsumption [91], which involves coinductive reasoning in the presence of recursive types [92]. For this reason, co-logic programming is a suitable inference engine for automating reasoning about web services, especially when they are annotated in USDL [86].

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

This dissertation has presented a method for extending traditional logic programming with coinduction. This extension is conservative in the sense that all of the features of traditional logic programming are strictly contained in the final result of the extension: co-logic programming. This extension involves three steps that compromise the three major contributions of this dissertation.

First it is necessary to note that traditional logic programming is actually *inductive* logic programming. Every aspect of traditional logic programming is inductively defined. The terms, proofs, and the canonical models of traditional logic programs are defined via least fixed-points. Furthermore, the operational semantics for traditional logic programming implicitly relies on the fact that a least fixed-point can be calculated in an interated fashion, as is demonstrated in its soundness and completeness proofs.

Chapter 3 demonstrates that the mathematical dual of every aspect of *inductive* logic programming's declarative semantics can be used to derive coinductive logic programming. While the principle of duality allows for the first contribution of this dissertation, the syntax and semantics of coinductive logic programming, to be obtained relatively easily, in order for it to be a practical programming language, it is necessary to also create a new operational semantics capable of automating coinductive reasoning. Hence the second major contribution of this dissertation is the creation of an operational semantics for coinduction, embodied in the co-SLD operational semantics for coinductive logic programming. It is important to note that co-SLD is dual to another kind of operational semantics for traditional logic programming: $OLDT$ [56]. Both methods use a form of

tabling that is dual to the other. As demonstrated in many aspects of computer science such as linear programming and programming languages [93], duality is an important property that yields both theoretical insights and practical benefit. Hence the concept of duality can be seen as an underlying theme of this dissertation.

Finally, in order to retain the features of traditional logic programming, both *inductive* and coinductive logic programming are combined, yielding the third major contribution of this dissertation: co-logic programming, as defined in chapter 4. Co-logic programming strictly contains traditional or inductive logic programming, and yet it extends it with coinductive reasoning, corecursive computation, and infinite data structures.

Co-logic programming is an extremely high-level and Church-Turing-complete programming language, and hence it has many practical applications. As demonstrated in chapter 5, co-logic programming has practical applications as a high-level systems verification language. It is capable of directly encoding model checking based verification problems, such as the verification of safety and liveness properties. This should come as no surprise, once one notices that safety, which is traditionally checked using inductive reasoning, is dual to liveness, which can be easily checked using coinductive reasoning. So again, this dissertation demonstrates the importance of recognizing duality in computer science.

Chapter 6 shows how co-logic programming, when extended with constraints over the real numbers, is capable of reasoning at a high-level about various planning and scheduling problem domains, and the applications do not stop there. Chapter 7 discusses how co-logic programming "completes" logic programming with rational trees, as rational trees are useless, unless rational proofs are allowed for reasoning about them. This extension of rational trees goes further, as co-logic programming's declarative semantics also allows for infinite terms and proofs that are not rational, which is necessary for both

lazy logic programming and concurrent logic programming. This generality is also useful for reasoning about Web services annotated with a semantic markup language such as USDL.

This dissertation is just the beginning of a line of research involving extending logic programming with coinductive reasoning. While this dissertation discusses a simple interpreter executing co-logic programs, a next-generation logic programming system is being developed, which is capable of executing inductive predicates and coinductive predicates with many additional features such as constraints, $OLDT$-style tabling, concurrency, and non-monotonic reasoning similar to Answer Set Programming [9, 10, 4, 5].

REFERENCES

[1] Stephen Muggleton. *Inductive Logic Programming.* Academic Press, New York, 1992.

[2] Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena.* CSLI Publications, Stanford, California, 1996.

[3] Simon Peyton Jones and et al, editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.

[4] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *Proceedings of the International Conference on Logic Programming*, 2006.

[5] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *Proceedings of the International Workshop on Software Verification and Validation*, 2006.

[6] Joxan Jaffar and Peter J. Stuckey. Semantics of infinite tree logic programming. *Theoretical Computer Science*, 46(2–3):141–158, 1986.

[7] John Wylie Lloyd. *Foundations of logic programming.* Springer Verlag, New York, second extended edition, 1987.

[8] Luke Simon. Co-inductive logic programming. Technical report, University of Texas at Dallas, March 2004.

[9] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. Technical Report UTDCS-11-06, University of Texas at Dallas, 2006.

[10] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. Technical Report UTDCS-21-06, University of Texas at Dallas, 2006.

[11] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[12] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, Heidelberg, and New York, March 1981. Springer-Verlag.

[13] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, September 1991.

[14] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.

[15] Peter Aczel. An introduction to inductive definitions. In K. Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.

[16] Peter Aczel. *Non-well-founded Sets*, volume 14 of *CSLI Lecture Notes*. CSLI Publications, Stanford, CA, 1988.

[17] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[18] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[19] Davide Sangiorgi and David. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[20] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 1999.

[21] Yves Deville. *Logic Programming: Systematic Program Development*. International Logic Programming Series. Addison-Wesley, Reading, Mass., 1990.

[22] SICS. *SICStus User Manual. Version 3.10.0*. Swedish Institute of Computer Science, 2002.

[23] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.

[24] J. A. Robinson. Computational logic: The unification computation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, pages 63–72. Edinburgh University Press, Edinburgh, Scotland, 1971.

[25] Jacques Corbin and Michel Bidoit. A rehabilitation of Robinson's unification algorithm. *Information Processing*, pages 909–914, 1983.

[26] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[27] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, February 1982.

[28] Gérard Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7, Paris, France, 1976.

[29] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in honor of Alan Robinson*, pages 257–321. MIT Press, 1991.

[30] Michael S. Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[31] Dennis de Champeaux. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and System Sciences*, 32(1):79–90, February 1986.

[32] Neil Vincent Murray. Linear and almost-linear methods for the unification of first order expressions. Master's thesis, Syracuse University, 1979.

[33] Kevin Knight. Unification: a multidisciplinary survey. *ACM computing surveys, March 1989*, 21(1):93–124, 1989.

[34] Franz Baader and Jörg Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, *Deduction Methodologies*, pages 41–125. Oxford University Press, 1994.

[35] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[36] Krzysztof R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 15, pages 493–574. MIT Press, 1990.

[37] Alain Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1982.

[38] Alain Colmerauer. PROLOG II reference manual and theoretical model. Technical report, Groupe Intelligence Artificielle, Université Aix– Marseille II, October 1982.

[39] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *FGCS-84: Proceedings International Conference on Fifth Generation Computer Systems*, pages 85–99, Tokyo, 1984. ICOT.

[40] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[41] Alain Colmerauer. Specifications of Prolog IV. Draft, 1996.

[42] Maarten H. van Emden and John W. Lloyd. A logical reconstruction of Prolog II. In Sten-Åke Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 35–40, Uppsala, 1984.

[43] M. A. Nait Abdallah. On the interpretation of infinite computations in logic programming. In Jan Paredaens, editor, *Automata, Languages and Programming, 11th Colloquium*, volume 172 of *Lecture Notes in Computer Science*, pages 358–370, Antwerp, Belgium, 16–20 July 1984. Springer-Verlag.

[44] Mathieu Jaume. Logic programming and co-inductive definitions. In *CSL*, pages 343–355, 2000.

[45] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. A CLP proof method for timed automata. In *RTSS*, pages 175–186, 2004.

[46] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. A CLP method for compositional and intermittent predicate abstraction. In *VMCAI*, pages 17–32, 2006.

[47] Elio Giovannetti, Giorgio Levi, Corrado Moiso, and Catuscia Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, April 1991.

[48] Werner Hans, Rita Loogen, and Stefan Winkler. On the interaction of lazy evaluation and backtracking. *Programming Lanugage Implementation and Logic Programming*, 631:355–369, 1992.

[49] J. J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12(3):191–223, 1992.

[50] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 268–279, Portland, Oregon, January 17–21, 1994. ACM Press.

[51] Michael Hanus. The integration of functions into logic programming: From theory to practice. *The Journal of Logic Programming*, 19 & 20:583–628, May 1994.

[52] Michael Hanus, Herbert Kuchen, and Rwth Aachen. Curry: A truly functional logic language, November 18 1995.

[53] Witold Charatonik, David A. McAllester, Damian Niwinski, Andreas Podelski, and Igor Walukiewicz. The horn mu-calculus. In *LICS*, pages 58–69, 1998.

[54] Jean-Marc Talbot. On the alternation-free horn mu-calculus. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955, pages 418–435. Springer, 2000.

[55] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, pages 95–212, 1983.

[56] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In Ehud Y. Shapiro, editor, *Third International Conference on Logic Programming, Imperial College of Science and Technology, London, United Kingdom, July 14-18, 1986, Proceedings*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.

[57] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[58] Gopal Gupta and Enrico Pontelli. A constraint-based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 230–239, 1997.

[59] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *CAV*, pages 143–154, 1997.

[60] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 132–144. ACM, 2005.

[61] Moshe Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proceedings, Symposium on Logic in Computer Science*, pages 167–176, Ithaca, New York, 22–25 June 1987. The Computer Society of the IEEE.

[62] Xinxin Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points (extended abstract). In Bernhard Steffen,

editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 1998.

[63] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci*, 149(1):79–96, 2006.

[64] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[65] Luke Simon, Ajay Mallya, and Gopal Gupta. Design and implementation of A_T: A real-time action description language. In Patricia M. Hill, editor, *Proceedings of the International Workshop on Logic-based Program Synthesis and Transformation*, volume 3901 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2005.

[66] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-prolog decision support system for the space shuttle. In I. V. Ramakrishnan, editor, *PADL*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001.

[67] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[68] Michael Gelfond and Vladimir Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, 17:301–322, 1993.

[69] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999.

[70] Kim Marriott and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.

[71] U.D. Ulusar and H.L. Akin. Design and implementation of a real time planner for robots. In *Proc. TAINN 2004*, pages 263–270, 2004.

[72] Raymond Reiter. Natural actions, concurrency and continuous time in the situation calculus. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 2–13. Morgan Kaufmann, San Francisco, California, 1996.

[73] Javier Pinto and Raymond Reiter. Reasoning about time in the situation calculus. *Ann. Math. Artif. Intell*, 14(2-4):251–268, 1995.

[74] Rob Miller and Murray Shanahan. Narratives in the situation calculus. *Journal of Logic and Computation*, 4(5):513–530, October 1994.

[75] Javier Pinto. Occurrences and narratives as constraints in the branching structure of the situation calculus. *Journal of Logic and Computation*, 8(6):777–808, 1998.

[76] Erik Sandewall. *Features and Fluents (vol. 1): The Representation of Knowledge about Dynamic Systems*. Oxford University Press, 1995.

[77] Vol Nr, Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnstrom. TAL: Temporal action logics language specification and tutorial, October 05 1998.

[78] Rob Miller and Murray Shanahan. Some alternative formulations of the event calculus. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic. Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 452–490. Springer, 2002.

[79] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3, 1998. Available at http://www.ep.liu.se/rs/cis/1998/016/.

[80] Vladimir Lifschitz. Answer set planning (abstract). In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *LPNMR*, volume 1730 of *Lecture Notes in Computer Science*, pages 373–374. Springer, 1999.

[81] Hudson Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, 298(31):245–298, 1997.

[82] Owl-s: Semantic markup for web services. http://www.daml.org/services/owl-s/1.0/owl-s.html.

[83] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. World Wide Web Consortium, Note NOTE-wsdl-20010315, March 2001.

[84] Luke Simon, Ajay Mallya, Ajay Bansal, Gopal Gupta, and Thomas D. Hite. A universal service description language. In *ICWS*, pages 823–824. IEEE Computer Society, 2005.

[85] Luke Simon, Ajay Bansal, Ajay Mallya, Srividya Kona, Gopal Gupta, and Thomas D. Hite. Towards a universal service description language. In *International Conference on Next Generation Web Services Practices, 2005. NWeSP 2005.* IEEE Computer Society, 2005.

[86] Ajay Bansal, Srividya Kona, Luke Simon, Ajay Mallya, Gopal Gupta, and Thomas D. Hite. A universal service-semantics description language. In *Third IEEE European Conference on Web Services, 2005. ECOWS 2005.* IEEE Computer Society, 2005.

[87] George A. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, November 1995.

[88] Claudia Cior Ascu, Iulian Cior Ascu, and Kilian Stoffel. knOWLer - ontological support for information retrieval systems, September 25 2003.

[89] Mike Dean, Guus Schreiber, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference. Working draft, W3C, March 2003.

[90] Eric Miller. An introduction to the resource description framework. *D-Lib Magazine*, May 1998. http://www.dlib.org/dlib/may98/miller/05miller.html.

[91] Tom Hite. Service composition and ranking: A strategic overview. Research report, Metallect Incorporated, 2005.

[92] Francois Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report 3483, INRIA, September 1998.

[93] Philip Wadler. Call-by-value is dual to call-by-name. *ACM SIGPLAN Notices*, 38(9):189–201, September 2003.

VITA

Luke Evans Simon was born and raised in Dallas, Texas on January 1, 1979, as the son of Denis and Denise Simon. He received both a Bachelor of Science degree in Computer Science from the University of Texas at Dallas in 2001, as well as a Master of Science degree in Computer Science in 2003. He married Atussa Kamalpour in August of 2003.

Technical Reports

1. Luke Simon. Observable Equivalence for Interaction Nets. Technical Report UTDCS-16-04, University of Texas at Dallas, 2004.
2. Luke Simon. Co-inductive logic programming. Technical report, University of Texas at Dallas, March 2004.
3. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. Technical Report UTDCS-11-06, University of Texas at Dallas, 2006.
4. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. Technical Report UTDCS-21-06, University of Texas at Dallas, 2006.

Conference Articles

1. Luke Simon, Ajay Mallya, Ajay Bansal, Gopal Gupta, and Thomas D. Hite. A universal service description language. In *International Conference on Web Services*. IEEE Computer Society, 2005.

2. Luke Simon, Ajay Bansal, Ajay Mallya, Srividya Kona, Gopal Gupta, and Thomas D. Hite. Towards a universal service description language. In *International Conference on Next Generation Web Services Practices, 2005. NWeSP 2005.* IEEE Computer Society, 2005.

3. Ajay Bansal, Srividya Kona, Luke Simon, Ajay Mallya, Gopal Gupta, and Thomas D. Hite. A universal service-semantics description language. *Third IEEE European Conference on Web Services, 2005. ECOWS 2005.* IEEE Computer Society, 2005.

4. Luke Simon, Ajay Mallya, and Gopal Gupta. Design and implementation of A_T: A real-time action description language. In Patricia M. Hill, editor, In *Proceedings of the International Workshop on Logic-based Program Synthesis and Transformation*, volume 3901 of Lecture Notes in Computer Science. Springer, 2005.

5. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *Proceedings of the International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer, 2006.

6. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *Proceedings of the International Workshop on Software Verification and Validation*, 2006.

Permanent address: 2700 Summit View Dr.
Plano, Texas 75025
U.S.A.