A Proof-Theoretic Approach to Coinduction in Horn Clause Logic

Yue Li

Submitted for the degree of Doctor of Philosophy

Heriot-Watt University

School of Mathematical and Computer Sciences

September 9, 2019

The copyright in this thesis is owned by the author. Any quotation from the thesis or use of any of the information contained in it must acknowledge this thesis as the source of the quotation or information.

Abstract

The thesis is on coinduction in Horn clauses. Specifically, it considers *productive corecursion*, and presents a framework called *Coinductive Uniform Proof* as a principled approach to coinduction in first-order Horn clause logic. It addresses the challenges of 1) discovering sufficient conditions for logic programs to be productive, 2) providing an explanation of why unification (without occur-check) between goals in a SLD derivation can be exploited to capture productive corecursion, and 3) identifying the principle that unifies the diverse approaches to Horn clause coinduction which are scattered across the literature.

The thesis advances the state of the art by 1) providing a sufficient condition for productive corecursion which requires that a logic program does not admit perpetual term-rewriting steps nor existential variables, 2) showing that the goal-unification technique can be used to capture productive corecursion if a goal is no less general than some previous goal to which it unifies, and 3) defining a coinductively sound proof construction method for Horn clauses where a Horn clause to be proved is first asserted as an assumption and then used for its own proof.

Acknowledgments

Supervision

Dr Ekaterina Komendantskaya, Heriot-Watt University, UK

Prof. Mark V. Lawson, Heriot-Watt University, UK

Progress Examination

Dr Lilia Georgieva, Heriot-Watt University, UK

Dr Murdoch James Gabbay, Heriot-Watt University, UK

Viva Committee

Prof. John Power, University of Bath, UK

Prof. Andrew Ireland, Heriot-Watt University, UK

Prof. Albert Burger, Heriot-Watt University, UK

Thank you *Katya* for providing me with the PhD opportunity, giving me interesting research questions to solve, helping me learning about the ethics of research and giving me advices and assistance for a better work-life balance.

Thank you *Mark* for coordinating the thesis writing stage, proof-reading the thesis draft and offering your valuable advices.

Thank you *Henning* for collaborating with me and Katya on publishing the work on Coinductive Uniform Proof and facing together with us the various sorts of difficulties occurred in that period.

Thank you *Robin* for discussing with me over these years about the multiple dimensions of the challenges of my PhD life.

Thank you *Diana, Tessa, Alasdair (staff), Alasdair (student), Xingkun, Laura, Dugald, Heiko, Desmond, Beatrice, Oliver, Lyonell, Dominic, Riccardo, Chris, Alex, Bryan, Christian, Bernd, Robert, Andrew, Martin, Ruth, Mike (math), Mike (CS), Santiago, Jenny, Hans, Manuel, Ron, Rob, Hamish, Nicholas, Joe, Reuben, Amani, Miruna, Kirsty, Kaite, Ben, Filip, Imran, Ahmad, Iain, Christine, Derek, Clair, Laura, Rodi, Adrian, June, Steve, Michael, Guandong, Xiaojiao, Matthew, Murat, Piyush and Hussam* for your good service or friendship.

Contents

1 Introduction **1**

1.1 Research Questions . . . 2

1.1.1 Explaining Question (1a) . . . 2

1.1.2 Explaining Questions (1b) and (1c) . . . 3

1.1.3 Explaining Question (2a) . . . 5

1.2 Contributions . . . 6

1.3 Outline . . . 7

2 Basic Notions **9**

2.1 Well-formed Terms . . . 10

2.1.1 The Tree Language . . . 10

2.1.2 Elementary Symbols for Terms . . . 11

2.1.3 Well-Formed Terms . . . 12

2.1.4 The Term Metric . . . 14

2.1.5 First-order Terms, Atoms . . . 15

2.2 Unification . . . 16

2.3 Model Theory . . . 18

2.4 A Type System . . . 20

2.5 Higher-order Variables . . . 21

2.5.1 Variable Contexts . . . 21

2.6 Guarded Lambda Terms . . . 22

2.6.1 Typed Symbols . . . 22

2.6.2 Lambda Terms . . . 23

2.6.3 Guarded Fixed-points . . . 25

2.6.4 Guarded Lambda Terms . . . 26

2.6.5 Order of Guarded Lambda Terms . . . 27

2.6.6 Mapping From Λ^G_Σ to $1st\mathbf{Term}^\omega(\Sigma)$ 27
2.7 Formulae 30
2.7.1 Order of Formula 31
2.7.2 The Hereditary Harrop Language 32
2.7.3 M-formulae and H-formulae 32

3 Productive LP 34
3.1 Structural Resolution 35
3.2 Productivity 37
3.3 Loop Detection 43

4 CUP System 50
4.1 Coinductive Uniform Proof 51
4.2 Basic Observations about CUP 54
4.2.1 The Common Opening Pattern 54
4.2.2 Trivial Coinductive Uniform Proofs 57

5 CUP Model-theoretic Soundness 59
5.1 Infinite-term Horn Clause 60
5.1.1 Horn^ω Immediate Consequence Operator 61
5.1.2 Horn^ω Greatest Complete Herbrand Model 62
5.2 Principal Back-chaining Sequent 63
5.3 Delta-substitution 66
5.4 Eigen-variable Substitution System 71
5.5 Post-fixed-point Construction 74
5.6 Theorems on Model-theoretic Soundness 78

6 CUP Proof-theoretic Soundness 80
6.1 Coinductive Intuitionistic Logic 80
6.2 CUP Proof-theoretic Soundness 83

7 Related Work 90
7.1 μMALL and CUP 90
7.1.1 Formulating predicates as fixed-points 91
7.1.2 Using the fixed-point theorem for inference rules 94
7.1.3 Relating μMALL and CUP 96

7.1.4 Conclusion 99
7.2 Abella and CUP 99
7.2.1 Line of work leading to the Abella prover 99
7.2.2 Comparing coinduction in Abella with CUP 101
7.2.3 Conclusion 104
7.3 Discussion: post-fixed-points vs. coinductive goals 105
7.4 Cyclic Proof, CUP and Abella 105
7.4.1 Introducing cyclic proof 106
7.4.2 Relating cyclic proof to Abella's induction 107
7.4.3 Conclusion 108

8 Conclusion and Future work **110**
8.1 Summary 110
8.2 Future work 111

A Tarski's Fixed-Point Theorem **112**

References **115**

Index of Definitions **121**

Index of Notation **124**

Chapter 1

Introduction

This thesis presents a body of *theoretical* work that potentially benefits the academic communities that work on design of functional programming languages, development of interactive theorem provers, and computer-aided formal verification of safety-critical software. The following two paragraphs explain in general terms the two topics discussed by the thesis.

Perpetual Computation in Logic Programming A sufficient condition has been formulated and proved, under which non-terminating Horn clause logic programs are guaranteed to produce infinite terms. Then follows the explanation of why the existing coinductive resolution rule [1] for Horn clauses can sometimes finitely capture an non-terminating computation as well as the infinite term that results from it.

Coinduction in Horn Clause Logic The thesis presents the novel framework *Coinductive Uniform Proof* that can finitely capture both regular and irregular proof trees in Horn clause logic, with both regular and irregular terms taken into account. Coinduction in Horn clause logic underlies the handling of coinductive data types in functional programming languages such as Haskell [2]. In turn, functional languages are used as the specification languages by interactive theorem provers (such as Coq and Isabelle/HOL) where we build and verify safety-critical computer software such as real-time operating systems [3–6]. Theorem provers need to provide tactics for reasoning with coinductively defined relations and data types [7, 8]. The insights embodied in the thesis would help with the designing of coinductive data types in functional languages, and may provide a new tactic for theorem provers to reason

coinductively[1].

Section 1.1 outlines the exact research questions and explains what they mean.

Section 1.2 summarizes the contributions of the thesis.

Section 1.3 gives an outline of the rest of the thesis.

1.1 Research Questions

The research questions are listed below, followed by explanations about what they mean.

1. Re perpetual computation in logic programming:

   (a) What kind of logic programs can produce infinite data, whenever its SLD[2] derivation is non-terminating ?

   (b) Under what circumstances can we use unification, between goals in a potentially non-terminating SLD derivation, to decide non-termination of that derivation ?

   (c) When can the infinite data generated by a non-terminating SLD derivation be the same as that computed by unification between goals in the derivation? And in such cases, why are the results the same ?

2. Re coinduction in Horn clause logic:

   (a) What could be a coinductively sound logic that could prove some irregular trees in the greatest complete Herbrand model of a first-order Horn clause logic program?

1.1.1 Explaining Question (1a)

Below are two examples of infinite computation in logic programming, one computes infinite data[3], the other not.

[1] More on the latter point is discussed in Section 7.3.

[2] SLD stands for *S*elective *L*inear *D*efinite clause.

[3] In the context of logic programming, *data* means *term* or *tree*.

Non-terminating SLD derivations can sometimes produce infinite data. For example, consider the clause

$$zeros\ z \to zeros\ (scons\ 0\ z)$$

which has a non-terminating SLD derivation $G_0, G_1, G_2, \ldots$ as follows:

$$\begin{array}{rll} & G_0 :\ zeros\ z_0 & C_1 :\ zeros\ z_1 \to zeros\ (scons\ 0\ z_1) \\ \theta_0 = [z_0 \mapsto scons\ 0\ z_1] \downarrow & & \swarrow \\ & G_1 :\ zeros\ z_1 & C_2 :\ zeros\ z_2 \to zeros\ (scons\ 0\ z_2) \\ \theta_1 = [z_1 \mapsto scons\ 0\ z_2] \downarrow & & \swarrow \\ & G_2 :\ zeros\ z_2 & \end{array}$$

As the derivation goes on, an arbitrarily large atom can be computed by applying the substitutions $\theta_0, \theta_1, \ldots$ to G_0. However, if the head and body of the above clause are swapped, as follows:

$$zeros\ (scons\ 0\ z) \to zeros\ z$$

then there would be a non-terminating SLD derivation $G_0, G_1, G_2, \ldots$ that does not compute infinite data, given below.

$$\begin{array}{ll} G_0 :\ zeros\ z_0 & C_1 :\ zeros\ (scons\ 0\ z_1) \to zeros\ z_1 \\ \downarrow & \swarrow \\ G_1 :\ zeros\ (scons\ 0\ z_0) & C_2 :\ zeros\ (scons\ 0\ z_2) \to zeros\ z_2 \\ \downarrow & \swarrow \\ G_2 :\ zeros\ (scons\ 0\ (scons\ 0\ z_0)) & \end{array}$$

It should now be clear that not all non-terminating SLD derivations can compute infinite data. Hence question (1a). Similar questions have been asked and answered for functional programming [9] but never for logic programming.

1.1.2 Explaining Questions (1b) and (1c)

Described below is the role that unification (without occur-check) can play in coinductive logic programming, and its limitation.

Unification between goals in an SLD derivation can imply non-termination of the derivation, and be used to compute the infinite data that results from the non-terminating derivation. For instance, consider the program

$$zeros\ z \to zeros\ (scons\ 0\ z)$$

A derivation of this program was given earlier. Unification between G_0 : *zeros* z_0 and G_1 : *zeros* z_1, together with θ_0, yields a mapping $z_0 \mapsto \mathit{scons}\ 0\ z_0$ that represents the substitution of z_0 by the infinite term *scons* 0 (*scons* 0 $\cdots$). Thus, curiously in this example, unification between goals, and existence of non-terminating derivation, coincide. In addition, the answer substitution given by unification between goals, is the same as that by the non-terminating derivation. This phenomenon was first discovered by Gupta and his colleagues [1]. They created an coinductively sound algorithm, called CoLP, that extends SLD-resolution by the rule that "a goal succeeds if it unifies with an ancestor goal".

However, the author of the thesis noticed that *unification between goals does not always imply existence of non-terminating derivation*, and in addition, *even if there is a non-terminating derivation, the answers computed by the two may not be the same.* Here is one example for each of the two points.

The first point is supported by the program:

$$q\,x \to p\,x\,(s\,x)$$

$$p\,x\,x \to q\,(s\,x)$$

A failed derivation is given below:

$$\begin{array}{rlcl}
 & G_0 :\ p\,x_0\,(s\,x_0) & & C_1 :\ q\,x_1 \to p\,x_1\,(s\,x_1) \\
 & \downarrow & \swarrow & \\
 & G_1 :\ q\,x_0 & & C_2 :\ p\,x_2\,x_2 \to q\,(s\,x_2) \\
\theta = [x_0 \mapsto (s\,x_2)] & \downarrow & \swarrow & \\
 & G_2 :\ p\,x_2\,x_2 & &
\end{array}$$

The above SLD derivation fails because it cannot proceed from G_2, but on the other hand G_2 still unifies with G_0 (without occur-check). This example shows that if in an SLD derivation two goals unify, then we are not guaranteed that the derivation is non-terminating.

The second point is supported by a non-terminating derivation related to the clause

$$p\,(s\,y)\,x \to p\,y\,(s\,x)$$

as follows:

$$
\begin{array}{rll}
 & G_0 : \; p\, y_0\, x_0 & C_1 : \; p\,(s\, y_1)\, x_1 \to p\, y_1\, (s\, x_1) \\
\theta_0 = [x_0 \mapsto (s\, x_1)] \downarrow & & \\
 & G_1 : \; p\,(s\, y_0)\, x_1 & C_2 : \; p\,(s\, y_2)\, x_2 \to p\, y_2\, (s\, x_2) \\
\theta = [x_1 \mapsto (s\, x_2)] \downarrow & & \\
 & G_2 : \; p\,(s\,(s\, y_0))\, x_2 &
\end{array}
$$

The result by this derivation is $py_0(s(s\cdots))$, while the result by unification (without occur-check) between G_0 and G_1 is $p\,(s\,(s\,\cdots))\,(s\,(s\,\cdots))$. This example shows that when some result is produced by a non-terminating SLD derivation, unification between goals in the derivation may produce a different result.

So questions (1b) and (1c) are asked. Non-termination decision is an interesting logic programming problem [10], and answering question (1b) could provide a new algorithm to deal with it. The goal of studying unification between goals in an SLD derivation is to seek a terminating algorithm that captures infinite SLD derivations and preserves the resulting infinite data (if any) [1]. Thus answering question (1c) helps researchers know how this goal can be achieved.

1.1.3 Explaining Question (2a)

Irregular trees are commonly involved in logic programming but currently they are not well handled by coinductive algorithms.

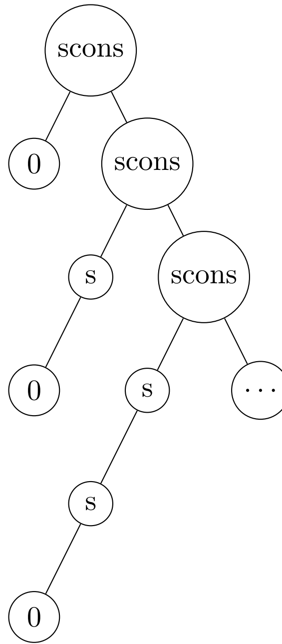
Some logic programs can compute irregular trees, i.e., trees that have an infinite number of distinct sub-trees. For example, the clause

$$\mathit{from}\ (s\ x)\ y \to \mathit{from}\ x\ (\mathit{scons}\ x\ y)$$

has a non-terminating SLD derivation:

$$
\begin{array}{rll}
 & G_0 : \mathit{from}\ 0\ y_0 & C_1 : \mathit{from}\ (s\, x_1)\ y_1 \to \mathit{from}\ x_1\ (\mathit{scons}\ x_1\ y_1) \\
\theta_0 = [y_0 \mapsto \mathit{scons}\ 0\ y_1] \downarrow & & \\
 & G_1 : \mathit{from}\ (s\, 0)\ y_1 & C_2 : \mathit{from}\ (s\, x_2)\ y_2 \to \mathit{from}\ x_2\ (\mathit{scons}\ x_2\ y_2) \\
\theta_1 = [y_1 \mapsto \mathit{scons}\ (s\, 0)\ y_2] \downarrow & & \\
 & G_2 : \mathit{from}\ (s\,(s\, 0))\ y_2 &
\end{array}
$$

What is being computed is the following irregular tree t for y_0:



The atomic formula *from* 0 t is in the greatest complete Herbrand model of the program. Existing coinductive operational semantics for first-order Horn clause logic programming are: infinite SLD-resolution (iSLD), CoLP [1] and proof-relevant corecursive resolution (CR) [2]. But, iSLD is non-terminating by definition, and CoLP can only work with regular trees in principle, and CR is designed to work with finite trees. Therefore, none of these semantics can prove within finite time that *from* 0 t is in the greatest complete Herbrand model. So question (2a) is posed. Answering this question would push forward the capability of coinductive operational semantics for logic programming.

1.2 Contributions

This thesis condenses the technical materials that were created by the author during September 2016 – June 2018, and that were published in two conferences: International Conference on Logic Programming (ICLP) 2017 and European Symposium on Programming (ESOP) 2019. The ICLP paper concerns *productive corecursion*, whose novelty is the identification of sufficient conditions for logic programs to be productive, and the revelation of the mechanism by which unification between goals in an SLD derivation preserves the correct infinite answer. The ESOP paper presents a framework called *Coinductive Uniform Proof* (CUP) for coinduction in first-order Horn clause logic, which can deal unprecedentedly with both non-cyclic SLD derivations and irregular trees.

The thesis uses new notation, which seem more succinct, to rephrase the contents of the ICLP paper, and reformulates the proofs therein with additional details and clearer organization. It presents a minimal version of CUP, but not in the language of category theory that prevails in the ESOP paper. The reader will find here the full proof of soundness of CUP relative to the greatest complete fixed-point model (which involves construction of a coinductive invariant, and which was published in part by the ESOP paper) and the full proof of soundness of CUP relative to the logic $\mathbf{iFOL}_{\blacktriangleright}$ (which was omitted in whole in the ESOP paper). In the end, there is an involved comparison among CUP and related systems such as μMALL, Cyclic Proof and the Abella prover. Interesting connections are found to exist among these systems, which identify future research directions that concern different but potentially equivalent rules of induction and coinduction in the proof-theoretic context.

1.3 Outline

The contents of the technical chapters of the thesis are summarized below.

Chapter 2 provides the basic definitions and notation used through the technical parts of the thesis.

Chapter 3 tackles questions (1a)(1b) and (1c). Theorem 3.33 answers question (1a) whilst Theorem 3.54 answers questions (1b) and (1c).

Chapter 4 answers question (2a) by defining the logic *Coinductive Uniform Proof* (CUP).

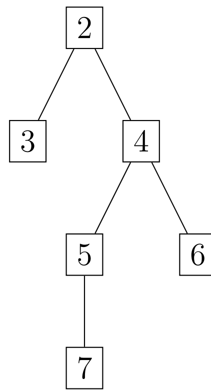
Chapter 5 proves that CUP is coinductively sound: formulae provable by CUP are true with respect to the greatest complete Herbrand model of logic programs.

Chapter 6 shows that CUP is constructive by associating it with intuitionistic logic.

Chapter 7 compares CUP to related systems μMALL, the Abella Prover, and Cyclic Proof.

Chapter 8 concludes the thesis and discusses future work. Below is a dependency diagram for the technical chapters. Note that Chapter 3 only depends on Sections

2.1, 2.2 and 2.3 of Chapter 2, while chapters on CUP depend on Chapter 2 as a whole.



Chapter 2

Basic Notions in Logic Programming

The technical construction in this thesis relies on the notions and notation presented in this chapter. There are certain definitions that are *not* part of the standard references: for instance, the syntax of *guarded λ-terms* and formulae that are based on such terms. The idea of CUP involves using guarded λ-terms to encode infinite trees in logic programming. The key component of a typical guarded λ-term is a *guarded fixed-point* that is based on the standard concept of fixed-points in λ-calculus [11, §1.10] [12, §11.11]. There are some differences between the guarded fixed-point defined in this thesis, and the fixed-point defined in the standard literature. First, the author follows the suggestion of Dr Henning Basold and uses the key word **fix** as a *binder*, not as a *combinator*. For instance, instead of writing $\mathbf{fix}\,(\lambda x.\, M)$, he writes $(\mathbf{fix}\; x.\, M)$. This notation is shorter and highlights the top-level variable which is x in the example above. Second, extra restrictions are introduced on the syntax of fixed-points so that they can reliably encode infinite trees in logic programming.

Section 2.1 defines finite and infinite first-order terms and the term metric.

Section 2.2 introduces unification with and without occur-check.

Section 2.3 presents standard model theory for logic programming.

Section 2.4 gives the first-order type system of CUP. Other versions of CUP have a more complicated type system, but it is the first-order type that CUP actually needs.

Section 2.5 introduces the only trace of higher-order in CUP, which concerns variables that can range over arbitrary first-order function symbols. Such variables are called *higher-order variables*, and are used in very restricted ways, i.e., they are always bounded by the binder **fix** inside a term and are never quantified.

Section 2.6 defines the notion of *guarded lambda term*, which we use to encode (possibly infinite and irregular) first-order terms that are encountered in logic programming. This is part of the CUP syntax.

Section 2.7 sets up the language of CUP, called *H-formula*, which is a Horn-clause-like language built using first-order predicates over guarded lambda terms, with the logic connectives $\forall, \to, \wedge$. We arrive at H-formula via a detour, through the language of hereditary Harrop formula, hinting at the point that taking H-formulae as both programs and goals in CUP, is actually working outwith Horn clause logic, but within hereditary Harrop formula logic.

A single sentence of the form "$\ldots A$ (A') $\ldots B$ (resp. B') $\ldots$" is a shorthand for two sentences "$\ldots A \ldots B \ldots$" and "$\ldots A' \ldots B' \ldots$" . For example, see Definitions 2.8. The word "iff" abbreviates "if and only if".

2.1 Well-formed Terms

The notion of a *term* is fundamental in formal logic and its applications. The reader will see this concept in logic programming, in functional programming, in interactive theorem proving, and in logic frameworks that are blueprints of theorem provers. In all of these fields, a term is essentially a composite of symbols formed according to some rules and is supposed to assume some interpretation. The notion of a *term* for logic programming is formally defined in this section.

Notation 2.1. Natural numbers are denoted using $0, 1, 2, 3, \ldots$ and we use $\mathbb{N}$ to denote the set of all natural numbers.

2.1.1 The Tree Language

Terms are tree-structured objects. A tree language [13, §25] is a systematic way to label nodes in a tree using lists of natural numbers. Each number list is called a

word and acts as a serial number of a tree node. The collection of all such labels of a tree, called a *tree language*, captures the structure of the tree.

Definition 2.2. A *word* refers to a finite and possibly empty list of natural numbers.

Notation 2.3. An empty word is denoted ϵ. A non-empty word with $k \geq 1$ numbers has the form $[n_1, \ldots, n_k]$. $\mathbb{N}^*$ denotes the set of all words.

Definition 2.4. The *length* of a word w is k iff w has the form $[n_1, \ldots, n_k]$. The *length* of w is 0 iff w is ϵ.

Notation 2.5. If a word w has length k then we write $\mathit{length}(w) = k$.

Notation 2.6. Given a word w, we write wi to denote the new word obtained by adding the number i to the end of the word w. Observe that wi is $[i]$, iff w is ϵ, while wi is $[n_1, \ldots, n_k, i]$, iff w is $[n_1, \ldots, n_k]$. The notation wj should be understood similarly.[1]

Definition 2.7. We say L is a *finitely branching tree language* (*tree language* for short), iff L is a subset of $\mathbb{N}^*$, such that for all words $wi \in \mathbb{N}^*$, if $wi \in L$, then

- $w \in L$, and
- for all $j < i$, $wj \in L$, and
- the set $\{wj \in L \mid j > i\}$ is finite.

Definition 2.8. A tree language L is *finite* (*infinite*) iff L is a finite (resp. infinite) set.

A tree language only gives a tree structure that can be regarded as a hotel with no guest. The nodes of the tree are like the rooms of the hotel, and the words in the tree language are like the door numbers of the rooms. The guests for this tree hotel are *elementary symbols*.

2.1.2 Elementary Symbols for Terms

The basic building blocks of terms are individual symbols. There are permanent symbols in a term which cannot be replaced, called *constants* or *function symbols*, and there are flexible symbols, called *variables*, that can be replaced by other symbols or even terms. Here are the formal definitions of the basic symbols.

[1] Intuitively, the notation wi, wj denote non-empty words while emphasizing that the last number in the word is i or j .

Definition 2.9. A *function symbol* is some designated letter, such as f, g, h and a, b, c.

Notation 2.10. We may use other names as function symbols, such as `nil`, `cons`, `suc`, etc.

The arity of a function symbol refers to the (maximum) number of arguments that this function symbol can take.

Definition 2.11. The *arity* of a function symbol f is a natural number fixed to f.

Notation 2.12. The arity of function symbol f is denoted $\mathit{arity}(f)$.

Definition 2.13. A *signature* is a finite set of function symbols.

Notation 2.14. We use the letter Σ or its variants such as Σ' and Σ_1, to denote a signature.

Definition 2.15. A *predicate* is a designated function symbol to be distinguished from any other function symbol that is not designated as a predicate.

Notation 2.16. We use p, q, r and variants thereof to denote predicates. Sometimes we use a letter-string as a predicate, e.g., `from`. Given a signature Σ, the set of all predicates in Σ is denoted Π. We require that $\Pi \subset \Sigma$.

Definition 2.17. A *variable* is some designated letter, such as x, y, z, which is distinguished from a function symbol.

Notation 2.18. We use Var to denote a countably infinite set of variables, and require that $\mathit{Var} \cap \Sigma = \emptyset$ for all Σ.

Definition 2.19. The *arity* of a variable is 0.

Now with all the symbols at our disposal and a tree structure to accommodate them, we can associate nodes of the tree with the symbols.

2.1.3 Well-Formed Terms

Basic symbols are *not* arbitrarily plugged into nodes of a tree skeleton in order to form a term. Instead, nodes and symbols are associated in a meaningful way, and the terms so formed are called *well-formed*. It is well-formed terms that are useful and of interest.

Definition 2.20. Given the set *Var* and a signature Σ, a *well-formed* Σ*-term* (*term* for short) is a mapping t from a non-empty tree language L to $Var \cup \Sigma$, such that for all $w \in L$, $arity(t(w))$ equals the cardinality of the set $\{wi \mid wi \in L\}$.[2]

Notation 2.21. We use a boldface lower-case letter, such as $\mathbf{t}$ or $\mathbf{u}$, to denote a finite list of terms. We use a thin italic letter, such as t or u, to denote a single term. $t \in \mathbf{t}$ means t is in the *list* (not set) $\mathbf{t}$. We may display a list in terms of its members and/or sub-lists, e.g., $t_1, \ldots, t_n$ (showing the first and the last member, omitting the rest) or $\mathbf{t}_1, t, \mathbf{t}_2$ (highlighting a member between two sub-lists) or $\mathbf{t}_1, \mathbf{u}, \mathbf{t}_2$ (showing three sub-lists concatenated).

Notation 2.22. The non-empty tree language, which is the domain of a term t, is denoted $domain(t)$.

Definition 2.23. A term is *ground* iff $t(w) \notin Var$ for all $w \in domain(t)$, i.e., no word in its domain is mapped to a variable.

Definition 2.24. A term t is *finite* (*infinite*) iff $domain(t)$ is a finite (resp. infinite) set.

Notation 2.25.

$$\begin{aligned}
\mathbf{Term}(\Sigma) &: \text{The set of all finite terms on } \Sigma \\
\mathbf{Term}^{\infty}(\Sigma) &: \text{The set of all infinite terms on } \Sigma \\
\mathbf{Term}^{\omega}(\Sigma) &: \text{The set of all terms on } \Sigma, \\
&: \mathbf{Term}(\Sigma) \cup \mathbf{Term}^{\infty}(\Sigma)
\end{aligned}$$

When a name is prefixed by **G**, we mean the subset of ground terms, e.g., $\mathbf{GTerm}(\Sigma)$ denotes the set of all finite ground Σ-terms. We add $*$ as a superscript to denote the set of all finite lists, e.g., $\mathbf{Term}^{*}(\Sigma)$ is the set of all finite lists of finite terms.

Definition 2.26. A variable x is called a *free variable* in $\mathbf{t}$ iff there exists a term $t \in \mathbf{t}$, and there exists a word $w \in domain(t)$, such that $t(w)$ is x.

[2] "$arity(t(w))$" reads "the arity of the symbol assigned to w by t". "the cardinality of the set $\{wi \mid wi \in L\}$" is the number of children of w. If you are familiar with the notion of "first-order term" in the sense of Prolog, you can imagine a typical term of our definition as follows. You first imagine a typical Prolog term such as `f(a,h(g(b),X))`, then you regard some arbitrary symbols (except the only variable `X`) in the term, e.g., `h` and `b`, as predicates. However, we do not actually need such complication as "functions on predicates" in our theory. Regarding predicates as a subset of function symbols is merely a way of definition, leading in the end to a first-order logic syntax.

Notation 2.27. We use $FV(\mathbf{t})$ to denote the set of all free variables in $\mathbf{t}$.

Definition 2.28. We say that $\mathbf{u}$ is *variable-disjoint* from $\mathbf{t}$ iff $FV(\mathbf{u}) \cap FV(\mathbf{t}) = \emptyset$.

So far we have seen what a (well formed) term is, and several related notions such as free variables, ground, variable-disjoint, etc. We have also set up a system of notation for sets and lists of terms. Next we will see how to quantify similarity between terms. We will follow the conventional way of "likeness quantification" that can be used to formally claim that an infinite term is the limit of a perpetual computation.

2.1.4 The Term Metric

We follow two steps to get a numerical measure of how similar two terms are. The first step is an "identical up to level n" count. If you and I wear the same outfit but our hats are different, we could say that our clothing is "identical up to the neck". Similarly, for two terms, we compare them from root downwards, level by level, to the deepest level where they remain the same (or alternatively, to the level where they start to differ), and take note of this level number. Then we apply a simple function to map this level number to a value between 0 and 1, as the final measure of how similar the two terms are [13, §25].

The first step is formally described by *truncating* the terms at the same level. If the part between the root and the cutting level for one term is the same as that for the other, then we say that the two terms are the same from the root up till at least that cutting level.

Notation 2.29. To define a truncated term, we introduce a new function symbol $\diamond$ of arity 0, and assume that $\diamond \notin \Sigma$ in the notation $\Sigma \cup \{\diamond\}$.[3]

Definition 2.30. The set of all symbols at the n-th *level* of a term t is $\{t(w) \mid \mathit{length}(w) = n\}$.

Definition 2.31. The *truncation* of a term $t \in \mathbf{Term}^{\omega}(\Sigma)$ at level n is a term $t|_n \in \mathbf{Term}(\Sigma \cup \{\diamond\})$, such that $\mathit{domain}(t|_n) = \{w \in \mathit{domain}(t) \mid \mathit{length}(w) \leq n\}$ and

$$t|_n(w) = \begin{cases} t(w) & \text{if } \mathit{length}(w) < n \\ \diamond & \text{if } \mathit{length}(w) = n \end{cases}$$

[3]The symbol $\diamond$ reads *diamond*.

Definition 2.32. The *distance* between two terms $t, u \in \mathbf{Term}^{\omega}(\Sigma)$ is 0 iff t and u are identical, and is 2^{-n} iff t and u are not identical, and n is the greatest number such that $t|_n$ and $u|_n$ are identical.[4]

Notation 2.33. We use $d(t, u)$ to denote the distance between t and u.

The distance we just defined applies to *all* well formed terms. Next we highlight some subsets of well formed terms that are of particular interest.

2.1.5 First-order Terms, Atoms

A formal language describes objects and their relations. Among well formed terms, some are first-order terms which encode objects and some are atoms (i.e., atomic formulae) which encode relations among objects.

Definition 2.34. A term t on $\Sigma(\supset \Pi)$, is *first-order* iff $t(w) \notin \Pi$ for all $w \in$ *domain*(t).

Notation 2.35. We use the prefix $1st$ to denote the subset of first-order terms, e.g. $1st\mathbf{Term}(\Sigma)$ is the set of all first-order finite terms on Σ, and $1st\mathbf{GTerm}^{\omega}(\Sigma)$ is the set of all first-order, possibly infinite, ground terms on Σ.

Definition 2.36. A term t is an *atom* on $\Sigma(\supset \Pi)$ iff $t(\epsilon) \in \Pi$ and for all $w \in$ *domain*(t) if $w \neq \epsilon$ then $t(w) \notin \Pi$. An atom t is *finite* (*infinite* (*ground*)) iff t is a finite (resp. infinite (resp. ground)) term.

Notation 2.37.

$$\begin{array}{ll} \mathbf{Atom}(\Sigma) & : \text{The set of all finite atoms on } \Sigma \\ \mathbf{Atom}^{\infty}(\Sigma) & : \text{The set of all infinite atoms on } \Sigma \\ \mathbf{Atom}^{\omega}(\Sigma) & : \text{The set of all atoms on } \Sigma, \\ & : \mathbf{Atom}(\Sigma) \cup \mathbf{Atom}^{\infty}(\Sigma) \end{array}$$

We use the prefix $\mathbf{G}$ and the superscript $*$ to denote, respectively, the ground subset, and the set of all finite lists. For instance, $\mathbf{GAtom}^{*}(\Sigma)$ is the set of all finite lists of finite ground atoms on Σ.

Assuming a signature Σ, we summarize below some defined sets of terms, where the left braces denote set partitioning. If the superscripts ω in the diagram are

[4] When t and u are not identical, an alternative definition of their distance is 2^{-m} where m is the least number such that $t|_m$ and $u|_m$ are not identical. Moreover, if n is the greatest number such that $t|_n$ and $u|_n$ are identical, then $n \geq 0$, and $m = n + 1$.

systematically replaced by ∞, then the resulting diagram shows the relation among sets of infinite terms. If the superscripts ω in the diagram are systematically omitted, then the resulting diagram shows the relation among sets of finite terms.

$$\mathbf{Term}^{\omega}\begin{cases} 1st\mathbf{Term}^{\omega}\begin{cases} 1st\mathbf{GTerm}^{\omega} \\ \\ \ldots \end{cases} \\ \\ \mathbf{Atom}^{\omega}\begin{cases} \mathbf{GAtom}^{\omega} \\ \\ \ldots \end{cases} \\ \\ \ldots \end{cases}$$

We have now introduced the notion of terms and the subsets of first-order terms and atoms. As fundamental as terms, is the operation of *unification* of terms.

2.2 Unification

If two terms are not identical, it is possible that we could replace some variables in these terms so that the resulting terms are identical. For example $f\ x\ (g\ z)$ and $f\ a\ y$ can be made equal by replacing x by a and y by $g\ z$. This process is called *unification* of the two terms. Unification is essential for understanding how logic programs work, and its importance is multiplied by the close ties between logic programming and other fields of computing such as functional programming and theorem proving.

Although not all pairs of terms can be unified (e.g., $f\ a$ and $g\ x$ cannot be unified), almost every successful unification involves the step of variable replacement. We formalize this step by introducing the notion of *substitution*.

Definition 2.38. A *pre-substitution* is a mapping from *Var* to $1st\mathbf{Term}(\Sigma)$.

Definition 2.39. A *substitution* is a mapping from *Var* to $1st\mathbf{Term}(\Sigma)$, such that for all $x \in Var$, if x is mapped to t and t is not a variable, then $x \notin FV(t)$.

Notation 2.40. Lower case Greek letters (e.g., θ, σ) and their variants (e.g., θ', θ_1) denote substitutions by default, but denote pre-substitutions iff we say so clearly (e.g., "a pre-substitution θ").

Definition 2.41. Given a pre-substitution θ, if there exists $x \in Var$, such that $\theta(x)$ is not a variable, but $x \in FV(\theta(x))$, then we say that θ is *circular*, and that the pair $(x, \theta(x))$ is a *circular component* of θ.

Proposition 2.42. *Every substitution is a pre-substitution. Every non-circular pre-substitution is a substitution.*

Proof. By Definitions 2.38, 2.39 and 2.41. □

Definition 2.43. A substitution (or pre-substitution) θ is *renaming*, iff θ is an injection from Var to Var, i.e., for all $x, y \in Var$ we have $\theta(x), \theta(y) \in Var$ and that if $\theta(x), \theta(y)$ are identical then x, y are identical.

Definition 2.44. The renaming substitution that maps every $x \in Var$ to x itself, is called the *identity substitution.*

Definition 2.45. To *apply* a substitution θ to $\mathbf{t}$, means that, simultaneously, for all $x \in FV(\mathbf{t})$, we replace all occurrences of x in $\mathbf{t}$ by $\theta(x)$, so that we obtain a new list of terms as a result.[5]

Notation 2.46. The new list of terms that results from applying a substitution θ to $\mathbf{t}$ is denoted $\theta(\mathbf{t})$. If $\mathbf{t}$ is $t_1, \ldots, t_n$ then $\theta(\mathbf{t})$ is $\theta(t_1), \ldots, \theta(t_n)$.

Definition 2.47. A *variant* of $\mathbf{t}$ is $\theta(\mathbf{t})$ where θ is a renaming substitution.

Notation 2.48. When we display a substitution (or pre-substitution) θ as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, we imply that for all $y \in Var$ such that $y \notin \{x_1, \ldots, x_n\}$, θ maps y to y itself, unless θ is renaming. In that case, we assume that y is so mapped that Definition 2.43 is not violated.

Notation 2.49. When we apply a sequence $\theta_1, \ldots, \theta_k$ of substitutions in succession to $\mathbf{t}$, the result $\theta_k(\cdots(\theta_1(\mathbf{t}))\cdots)$ is more simply denoted as $\theta_k \cdots \theta_1(\mathbf{t})$.

Definition 2.50. Given the notation $\theta_k \cdots \theta_1(\mathbf{t})$, we call the expression $\theta_k \cdots \theta_1$ a *composition* of substitutions $\theta_1, \ldots, \theta_k$.[6]

[5] A more involved definition of applying substitutions is given at [13, pp.180-181]. Also note that here we do not define what it means to apply a pre-substitution.

[6] Our notion of "composition" is simplified, only stating that "a composition is a string of individual substitutions, and when a composition is applied to a list of terms, the result is computed by applying the individual substitutions one by one". We omit the process of computing an equivalent individual substitution from a composition. Such a process is part of a move involved definition (see, e.g., [13, p. 21]), but is unnecessary for our theory.

Definition 2.51. To perform *occur-check* on a pair $(x, t) \in Var \times 1st\mathbf{Term}(\Sigma)$, means to judge whether the statement "x is not identical to t but $x \in FV(t)$" is true.

Notation 2.52. Let $t, u \in \mathbf{Atom}(\Sigma)$. We write $t \sim_\theta u$ iff running an occur-check enforced unification algorithm (see, e.g. [13, §4]) on t and u returns a substitution θ. We write $t \approx_\sigma u$ iff running an occur-check disabled unification algorithm (see, e.g. [14]) on t and u returns a pre-substitution σ.[7]

Definition 2.53. A *unifier* for t and u, is either a substitution θ such that $t \sim_\theta u$, or a pre-substitution σ such that $t \approx_\sigma u$.

Definition 2.54. A *matcher* for $t, u \in \mathbf{Atom}(\Sigma)$, is a substitution θ such that $\theta(t)$ is u, or $\theta(u)$ is t.

Notation 2.55. We write $t \sqsupseteq_\theta u$ iff θ is a matcher such that $\theta(t)$ is u.[8]

We have now defined unification, and its special case known as term matching. Unification may give a unifier, and term matching may give a matcher. Occur-check can be turned on or off in unification. We also defined substitution and its special forms such as pre-substitution, renaming and identity substitution. A variant of a term is obtained by applying a renaming substitution. Next we give standard definitions of models of logic programs.

2.3 Model Theory

If we regard logic programs as operators (or functions), then fixed-points of such an operator are models of the program. We can choose whether or not to take infinite terms into account. The least fixed-point contains all and only atoms that can be inductively proved using provisions of the program, while the greatest fixed-point contains all and only atoms that cannot be refuted by the program, i.e., atoms that are either inductively provable or causing non-termination of the program. The

[7] A mnemonic for distinguishing $\sim$ from $\approx$ is that $\sim$ is used in Prolog I (hence one tilde $\sim$), while $\approx$ is used in Prolog II (hence two tildes $\approx$). We can observe the difference between these two types of algorithms in a Prolog implementation such as SWI Prolog (`http://www.swi-prolog.org/pldoc/doc_for?object=unify_with_occurs_check/2`).

[8] The notation $t \sqsupseteq u$ suggests that t is more general a term than u, since we can obtain u by applying a substitution to t to make it more specific.

correctness of a coinductive operational semantics for logic programming is judged with respect to the greatest fixed-point.

Definition 2.56. The *Herbrand universe* on Σ is $1st\mathbf{GTerm}(\Sigma)$. The *complete Herbrand universe* on Σ is $1st\mathbf{GTerm}^{\omega}(\Sigma)$.

Definition 2.57. $\mathbf{GAtom}(\Sigma)$ is the *Herbrand base* on Σ. $\mathbf{GAtom}^{\omega}(\Sigma)$ is the *complete Herbrand base* on Σ.

Definition 2.58. A *Herbrand interpretation* (*complete Herbrand interpretation*) on Σ is any subset of the Herbrand base (resp. complete Herbrand base).

Definition 2.59. A *Horn clause* on Σ is a pair $(t, \mathbf{t}) \in \mathbf{Atom}(\Sigma) \times \mathbf{Atom}^{*}(\Sigma)$, where t is the *head*, and $\mathbf{t}$ is the *body*. When we say "*clause*", we mean a Horn clause.

Definition 2.60. An *instance* of a clause $(t, \mathbf{t})$ is $(\theta(t), \theta(\mathbf{t}))$ for some substitution θ. Further, $(\theta(t), \theta(\mathbf{t}))$ is a *ground instance*, iff all atoms in $(\theta(t), \theta(\mathbf{t}))$ are ground.

Definition 2.61. A *variant* of a clause $(t, \mathbf{t})$ is any instance $(\theta(t), \theta(\mathbf{t}))$ where θ is a renaming substitution.

Definition 2.62. A *logic program* on Σ is a finite set of Horn clauses on Σ.

Notation 2.63. We write $(t, \mathbf{t}) \in P$ to mean that $(t, \mathbf{t})$ is a variant of some Horn clause in the logic program P.

Definition 2.64. Fix a signature Σ. The *Herbrand operator* $\mathcal{T}_P$ (*complete Herbrand operator* $\mathcal{T}_P^{\omega}$) of a logic program P is a mapping from and to the set of all Herbrand interpretations (resp. complete Herbrand interpretations). An atom u is in $\mathcal{T}_P(I)$ ($\mathcal{T}_P^{\omega}(I')$) iff there exists a ground instance $(\theta(t), \theta(\mathbf{t}))$ of some clause $(t, \mathbf{t}) \in P$, such that all atoms in $\theta(\mathbf{t})$ are in I (resp. I'), and u is $\theta(t)$.[9]

Definition 2.65. I is a *pre-fixed-point* (*post-fixed-point*) of $\mathcal{T}_P$ iff $\mathcal{T}_P(I) \subseteq I$ (resp. $I \subseteq \mathcal{T}_P(I)$). I is a *fixed-point* of $\mathcal{T}_P$ iff I is both a pre-fixed-point and a post-fixed-point. A (pre- or post-)fixed-point of $\mathcal{T}_P^{\omega}$ is defined similarly.

Definition 2.66. Given $\mathcal{T}_P$ or $\mathcal{T}_P^{\omega}$, a fixed-point I is the least fixed-point (greatest fixed-point) iff for all fixed-points J, $I \subseteq J$ (resp. $J \subseteq I$).

[9] A Herbrand operator is also known as an *immediate consequence operator*. We reserve the latter name to use elsewhere in the thesis.

Proposition 2.67. *Let T be either $\mathcal{T}_P$ or $\mathcal{T}_P^\omega$. Then T has a least fixed-point $lfp(T)$, and a greatest fixed-point $gfp(T)$, given by:*

$$lfp(T) = \bigcap\{J \mid T(J) \subseteq J\} \qquad\qquad gfp(T) = \bigcup\{J \mid J \subseteq T(J)\}$$

Proof. Apply Tarski's fixed-point theorem. See Appendix A. □

Definition 2.68. A logic program P has the following models,

$$\textit{Models of } P \begin{cases} \textit{Inductive Models} \begin{cases} \textit{Least Herbrand Model:} & lfp(\mathcal{T}_P) \\ \textit{Least C. Herbrand Model:} & lfp(\mathcal{T}_P^\omega) \end{cases} \\ \textit{Coinductive Models} \begin{cases} \textit{Greatest Herbrand Model:} & gfp(\mathcal{T}_P) \\ \textit{Greatest C. Herbrand Model:} & gfp(\mathcal{T}_P^\omega) \end{cases} \end{cases}$$

where "C." abbreviates "Complete".

Besides defining models of logic programs, we have also defined the variant of a Horn clause and use the notation $\in$ to mean getting a program clause variant. So far we have set up all the definitions and notation necessary for a theoretical computing expert to examine the new results on perpetual computation in logic programming. Next we will introduce a simple type theory as a meta-language in order to formulate a customized Horn clause syntax that will allow us to approach coinduction that involves irregular trees. We take [11] as a standard reference for λ-calculus.

2.4 A Type System

We set up a simple type system tailored for typing first-order predicates, first-order function symbols, and variables that range over first-order function symbols. It contains only two base types: ι and o. Composite types are built using the base types and the right associative arrow $\to$ without any parentheses, so that nesting on the left is not allowed in a type expression. Moreover, the base type o can only occur at the very right end of a type expression.

Definition 2.69. Let $\mathbb{B}$ satisfy $o \notin \mathbb{B} = \{\iota\}$. A *type* is a member of $\mathbb{T} \cup \mathbb{P}$, where:[10]

[10] Intuitively, $\mathbb{T} = \{\iota,\ \iota \to \iota,\ \iota \to \iota \to \iota,\ \ldots\}$ and $\mathbb{P} = \{o,\ \iota \to o,\ \iota \to \iota \to o,\ \ldots\}$. Given Definition 2.71, any $\tau \in \mathbb{T}$ can be depicted as $\iota^{\mathrm{ar}(\tau)} \to \iota$, while any $\rho \in \mathbb{P}$ can be depicted as $\iota^{\mathrm{ar}(\rho)} \to o$.

$\mathbb{T}$ is the set of *(simple) types*, given by $\mathbb{T} ::= \mathbb{B} \mid \mathbb{B} \to \mathbb{T}$

$\mathbb{P}$ is the set of *proposition types*, given by $\mathbb{P} ::= o \mid \mathbb{B} \to \mathbb{P}$

Notation 2.70. The Greek letters ι and o are reserved as base types. Other lower case Greek letters are used to denote arbitrary types. In particular, we use τ to denote a typical member of $\mathbb{T}$, ρ for a typical member of $\mathbb{P}$, and π for a typical member of $\mathbb{T} \cup \mathbb{P}$.

Definition 2.71. The *arity* of a type π is denoted using $\mathrm{ar}(\pi)$. Let $\mathrm{ar}(\iota) = \mathrm{ar}(o) = 0$. If $\pi \in \mathbb{T} \cup \mathbb{P}$ then $\mathrm{ar}(\iota \to \pi) = \mathrm{ar}(\pi) + 1$.

Definition 2.72. The *order* of a type π is denoted using $\mathrm{ord}(\pi)$. $\mathrm{ord}(\iota) = \mathrm{ord}(o) = 0$. All other types $\pi \in \mathbb{T} \cup \mathbb{P}$ have $\mathrm{ord}(\pi) = 1$.

In order to encode irregular trees, we need variables that range over (non-predicate) function symbols that take arguments. Such variables are called *higher-order variables.*

2.5 Higher-order Variables

We introduce one more class of elementary symbols.

Definition 2.73. We use Var' to denote a countably infinite set of symbols disjoint from Var and from any Σ, and call members of Var' *higher-order variables.*

Definition 2.74. The *arity* of any $x \in \mathit{Var}'$, denoted $\mathit{arity}(x)$, is a positive integer fixed to x.

2.5.1 Variable Contexts

We define a variable context to be a finite set that may contain both first-order and higher order variables.

Definition 2.75. A *context* is a finite subset of $\mathit{Var} \cup \mathit{Var}'$.

Notation 2.76. We use the letter Γ or its variants such as Γ' and Γ_1, to denote a context.

With function symbols, variables, higher-order variables and the type system at our disposal, all finite first-order terms and some infinite and possibly irregular first-order terms can be encoded as *guarded λ-terms.*

2.6 Guarded Lambda Terms

We introduce the notion *guarded λ-term* as a type-theoretic encoding of all of $1st\mathbf{Term}(\Sigma)$ and some of $1st\mathbf{Term}^{\infty}(\Sigma)$.

2.6.1 Typed Symbols

We give the rules for assigning types to the elementary symbols.

Notation 2.77. We assume that every function symbol, every variable, and every higher-order variable has a type, and use the expression $s : \pi$ to denote that the (function symbol or variable or high-order variable) s has type π. Given a signature $\Sigma(\supset \Pi)$, the following rules should be observed when assigning types to symbols :

- If $x \in \mathit{Var}$, then $x : \iota$.
- If $y \in \mathit{Var}'$, then $y : \tau$ and $\tau \in \mathbb{T}$ and $\tau \notin \mathbb{B}$.
- If $r \in \Pi$, then $r : \rho$ and $\rho \in \mathbb{P}$.
- If $f \in \Sigma$ and $f \notin \Pi$, then $f : \tau$ and $\tau \in \mathbb{T}$.
- If $s : \pi$, then $\mathit{arity}(s) = \mathrm{ar}(\pi)$.

Definition 2.78. Some useful subsets of a signature $\Sigma(\supset \Pi)$ and a context Γ are defined as follows:

$\Sigma_{\mathbb{T}}$ — The set of all non-predicate symbols in Σ, i.e., $\Sigma \setminus \Pi$.

$\Sigma_{\mathbb{T}}^{n}$ — The subset $\{c : \tau \in \Sigma_{\mathbb{T}} \mid \mathrm{ord}(\tau) \leq n\}$ of $\Sigma_{\mathbb{T}}$.

$\Sigma_{\mathbb{P}}$ — A synonym of Π.

$\Sigma_{\mathbb{P}}^{n}$ — The subset $\{r : \rho \in \Sigma_{\mathbb{P}} \mid \mathrm{ord}(\rho) \leq n\}$ of $\Sigma_{\mathbb{P}}$.

$\Gamma_{\mathbb{T}}$ — A synonym of Γ.

$\Gamma_{\mathbb{T}}^{n}$ — The subset $\{x : \tau \in \Gamma_{\mathbb{T}} \mid \mathrm{ord}(\tau) \leq n\}$ of $\Gamma_{\mathbb{T}}$.

Example 2.79. If $\Sigma = \{a : \iota\}$ then $\Sigma_{\mathbb{T}} = \Sigma_{\mathbb{T}}^{1} = \Sigma_{\mathbb{T}}^{0} \ni a$. If $\Gamma = \{y : \iota \to \iota\}$ then $\Gamma_{\mathbb{T}} = \Gamma_{\mathbb{T}}^{1} \ni y \notin \Gamma_{\mathbb{T}}^{0} = \varnothing$.

The systematic notation of subsets of a signature allows us to precisely specify the components of a *λ-term*.

2.6.2 Lambda Terms

We define λ-terms as variables, constants, application, abstraction and fixed-points. Convertibility relations for λ-terms are defined based on β-reduction and fixed-point transform.

Definition 2.80. The relation $\Sigma; \Gamma \vdash_{(m;n)} M : \tau$, where $m, n \geq 0$ are order constraints and $\tau \in \mathbb{T}$, is defined in Figure 2.1.

$$\frac{c : \tau \in \Sigma^m_{\mathbb{T}}}{\Sigma; \Gamma \vdash_{(m;n)} c : \tau} \qquad \frac{x : \tau \in \Gamma^n_{\mathbb{T}}}{\Sigma; \Gamma \vdash_{(m;n)} x : \tau}$$

$$\frac{\Sigma; \Gamma \vdash_{(m;n)} M : \sigma \to \tau \qquad \Sigma; \Gamma \vdash_{(m;n)} N : \sigma}{\Sigma; \Gamma \vdash_{(m;n)} M\, N : \tau}$$

$$\frac{\Sigma; \Gamma, x : \sigma \vdash_{(m;n)} M : \tau}{\Sigma; \Gamma \vdash_{(m;n)} \lambda x.\, M : \sigma \to \tau} \qquad \frac{\Sigma; \Gamma, x : \tau \vdash_{(m;n)} M : \tau}{\Sigma; \Gamma \vdash_{(m;n)} \mathbf{fix}\, x.\, M : \tau}$$

Figure 2.1: Definition of $\Sigma; \Gamma \vdash_{(m;n)} M : \tau$.

Proposition 2.81. *Provided $m \leq m'$ and $n \leq n'$, if $\Sigma; \Gamma \vdash_{(m;n)} N : \tau$ then $\Sigma; \Gamma \vdash_{(m';n')} N : \tau$.*

Proof. By Definition 2.78, $\Sigma^m_{\mathbb{T}} \subseteq \Sigma^{m'}_{\mathbb{T}}$, and $\Gamma^n_{\mathbb{T}} \subseteq \Gamma^{n'}_{\mathbb{T}}$. □

Definition 2.82. M is a *λ-term* on Σ iff $\Sigma; \Gamma \vdash_{(m;n)} M : \tau$ for some m, n, Γ, τ.

Notation 2.83. Λ_Σ denotes the set of all λ-terms on Σ.

Definition 2.84. The relation $\Sigma; \Gamma \vdash^*_{(m;n)} M : \tau$ holds iff $\Sigma; \Gamma \vdash_{(m;n)} M : \tau$ and M does *not* contain any of the binders $\{\mathbf{fix}, \lambda\}$.

Example 2.85. Let $\Sigma = \{a : \iota, f : \iota \to \iota\}, \Gamma = \{y : \iota \to \iota\}$. Below we compare true

statements with similar but false ones.

True	False
$\Sigma; \Gamma \vdash_{(0;1)} y\ a : \iota$	$\Sigma; \Gamma \vdash_{(0;0)} y\ a : \iota$
$\Sigma; \Gamma \vdash_{(1;0)} f\ a : \iota$	$\Sigma; \Gamma \vdash_{(0;0)} f\ a : \iota$
$\Sigma; \varnothing \vdash_{(1;0)} \lambda x.\, f\ x : \iota \to \iota$	$\Sigma; \varnothing \vdash^{*}_{(1;0)} \lambda x.\, f\ x : \iota \to \iota$
$\Sigma; \varnothing \vdash_{(1;0)} \mathbf{fix}\, x.\, f\ x : \iota$	$\Sigma; \varnothing \vdash^{*}_{(1;0)} \mathbf{fix}\, x.\, f\ x : \iota$

Definition 2.86. A (λ-term) *substitution* is a pair $[N/x]$ where N is a λ-term and $x \in \mathit{Var} \cup \mathit{Var}'$.

Notation 2.87. We use $M\,[N/x]$ to denote the result of applying $[N/x]$ to M in the standard capture-avoiding manner.[11]

Definition 2.88. The following binary relations are over Λ_Σ.

- *β-reduction* ($\longrightarrow_\beta$): $(\lambda x.\, M)N \longrightarrow_\beta M\,[N/x]$
- **fix**-*reduction* ($\longrightarrow_{\mathbf{fix}}$): $(\mathbf{fix}\, x.\, M) \longrightarrow_{\mathbf{fix}} M\,[\mathbf{fix}\, x.\, M/x]$
- *combined reduction* ($\longrightarrow$): The union between the compatible closure[12]of $\longrightarrow_\beta$ and the compatible closure of $\longrightarrow_{\mathbf{fix}}$.
- *convertible relation* ($\equiv$): The equivalence closure[13] of $\longrightarrow$.

Proposition 2.89. *If $M : \tau_1 \equiv N : \tau_2$, then τ_1 and τ_2 are identical.*

Proof. Similar to proofs of standard theorems of λ-calculus. See e.g. [11, §2.11]. □

Notation 2.90. λ-terms may be abbreviated in the following ways.

- Given a λ-term M of the form $F\ N_1 \cdots N_m$, we may use $\vec{N}$ to represent the sub-expression $N_1 \cdots N_m$, and depict M as $F\,\vec{N}$. Then, $|\vec{N}| = m$, and $N \in \vec{N}$ means $N \in \{N_1, \ldots, N_m\}$.

[11]See also [11, §1.6].

[12]The *compatible closure* of a binary relation R over λ-terms (e.g., R could be either β-reduction or **fix**-reduction), refers to the smallest binary relation R' that includes R, which also satisfies that if $(M)R'(N)$, then $(M\ F)R'(N\ F)$, $(F\ M)R'(F\ N)$, $(\lambda x.\, M)R'(\lambda x.\, N)$ and $(\mathbf{fix}\, x.\, M)R'(\mathbf{fix}\, x.\, N)$, for arbitrary x, F.

[13]In other words, the reflexive transitive symmetric closure.

- Similarly, if M has the form $f\ N_1 \cdots N_k\ M'\ N_{k+1} \cdots N_m$, then we may depict M as $f\ \vec{N}_1\ M'\ \vec{N}_2$. We write $N \in \vec{N}_{1,2}$ to mean that $N \in \vec{N}_1$ or $N \in \vec{N}_2$.

- Given a λ-term M of the form $\lambda x_1 : \iota. \cdots \lambda x_m : \iota. N$, we may use $\lambda\vec{x} : \iota.$ to represent the sub-expression $\lambda x_1 : \iota. \cdots \lambda x_m : \iota.$, and depict M as $\lambda\vec{x} : \iota. N$. Then, $|\vec{x}| = m$.

The liberally defined notion of fixed-point does not guarantee that a fixed-point necessarily corresponds to an infinite first-order term. We now introduce further syntactical restrictions to filter out all useless fixed-points[14].

2.6.3 Guarded Fixed-points

We define a guarded fixed-point in such a way that it is easy to check that they correspond to infinite first-order terms and that they can encode some irregular terms. The formal definition is phrased to capture the shape of certain concrete examples at hand during the author's research.

Definition 2.91. The relation $\Sigma; \varnothing \vdash_{\triangleright} M : \tau$ is defined in Figure 2.2. M is a *guarded fixed-point* on Σ iff $\Sigma; \varnothing \vdash_{\triangleright} M : \tau$ for some τ.

$$\frac{\{\Sigma; \vec{x} : \iota \vdash^{*}_{(1;0)} N : \iota \mid N \in \vec{N}_{1,2,3}\} \qquad \left[\begin{array}{l} f : \tau' \in \Sigma^1_{\mathbb{T}} \\ y : \tau \notin \vec{x} \\ \mathrm{ar}(\tau') = |\vec{N}_1| + 1 + |\vec{N}_3| \\ \mathrm{ar}(\tau) = |\vec{x}| = |\vec{N}_2| \end{array}\right]}{\Sigma; \varnothing \vdash_{\triangleright} \mathbf{fix}\, y.\, \lambda\vec{x}.\, f\ \vec{N}_1\ (y\ \vec{N}_2)\ \vec{N}_3 : \tau}$$

Figure 2.2: Definition of $\Sigma; \varnothing \vdash_{\triangleright} M : \tau$.

We explain Figure 2.2. A guarded fixed-point has the form ($\mathbf{fix}\, y.\, \textit{Something}$) where *Something* is an abstraction of the form ($\lambda x_1. \ldots \lambda x_n. \textit{Some-Term}$), abbreviated ($\lambda\vec{x}. \textit{Some-Term}$), where the λ-bound variables have arity 0 and *Some-Term* is an application where the function symbol f is applied to as many arguments as it can take. The arguments of f involve a particular term ($y\ \vec{N}_2$) that is applying the

[14]Some useful fixed-points are filtered out too. So the design of the filter is open to future improvement.

(higher-order) variable y to as many arguments as it can take. In addition, the arity of y equals the number of arguments expected by the abstraction *Something*, and this number is denoted $|\vec{x}|$. All terms in the list $\vec{N_i}$ ($i \in \{1, 2, 3\}$) are built using (non-predicate) function symbols from Σ and variables from $\vec{x}$, but do not contain the binders $\{\mathbf{fix}, \lambda\}$.

Proposition 2.92. *A guarded fixed-point is a λ-term.*

Proof. By Definitions 2.91, 2.84, 2.80 and Proposition 2.81. □

With the notion of guarded fixed-point, we can now focus on a group of λ-terms that are guaranteed to encode what they are supposed to encode. Such λ-terms are called *guarded λ-terms*.

2.6.4 Guarded Lambda Terms

A guarded λ-term is a binder-free first-order λ-term, or a guarded fixed-point with suitable arguments, or their equivalents.

Definition 2.93. The relations $\vdash_{\mathrm{I}}$, $\vdash_{\mathrm{II}}$ and $\vdash_g$ are defined by Figure 2.3.

$$\frac{\Sigma; \Gamma \vdash^*_{(1;0)} M : \iota}{\Sigma; \Gamma \vdash_{\mathrm{I}} M : \iota}$$

$$\frac{\Sigma; \varnothing \vdash_{\triangleright} M : \tau \qquad \mathrm{ar}(\tau) = |\vec{N}| \qquad \{\Sigma; \Gamma \vdash_{\mathrm{I}} N : \iota \mid N \in \vec{N}\}}{\Sigma; \Gamma \vdash_{\mathrm{II}} M\, \vec{N} : \iota}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathrm{I}} M : \tau}{\Sigma; \Gamma \vdash_g M : \tau} \qquad \frac{\Sigma; \Gamma \vdash_{\mathrm{II}} M : \tau}{\Sigma; \Gamma \vdash_g M : \tau} \qquad \frac{\Sigma; \Gamma \vdash_g N : \tau \qquad N \equiv M}{\Sigma; \Gamma \vdash_g M : \tau}$$

Figure 2.3: Definition of $\vdash_{\mathrm{I}}$, $\vdash_{\mathrm{II}}$ and $\vdash_g$. The letter g refers to "guarded".

Definition 2.94. $M : \tau$ is a *type-I guarded λ-term* iff $\Sigma; \Gamma \vdash_{\mathrm{I}} M : \tau$. It is a *type-II guarded λ-term* iff $\Sigma; \Gamma \vdash_{\mathrm{II}} M : \tau$. It is a *guarded λ-term* iff it is a type-I or type-II guarded λ-term, or it is convertible ($\equiv$) to a type-I or type-II guarded λ-term. In other words, $\Sigma; \Gamma \vdash_g M : \tau$.

Proposition 2.95. *A guarded λ-term is a λ-term of type ι.*

Proof. By Definition 2.94 and Proposition 2.89. □

Notation 2.96. We denote the set of all guarded λ-terms on Σ by Λ^G_Σ.

Since guarded λ-terms may involve higher-order variables, we can further sort them into a first-order group and a higher-order group, despite the fact that they all encode first-order terms of logic programming.

2.6.5 Order of Guarded Lambda Terms

The order of a guarded λ-term is solely determined by the presence or absence of a higher-order variable in it.

Definition 2.97. $M \in \Lambda^G_\Sigma$ is *higher-order* iff there is $y : \tau$ **fix**-bound[15] in M and $\mathrm{ord}(\tau) > 0$. Otherwise, M is *first-order*.

Example 2.98. Below are examples of first-order and higher-order guarded λ-terms.

- First-order: $\mathbf{fix}\, y.\, f\ y$ and $f\ a$ where $y : \iota$ is a variable, and $f : \iota \to \iota$, $a : \iota$ are function symbols.
- Higher-order: $(\mathbf{fix}\, y.\, \lambda x.\, f\ (y\ x))\ a$ where $y : \iota \to \iota$ is a higher-order variable, $x : \iota$ is a variable, and $f : \iota \to \iota$, $a : \iota$ are function symbols.

The correspondence between a guarded λ-term and the first order term that it encodes can be formally established.

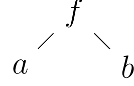
2.6.6 Mapping From Λ^G_Σ to $1st\mathbf{Term}^\omega(\Sigma)$

Lambda terms in Λ^G_Σ can be associated to terms in $1st\mathbf{Term}^\omega(\Sigma)$. This is straightforward for type-I guarded terms. A typical type-II guarded term has an infinite reduction chain that consists of an endless series of equivalent guarded terms. We show that this series converge to an infinite tree that is what all the guarded terms in the series are supposed to encode. We define a reduction named *fix-beta* as the unit operation on guarded terms, omitting intermediate reduction products that are not needed for defining the infinite tree to which the reduction series converge.

Examples 2.99 and 2.100 give some intuition.

Example 2.99. $f\ a\ b \in \Lambda^G_\Sigma$ corresponds to the $1st\mathbf{Term}^\omega(\Sigma)$ member:

[15] We say that y is **fix**-*bound* in a term M if M has a sub-term of the form $(\mathbf{fix}\, y.\, N)$.



Example 2.100. The λ-term $\mathbf{fix}\,x.\,f\,x \in \Lambda^G_\Sigma$ has the following reduction sequence, and each λ-term in the sequence can be rendered as a tree:

λ-terms	$\mathbf{fix}\,x.\,f\,x$	$\longrightarrow$	$f\,\mathbf{fix}\,x.\,f\,x$	$\longrightarrow$	$f\,(f\,\mathbf{fix}\,x.\,f\,x)$	$\cdots$
trees	$\mathbf{fix}\,x.\,f\,x$	$\longrightarrow$	f — $\mathbf{fix}\,x.\,f\,x$	$\longrightarrow$	f — f — $\mathbf{fix}\,x.\,f\,x$	$\cdots$

Therefore, $\mathbf{fix}\,x.\,f\,x$ corresponds to the $1st\mathbf{Term}^\omega(\Sigma)$ member $f - f - f - \cdots$ which is an infinite term.

Definition 2.101. $\dot{\Im}$ is a function from Λ^G_Σ to $1st\mathbf{Term}^\omega(\Sigma \cup \{\diamond\})$,[16] such that:

- $\dot{\Im}(s) = s$, if $s \in \Sigma^0_{\mathbb{T}} \cup \mathit{Var}$.
- $\dot{\Im}(f\ N_1 \cdots N_k) =$ the tree with root f and children $\dot{\Im}(N_1)\ \cdots\ \dot{\Im}(N_k)$
- $\dot{\Im}(M) = \diamond$ otherwise.

Example 2.102. The following table shows how $\dot{\Im}$ evaluates some guarded λ-terms that contain the binder $\mathbf{fix}$.

$\dot{\Im}(\mathbf{fix}\,x.\,f\ x)$	$\dot{\Im}(f\ \mathbf{fix}\,x.\,f\ x)$	$\dot{\Im}(f\ (f\ \mathbf{fix}\,x.\,f\ x))$
$\diamond$	f — $\diamond$	f — f — $\diamond$

Definition 2.103. Let $k \in \mathbb{N}$ and $0 \leq j \leq k-1$. A type-II guarded λ-term $F\ \vec{X}$ $\mathbf{fix}\beta$*-reduces* to M, denoted $F\ \vec{X} \longrightarrow_{\mathbf{fix}\beta} M$, iff there exist F' and $N_0, \ldots, N_k$ such that:

1. $F \longrightarrow_{\mathbf{fix}} F'$;
2. for all j, $N_j \longrightarrow_\beta N_{j+1}$;

[16] $\dot{\Im}$ reads "approximate term-value".

3. N_k cannot be further reduced by $\longrightarrow_\beta$;

4. N_0 is $F' \vec{X}$. N_k is M.

We use $\longrightarrow_{\text{Fix}\beta}$ to denote the compatible closure of $\longrightarrow_{\mathbf{fix}\beta}$. The statement that "there exist $M_0, \ldots, M_k$, such that $M_j \longrightarrow_{\text{Fix}\beta} M_{j+1}$ for all j", is denoted by $M_0 \longrightarrow^k_{\text{Fix}\beta} M_k$.

Example 2.104. A typical type-II guarded λ-term

$$(\mathbf{fix}\, y.\, \lambda\vec{x}.\, h\, \vec{N_1}\, (y\, \vec{N_2})\, \vec{N_3})\, \vec{M}$$

written as $F\, \vec{M}$ for short, $\mathbf{fix}\beta$-reduces ($\longrightarrow_{\mathbf{fix}\beta}$) to

$$(h\, \vec{N_1}\, (F\, \vec{N_2})\, \vec{N_3})\, [\vec{M}/\vec{x}]$$

Example 2.105. Let N denote the guarded fixed-point $\mathbf{fix}\, y.\, \lambda x.\, h\, x\, (y\, (f\, x))$. Then, the type-II guarded λ-term $N\, z$ has the following reductions:

$$\begin{array}{ll} & N\, z \\ \longrightarrow_{\mathbf{fix}\beta} & h\, z\, (N\, (f\, z)) \\ \longrightarrow_{\text{Fix}\beta} & h\, z\, (h\, (f\, z)\, (N\, (f\, (f\, z)))) \\ \longrightarrow_{\text{Fix}\beta} & h\, z\, (h\, (f\, z)\, (h\, (f\, (f\, z))\, (N\, (f\, (f\, (f\, z)))))) \end{array}$$

Proposition 2.106. *If $M_0 \longrightarrow^k_{\text{Fix}\beta} M_k$ and M_0 is a type-II guarded λ-term, then there exists one and only one $t \in 1st\,\boldsymbol{Term}^\infty(\Sigma)$, such that*

$$\lim_{k\to\infty} d(t, \dot{\Im}(M_k)) = 0$$

Proof. The j-th level ($j \in \mathbb{N}$) of t is defined in the same way as the j-th level of $\dot{\Im}(M_{j+1})$. □

Motivated by Proposition 2.106, we have the following definition.

Definition 2.107. If $M_0 \longrightarrow^k_{\text{Fix}\beta} M_k$ and M_0 is a type-II guarded λ-term, then the $t \in 1st\mathbf{Term}^\infty(\Sigma)$, such that $d(t, \dot{\Im}(M_k)) \to 0$ as $k \to \infty$, is called *the intended infinite term of M_0*.

Definition 2.108. $\Im$ is a function from Λ_Σ^G to $1st\mathbf{Term}^\omega(\Sigma)$,[17] such that:

- $\Im(M) = \dot{\Im}(M)$, if M is a type-I guarded λ-term.
- $\Im(N) = t$, if N is a type-II guarded λ-term and t is the intended infinite term of N.
- $\Im(M) = \Im(N)$, if $M \equiv N$.

Observe that every infinite tree encoded by a guarded term is guaranteed to have a finite number of distinct free variables. Now we have finished the constructive journey from basic symbols to lambda terms, and then to guarded lambda terms. Based on these we can define formulae.

2.7 Formulae

Atomic formulae are built using predicates and guarded terms. Composite formulae are built using atomic formulae and logic connectives from first-order predicate logic. Once we have formally defined the mapping from guarded terms to trees, it is easy to define the mapping from atomic formulae to trees. The convertibility relation between formulae is based on convertibility between guarded terms that occur in the formulae.

Definition 2.109. The relations $\Sigma;\Gamma \Vdash_a \varphi$ and $\Sigma;\Gamma \Vdash \varphi$ are defined by Figure 2.4.

$$\frac{q : \rho \in \Sigma^1_{\mathbb{P}} \quad \mathrm{ar}(\rho) = |\vec{M}| \quad \{\Sigma;\Gamma \vdash_g M : \iota \mid M \in \vec{M}\}}{\Sigma;\Gamma \Vdash_a q\, \vec{M}}$$

$$\frac{\Sigma;\Gamma \Vdash_a \varphi}{\Sigma;\Gamma \Vdash \varphi} \qquad \frac{\Sigma;\Gamma, x : \iota \Vdash \varphi}{\Sigma;\Gamma \Vdash \forall x : \iota.\, \varphi} \qquad \frac{\Sigma;\Gamma, x : \iota \Vdash \varphi}{\Sigma;\Gamma \Vdash \exists x : \iota.\, \varphi}$$

$$\frac{\Sigma;\Gamma \Vdash \varphi \qquad \Sigma;\Gamma \Vdash \psi \qquad \Box \in \{\wedge, \vee, \rightarrow\}}{\Sigma;\Gamma \Vdash \varphi \,\Box\, \psi}$$

Figure 2.4: The relations $\Sigma;\Gamma \Vdash_a \varphi$ and $\Sigma;\Gamma \Vdash \varphi$.

Definition 2.110. φ is a *guarded atom* on Σ iff $\Sigma;\Gamma \Vdash_a \varphi$ for some Γ.

[17] $\Im$ reads "term-value".

Notation 2.111. Given some Σ, we use A_Σ to denote the set of all guarded atoms on Σ.

Notation 2.112. Let $p\,\vec{N}$ and $p\,\vec{N}'$ be two guarded atoms, where $\vec{N}$ ($\vec{N}'$) abbreviates $N_1 \cdots N_m$ (resp. $N'_1 \cdots N'_m$). We write $p\,\vec{N} \equiv p\,\vec{N}'$ iff $N_k \equiv N'_k$ for all $k \in \{1, ..., m\}$.

Definition 2.113. $\Im$ is a function from A_Σ to $\mathbf{Atom}^\omega(\Sigma)$[18], such that:

- $\Im(p\,N_1 \cdots N_k) = $

$$\begin{array}{ccc} & p & \\ \swarrow & & \searrow \\ \Im(N_1) & \cdots & \Im(N_k) \end{array}$$

Definition 2.114. φ is a *formula* on Σ iff $\Sigma; \Gamma \Vdash \varphi$ for some Γ.

Definition 2.115. A formula φ is *closed* iff $\Sigma; \varnothing \Vdash \varphi$.

We can sort formulae into order groups based on the order of the lambda terms that occur in the formulae.

2.7.1 Order of Formula

Definition 2.116. A formula φ is *first-order* iff all guarded λ-terms involved are first-order. Otherwise φ is *higher-order*.

Example 2.117. We show examples of first-order and higher-order formulae. Assume function symbols $\texttt{from} : \iota \to \iota \to o$ (which is also a predicate), $s : \iota \to \iota$ and $\texttt{cons} : \iota \to \iota \to \iota$.

- A closed first-order formula:

$$\forall x : \iota.\, \forall y : \iota.\, \texttt{from}\,(s\ x)\ y \to \texttt{from}\,x\ (\texttt{cons}\ x\ y)$$

- A closed higher-order formula:

$$\forall x : \iota.\, \texttt{from}\,x\ (F\ x)$$

  where F is $\mathbf{fix}\,y.\,\lambda z.\,\texttt{cons}\ z\ (y\ (s\ z))$ with $y : \iota \to \iota, z : \iota$.

We have defined the set of all formulae, but it is a subset of it, called *hereditary Harrop formulae* [15, 16], that is suitable for logic programming.

[18]Definition 2.113 overloads the term-value function $\Im$ of Definition 2.108 to calculate the *atom-value* of guarded atoms.

2.7.2 The Hereditary Harrop Language

Hereditary Harrop formulae are given by two mutually inductive sets. In general, proving a Horn clause with respect to a Horn clause program is a task within the hereditary Harrop language.

Definition 2.118. On some Σ, the set of *hereditary Harrop goal formulae* (denoted G), and the set of *hereditary Harrop program formulae* (denoted D), are defined mutual-inductively:

$$G ::= A_{\Sigma} \mid G \wedge G \mid \forall x : \iota. G \mid D \to G \mid \exists x : \iota. G \mid G \vee G$$

$$D ::= A_{\Sigma} \mid D \wedge D \mid \forall x : \iota. D \mid G \to D$$

The *hereditary Harrop pre-language* on Σ is the pair (G, D).

Definition 2.119. Let (G, D) be the hereditary Harrop pre-language on Σ, and let G' and D' be respectively the largest subsets of G and D, which contain all and only *closed* formulae. Then the pair (G', D') is the *hereditary Harrop language* on Σ.

Notation 2.120. The letters G, D and their variants G', D' are not used consistently: we may use them to denote sets of formulae or a single formula, and they may be used in the notation of a hereditary Harrop pre-language or a hereditary Harrop language. We will always make it clear what they refer to.

The two kinds of formulae in a hereditary Harrop language have an intersection, so that all and only formulae in this intersection can be used both as goals and as program clauses. The coinduction rule in CUP involves asserting the goal as a program clause, implying that goal formulae in CUP must come from this intersection, which we call *M-formulae*.

2.7.3 M-formulae and H-formulae

We introduce M-formulae and focus on a subset of it called H-formulae that corresponds immediately to Horn clauses. We will work with H-formula programs and H-formula goals. This benefits us in terms that standard Herbrand models for Horn clause programs can be modified for H-formula programs. In turn, since programs involve only H-formulae, and goals will be asserted into programs, it follows that we require goals to be H-formulae as well.

Definition 2.121. Let (G, D) be the hereditary Harrop language on Σ. The set of *M-formulae* on Σ, denoted M_Σ, is the intersection of G and D. In other words, M_Σ is the subset of all and only *closed* formulae of M, with M given by

$$M ::= A_\Sigma \mid M \wedge M \mid \forall x : \iota.\, M \mid M \to M$$

Definition 2.122. An *H-formula* is an M-formula of the form

$$\forall x_1 : \iota.\, \cdots \forall x_m : \iota.\, A_n \wedge \cdots \wedge A_1 \to A_0$$

where $A_i \in A_\Sigma$ $(0 \leq i \leq n)$.[19]

Notation 2.123. We use H_Σ to denote the set of all H-formulae on Σ.

Now we have set up H-formula as the language for CUP. It is essentially Horn clause extended with guarded terms that encode infinite trees. We will come back to it in the chapter after the next.

[19]The letter H in the name "H-formula" alludes to "Horn", since H-formulae are Horn clauses of the most commonly known form, with extension of syntax allowing binders like λ and **fix**.

Chapter 3

Productive Corecursion in Logic Programming

Corecursion refers to recursive computations that do not terminate. *Productive corecursion* refers to those corecursion that compute infinite data, i.e., arbitrarily large finite data if left uninterrupted. During his PhD studies, the author established a sufficient condition for corecursion to be productive, and identified circumstances in which the resulting infinite (circular) data from corecursion can be faithfully computed instead by an alternative algorithm. The work has been published in 2017 International Conference on Logic Programming under the same title as this chapter. Recently, he reviewed that paper, and found further technical clarifications resulting in this chapter. Compared with the paper, this chapter features a new presentation of the research, involving a systematic change of notation into more succinct forms, and this in turn facilitated more accurate statements of major theorems as well as more modularized proofs.

Section 3.1 To study computation of infinite data by non-terminating SLD resolution, we set up *structural resolution* as a finer view of SLD resolution, in which we distinguish rewriting steps, which do not produce any data, from non-rewriting steps, which do produce data. We show that SLD resolution and structural resolution are equivalent (Theorem 3.14).

Section 3.2 We introduce two properties of logic programs: *universality* and *observational productivity*. Then, using structural resolution as machinery, we show that simultaneously possessing the above two properties is a sufficient

condition for a logic program to compute infinite data when SLD resolution does not terminate (Theorem 3.33).

Section 3.3 We show that if a logic program satisfies the sufficient condition of productivity that we established above then *unification*, between certain atoms in a structural derivation can be used to determine existence of non-terminating SLD derivations, and the infinite data to be computed — if any exists (Theorem 3.54).

3.1 Structural Resolution

The main goal of this section is to recall standard SLD resolution, and define an equivalent operational semantics called structural resolution [17]. Interesting discoveries can be made when we review some familiar concept and realize that it can actually be further divided. A notable example is the invention of linear logic by Jean-Yves Girard, where the atomic notion of conjunction in classical (and intuitionistic) logic is further divided into additive conjunction and multiplicative conjunction. In a similar fashion, the atomic notion of SLD resolution is further divided into "essentially-term-rewriting" SLD resolution, and "answer-substitution-creating" SLD resolution. The latter is performed in two steps: first, create the substitution and use it to instantiate the goal, and second, do "essentially-term-rewriting" SLD resolution.

Definition 3.1. A *goal* is any $\mathbf{u} \in \mathbf{Atom}^*(\Sigma)$.

Notation 3.2. We may also use the letter G and its variants (such as G_n and G') to denote goals.

Definition 3.3. A clause $(u, \mathbf{u})$ is *variable-disjoint* from a goal $\mathbf{t}$ iff both u and $\mathbf{u}$ are variable-disjoint from $\mathbf{t}$.

Definition 3.4. A *reduction* is a pair $(\mathbf{t}, \mathbf{u}) \in \mathbf{Atom}^*(\Sigma) \times \mathbf{Atom}^*(\Sigma)$, where the goal $\mathbf{t}$ is said to be *reduced* to the goal $\mathbf{u}$.

When we say that "we remove a term t from a list $\mathbf{t} \ni t$, and then add back a list $\mathbf{u}$", we mean that if $\mathbf{t}$ is $(\mathbf{t}_1, t, \mathbf{t}_2)$, then we get $(\mathbf{t}_1, \mathbf{u}, \mathbf{t}_2)$.

Definition 3.5. A reduction $(\mathbf{t}, \mathbf{t}')$ is an *SLD resolution reduction* with respect to P iff there exists $t \in \mathbf{t}$ and there exists $(u, \mathbf{u}) \in P$ variable-disjoint from $\mathbf{t}$, $u \sim_\theta t$, and $\mathbf{t}'$ is obtained from $\theta(\mathbf{t})$ by first removing $\theta(t)$ then adding back $\theta(\mathbf{u})$.

Definition 3.6. A reduction $(\mathbf{t}, \mathbf{t}')$ is a *rewriting reduction* with respect to P iff there exists $t \in \mathbf{t}$ and there exists $(u, \mathbf{u}) \in P$ variable-disjoint from $\mathbf{t}$, $u \sqsupseteq_\theta t$, and $\mathbf{t}'$ is obtained from $\mathbf{t}$ by first removing t and then adding back $\theta(\mathbf{u})$.

Notation 3.7. We write $\mathbf{t} \to_\theta \mathbf{t}'$ iff $(\mathbf{t}, \mathbf{t}')$ is a rewriting reduction with matcher θ. We may omit the subscript θ, as in $\mathbf{t} \to \mathbf{t}'$. We write $\mathbf{t}_0 \to^n \mathbf{t}_n$ to denote $n \geq 0$ consecutive steps of rewriting reduction, starting with $\mathbf{t}_0$ and finishing with $\mathbf{t}_n$.

Definition 3.8. A reduction $(\mathbf{t}, \mathbf{t}')$ is a *substitution reduction* with respect to P iff there exists $t \in \mathbf{t}$ and there exists $(u, \mathbf{u}) \in P$ variable-disjoint from $\mathbf{t}$, $u \sim_\theta t$ (but *not* $u \sqsupseteq t$), and $\mathbf{t}'$ is $\theta(\mathbf{t})$.

Notation 3.9. We write $\mathbf{t} \hookrightarrow_\theta \mathbf{t}'$ iff $(\mathbf{t}, \mathbf{t}')$ is a substitution reduction with unifier θ. The subscript θ may be omitted, as in $\mathbf{t} \hookrightarrow \mathbf{t}'$.

Lemma 3.10. *If* $\mathbf{t} \hookrightarrow_\theta \mathbf{t}'$ *with respect to* $(u, \mathbf{u}) \in P$ *and* $t \in \mathbf{t}$, *then a) there exists* $\mathbf{t}''$, *such that* $\mathbf{t}' \to \mathbf{t}''$ *with respect to* $\theta(t) \in \mathbf{t}'$ *and a variant of* $(u, \mathbf{u})$, *and b)* $(\mathbf{t}, \mathbf{t}'')$ *is a SLD resolution reduction.*

Proof. By definition of $\hookrightarrow$, $u \sim_\theta t$[1], and $\mathbf{t}'$ is $\theta(\mathbf{t}) \ni \theta(t)$. Let $\mathbf{t}''$ be obtained from $\mathbf{t}'$ by first removing $\theta(t)$, then adding back $\theta(\mathbf{u})$. Since $u \sim_\theta t$ implies that $u \sqsupseteq_\theta \theta(t)$, we have $\mathbf{t}' \to \mathbf{t}''$ with respect to $\theta(t) \in \mathbf{t}'$ and some variant of $(u, \mathbf{u})$. Moreover, $(\mathbf{t}, \mathbf{t}'')$ satisfies the definition of a SLD resolution reduction. □

Lemma 3.10 justifies Definition 3.11.

Definition 3.11. A reduction $(\mathbf{t}, \mathbf{t}'')$ is a *structural resolution reduction* with respect to P iff a) $\mathbf{t} \to \mathbf{t}''$, or b) there exists a goal $\mathbf{t}'$ such that $\mathbf{t} \hookrightarrow_\theta \mathbf{t}'$ with respect to some $(u, \mathbf{u}) \in P$ and $t \in \mathbf{t}$, and that $\mathbf{t}' \to \mathbf{t}''$ with respect to $\theta(t) \in \mathbf{t}'$ and a variant of $(u, \mathbf{u})$.

Notation 3.12. A structural resolution reduction $(\mathbf{t}, \mathbf{t}'')$ either has the form $\mathbf{t} \to \mathbf{t}''$, or has the form $\mathbf{t} \hookrightarrow \mathbf{t}' \to \mathbf{t}''$.

[1] We also know that $u \sqsupseteq t$ does *not* hold, but this is not used in the proof.

Lemma 3.13. *If* $(\mathbf{t}, \mathbf{t}'')$ *is a SLD resolution reduction with* $u \sim_\theta t$ *(but not* $u \sqsupseteq t$*) for some* $(u, \mathbf{u}) \in P$ *and* $t \in \mathbf{t}$*, then,* $(\mathbf{t}, \mathbf{t}'')$ *is a structural resolution reduction .*

Proof. We show that there exists $\mathbf{t}'$, such that $\mathbf{t} \hookrightarrow_\theta \mathbf{t}'$ with respect to $(u, \mathbf{u}) \in P$ and $t \in \mathbf{t}$, and that $\mathbf{t}' \to \mathbf{t}''$ with respect to $\theta(t) \in \mathbf{t}'$ and a variant of $(u, \mathbf{u})$. Let $\mathbf{t}'$ be $\theta(\mathbf{t}) \ni \theta(t)$, then $\mathbf{t} \hookrightarrow_\theta \mathbf{t}'$ with respect to $(u, \mathbf{u}) \in P$ and $t \in \mathbf{t}$. Since $\mathbf{t}''$ is obtained from $\theta(\mathbf{t})$, i.e., $\mathbf{t}'$, by first removing $\theta(t)$, then adding back $\theta(\mathbf{u})$, and given the fact that $u \sqsupseteq_\theta \theta(t)$, we have $\mathbf{t}' \to \mathbf{t}''$ with respect to $\theta(t) \in \mathbf{t}'$ and a variant of $(u, \mathbf{u})$. □

Theorem 3.14. $(\mathbf{t}, \mathbf{t}'')$ *is a structural resolution reduction iff* $(\mathbf{t}, \mathbf{t}'')$ *is a SLD resolution reduction.*

Proof. ($\Rightarrow$) By case analysis on the structural resolution reduction $(\mathbf{t}, \mathbf{t}'')$. If it is rewriting, then it is also a SLD resolution reduction. If it has the form $\mathbf{t} \hookrightarrow \mathbf{t}' \to \mathbf{t}''$, then by Lemma 3.10 it is a SLD resolution reduction.

($\Leftarrow$) By case analysis on the unifier θ involved in the SLD resolution reduction. If θ is a matcher, then the SLD resolution reduction is also a rewriting reduction, hence a structural resolution reduction. If θ is not a matcher, then by Lemma 3.13, $(\mathbf{t}, \mathbf{t}'')$ is a structural resolution reduction. □

Using structural resolution instead of SLD resolution, we can reveal some secrets about productivity of logic programs.

3.2 Productivity

A logic program is called *productive* iff it computes infinite trees. Infinite trees result from the accumulation of substitutions computed during the infinite SLD derivation. Now that we are looking at SLD resolution in light of structural resolution, we realize that it is the non-rewriting steps in an SLD derivation that are critical for productivity, since rewriting steps produce no substitution for variables in the goal. Therefore, for a program to be productive its infinite SLD derivation must satisfy two conditions: first, there must be an infinite number of non-rewriting steps, and second, substitutions produced therein can accumulate.

The first condition is met if we can show that rewriting always terminates for the program. Because if so, any infinite SLD derivation cannot be an infinite sequence of rewriting steps, implying that it is interspersed by an infinite amount of non-rewriting steps. Termination for rewriting is called *observational productivity.* The

second condition is met if Horn clauses in the program do not have existential variables, i.e., Horn clauses are *universal*.

Definition 3.15. A structural (SLD) *derivation* with respect to a logic program P on Σ, is a possibly infinite sequence of goals on Σ: $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3, \ldots$, such that, for all $n \geq 1$, $(\mathbf{t}_n, \mathbf{t}_{n+1})$ is a structural (resp. SLD) resolution reduction with respect to P, and the clause used to reduce the goal $\mathbf{t}_n$ is variable-disjoint, from all earlier goals $\mathbf{t}_1, \ldots, \mathbf{t}_{n-1}$ and all clauses that are used to reduce these earlier goals.

Definition 3.16. A logic program P (on Σ) is *observationally productive* iff there does not exist an infinite sequence of goals on Σ: $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3, \ldots$, such that for all $n \geq 1$, $\mathbf{t}_n \to \mathbf{t}_{n+1}$ with respect to P.

A decision algorithm for observational productivity is given by [18].

Proposition 3.17. *Every infinite structural derivation, with respect to an* observationally productive *logic program, has the form:*

$$G_0 \to^{n_0} G'_0 \hookrightarrow G_1 \to^{n_1} G'_1 \hookrightarrow G_2 \cdots$$

Proof. Observational productivity guarantees termination of all $\to$ steps. Since the structural derivation is infinite, wherever $\to$ terminates, it must be followed by a step of $\hookrightarrow$, which necessarily initiates a new (and finite) sequence of $\to$, and so forth. Hence the pattern $G \to^n G' \hookrightarrow G''$ where $n \geq 0$. □

Proposition 3.17 justifies Definition 3.18.

Definition 3.18. Let

$$G_0 \to^{n_0} G'_0 \hookrightarrow G_1 \to^{n_1} G'_1 \hookrightarrow G_2 \cdots$$

be an infinite structural derivation with respect to an observationally productive logic program. The *unifiers* of this derivation is the sequence $\theta_m (m \geq 1)$ where $G_m = \theta_m(G'_{m-1})$.

Definition 3.19. An infinite atom $t \in \mathbf{Atom}^\infty(\Sigma)$ is *SLD (structurally) computable at infinity*[2] with respect to an atomic goal $u \in \mathbf{Atom}(\Sigma)$ and a logic program P

[2] An atom that is SLD computable at infinity was defined to be ground, and the derivation was required to be fair, by [13, p.189]. We redefine this notion to include atoms that may have variables, and allow the derivation to be not fair. By the way, an infinite (structural or SLD) derivation is *fair* iff for all goal $\mathbf{t}$ of the derivation, and for all $t \in \mathbf{t}$, either $\mathbf{t}$ is reduced with respect to t, or there exists a subsequent goal $\mathbf{t}' \ni \theta(t)$ for some θ, and $\mathbf{t}'$ is reduced with respect to $\theta(t)$.

iff there exists an infinite SLD (resp. structural) derivation starting with u, with unifiers θ_j ($j \geq 1$), and

$$\lim_{k\to\infty} d(t, \theta_k \cdots \theta_1(u)) = 0.$$

Proposition 3.20. *An atom $t \in \boldsymbol{Atom}^{\infty}(\Sigma)$ is SLD computable at infinity iff it is structurally computable at infinity.*

Proof. By Theorem 3.14, the *SLD derivation* that computes towards t is also a *structural derivation*, and vice versa. □

Definition 3.21 (n-t)**.** Let $\vec{\theta}$ be either a single substitution or a composition of substitutions. $\vec{\theta}$ is *non-trivial* (n-t for short) for $\mathbf{u}$, iff there exists some $x \in FV(\mathbf{u})$, such that $\vec{\theta}(x)$ is not a variable.

Lemma 3.22. *If $FV(\mathbf{t}) \supseteq FV(\mathbf{u})$, and θ is n-t for $\mathbf{u}$, then θ is n-t for $\mathbf{t}$.*

Proof. By definition of n-t. Since θ is n-t for $\mathbf{u}$, there exists some $x \in FV(\mathbf{u})$ such that $\theta(x)$ is not a variable. Since $FV(\mathbf{t}) \supseteq FV(\mathbf{u})$, that particular x must also be in $FV(\mathbf{t})$. □

There is a stronger sense of non-triviality for compositions of substitutions. For example, the composition $\theta_3\theta_2\theta_1$, which we suppose to be n-t for $\mathbf{u}$, could additionally satisfy that: θ_1 is n-t for $\mathbf{u}$, θ_2 is n-t for $\theta_1(\mathbf{u})$, and θ_3 is n-t for $\theta_2(\theta_1(\mathbf{u}))$.

Definition 3.23 (s-n-t)**.** A composition $\theta_k \cdots \theta_1$ of substitutions is *strongly non-trivial* (s-n-t) for $\mathbf{u}$, iff θ_1 is n-t for $\mathbf{u}$, and for all $1 < j \leq k$, θ_j is n-t for $\theta_{j-1} \cdots \theta_1(\mathbf{u})$.

Proposition 3.24. *Strong non-triviality implies non-triviality, but not conversely.*

Proof. 1) We show that s-n-t implies n-t. Suppose the composition $\theta_k \cdots \theta_1$ is s-n-t for some $\mathbf{u}$. Then, by definition of s-n-t, θ_1 is n-t for $\mathbf{u}$, so there exists $y \in FV(\mathbf{u})$, such that $\theta_1(y)$ is not a variable. Then $\theta_k \cdots \theta_1(y)$ cannot be a variable as well. Therefore, $\theta_k \cdots \theta_1$ is n-t for $\mathbf{u}$.

2) We show that n-t does not imply s-n-t. Let θ_1 be $x \mapsto f(y)$, and θ_2 be $y \mapsto z$. Then the composition $\theta_2\theta_1$ is n-t for the term x, but is not s-n-t for x, since θ_2 is not n-t for $\theta_1(x)$. □

Lemma 3.25. *If $FV(\mathbf{t}) \supseteq FV(\mathbf{u})$, then for all θ, $FV(\theta(\mathbf{t})) \supseteq FV(\theta(\mathbf{u}))$.*

Proof. Suppose

$$FV(\mathbf{u}) = \{x_1, \ldots, x_n\}$$

$$FV(\mathbf{t}) = FV(\mathbf{u}) \cup \{y_1, \ldots, y_m\}$$

Observe that

$$FV(\theta(\mathbf{u})) = \bigcup_{j=1}^{n} FV(\theta(x_j))$$

$$FV(\theta(\mathbf{t})) = FV(\theta(\mathbf{u})) \cup \left(\bigcup_{k=1}^{m} FV(\theta(y_k)) \right)$$

The argument is similar when $FV(\mathbf{u})$ is an infinite set. □

Definition 3.26. A clause $(u, \mathbf{u})$ is *universal* iff $FV(u) \supseteq FV(\mathbf{u})$. A logic program P is *universal* iff all clauses in P are universal.

Lemma 3.27. *If $\mathbf{t} \to \mathbf{u}$ with respect to a universal logic program, then $FV(\mathbf{t}) \supseteq FV(\mathbf{u})$.*

Proof. Suppose we use the universal clause $(t', \mathbf{t}')$ and choose $t \in \mathbf{t}$, such that $t' \sqsupseteq_\sigma t$.

Our universality assumption entails that $FV(t') \supseteq FV(\mathbf{t}')$ (1). Then, using Lemma 3.25, from (1) we have $FV(\sigma(t')) \supseteq FV(\sigma(\mathbf{t}'))$ (2). Since $\sigma(t')$ is t, we rewrite (2) into $FV(t) \supseteq FV(\sigma(\mathbf{t}'))$ (3). We display $\mathbf{t}$ and $\mathbf{u}$ as follows:

$$\mathbf{t} : \mathbf{t_1}, \quad t \quad, \mathbf{t_2}$$

$$\mathbf{u} : \mathbf{t_1}, \sigma(\mathbf{t}'), \mathbf{t_2}$$

Given (3), we can see from above that $FV(\mathbf{t}) \supseteq FV(\mathbf{u})$. □

Corollary 3.28. *For all $n > 0$, if $\mathbf{t} \to^n \mathbf{u}$ with respect to a universal logic program, then $FV(\mathbf{t}) \supseteq FV(\mathbf{u})$.*

Proof. By Lemma 3.27 and transitivity of $\supseteq$. □

Lemma 3.29. *If $FV(\mathbf{t}) \supseteq FV(\mathbf{u})$, and the composition $\vec{\theta}$ is s-n-t for $\mathbf{u}$, then $\vec{\theta}$ is s-n-t for $\mathbf{t}$.*

Proof. Expand $\vec{\theta}$ as $\theta_k \cdots \theta_1$. By definition of s-n-t, we have

$$\theta_1 \text{ is n-t for } \mathbf{u}$$

$$\text{For } 1 < j \leq k,\ \theta_j \text{ is n-t for } \theta_{j-1}(\cdots \theta_1(\mathbf{u}) \cdots)$$

Corresponding to each line above, using Lemmas 3.22 and 3.25, we have:

θ_1 is n-t for $\mathbf{t}$ by Lem. 3.22

For $1 < j \leq k$, θ_j is n-t for $\theta_{j-1}(\cdots\theta_1(\mathbf{t})\cdots)$ by Lem. 3.25, 3.22

So $\vec{\theta}$ is s-n-t for $\mathbf{t}$. □

Proposition 3.30. *Let $\theta_j (j \geq 1)$ be the unifiers of an infinite structural derivation*

$$G_0 \to^{n_0} G'_0 \hookrightarrow G_1 \to^{n_1} G'_1 \hookrightarrow G_2 \cdots$$

with respect to a logic program that is both universal and observationally productive. Then, for all $k > 1$, and for all $1 \leq r < k$, the composition $\theta_k \cdots \theta_r$ is s-n-t for G_{r-1}.

Proof. By induction on r, with base case $r = k-1$, and inductive step:

"If $\theta_k \cdots \theta_r$ is s-n-t for G_{r-1}, then $\theta_k \cdots \theta_r \theta_{r-1}$ is s-n-t for G_{r-2}."

We first show the base case that $\theta_k \theta_{k-1}$ is s-n-t for G_{k-2}. Observe the fragment:

$$G_{k-2} \to^{n} G'_{k-2} \hookrightarrow G_{k-1} \to^{n'} G'_{k-1} \hookrightarrow G_k$$

By definition of $\hookrightarrow$, θ_k is n-t for G'_{k-1}. By Corollary 3.28, $FV(G_{k-1}) \supseteq FV(G'_{k-1})$. Then by Lemma 3.22, we have: θ_k is n-t for G_{k-1} (Fact A).

Note that $G_{k-1} = \theta_{k-1}(G'_{k-2})$, we can rewrite (Fact A) as: θ_k is n-t for $\theta_{k-1}(G'_{k-2})$ (Fact A'). Moreover, and again by definition of $\hookrightarrow$, θ_{k-1} is n-t for G'_{k-2} (Fact B). Combining Facts A' and B, we conclude that $\theta_k\theta_{k-1}$ is s-n-t for G'_{k-2}. Using Lemma 3.29, we conclude that $\theta_k\theta_{k-1}$ is s-n-t for G_{k-2}, completing the base case, because by Corollary 3.28, $FV(G_{k-2}) \supseteq FV(G'_{k-2})$.

Next we make the inductive step. Observe the fragment

$$G_{r-2} \to^{n} G'_{r-2} \hookrightarrow G_{r-1}$$

We have that $G_{r-1} = \theta_{r-1}(G'_{r-2})$, so the inductive hypothesis can be rewritten as: $\theta_k \cdots \theta_r$ is s-n-t for $\theta_{r-1}(G'_{r-2})$ (IH). Now we have that θ_{r-1} is n-t for G'_{r-2}, which, together with (IH), allows us to derive that: $\theta_k \cdots \theta_r \theta_{r-1}$ is s-n-t for G'_{r-2}. We then use Lemma 3.29 to establish that $\theta_k \cdots \theta_r \theta_{r-1}$ is s-n-t for G_{r-2}, thus completing the proof. □

Corollary 3.31. *Let $\theta_j (j \geq 1)$ be the unifiers of an infinite structural derivation with respect to an initial goal G_0 and a logic program that is both universal and observationally productive. Then, for all $k > 1$, the composition $\theta_k \cdots \theta_1$ is s-n-t for G_0.*

Proof. Set $r = 1$ in Proposition 3.30. $\square$

Lemma 3.32. *Let $\theta_j (j \geq 1)$ be the unifiers of an infinite structural derivation with respect to an initial goal u and a logic program that is both universal and observationally productive. Then, for all $n \in \mathbb{N}$, there exists $k_n > 1$, such that for all $k > k_n$, $\theta_k \cdots \theta_1(u)|_n$ is identical to $\theta_{k_n} \cdots \theta_1(u)|_n$.*

Proof. By contradiction. Assume the negation of our proposition, which says there exists $n \in \mathbb{N}$, such that for all $k > 1$, there exists some $k' > k$, such that $\theta_{k'} \cdots \theta_1(u)|_n$ is different from $\theta_k \cdots \theta_1(u)|_n$.

However, observe that:

- Corollary 3.31 indicates that for all $j > 1$, the composition $\theta_j \cdots \theta_1$ is s-n-t for u.

- The definitions of structural reduction and derivation entail that every Horn clause, when being used to resolve a goal in a derivation, is variable-disjoint from the current goal and all earlier goals in the derivation.

- Furthermore, we assume that unifiers do not rename variables in goals, i.e., for all $G \hookrightarrow_\theta G'$ steps in the derivation, if $x \in FV(G)$, then either $\theta(x)$ is not a variable, or $\theta(x)$ is x itself. It is possible to configure unification algorithms to work in this way, since whenever a variable x in a goal is renamed to a variable y of a Horn clause, we can instead rename y into x.

The points above imply that the only way in which $\theta_{k'} \cdots \theta_1(u)|_n$ becomes different from $\theta_k \cdots \theta_1(u)|_n$, is that some variables in $\theta_k \cdots \theta_1(u)|_n$ are instantiated into non-variables while applying $\theta_{k+1}, \ldots, \theta_{k'}$, and no variable is renamed in this process.

Instantiation of variables into non-variables cannot happen infinitely often within the truncation at level n, since no finitely branching tree can accommodate an infinite amount of variables up to any fixed level. So the negation of our proposition leads to absurdity. $\square$

Theorem 3.33. *Let $\theta_j (j \geq 1)$ be the unifiers of an infinite structural derivation with respect to an initial goal u and a logic program P that is both universal and observationally productive. Then, there exists $t \in \boldsymbol{Atom}^\infty(\Sigma)$ structurally computable at infinity with respect to u and P.*

Proof. We build t inductively, for each level n of t. At level $n = 0$, we let $t(\epsilon)$ be $u(\epsilon)$. Now assume t is defined up till level $n \geq 0$. By Lemma 3.32, there is some k_{n+2} such that for all $k > k_{n+2}$, $\theta_k \ldots \theta_1(u)|_{n+2}$ is identical to $\theta_{k_{n+2}} \cdots \theta_1(u)|_{n+2}$. We define the nodes at level $n + 1$ for t in the same way as nodes at level $n + 1$ of $\theta_{k_{n+2}} \cdots \theta_1(u)$.

With t defined as above, and using Lemma 3.32, we have that, for all $n \in \mathbb{N}$, there exists $k_n > 1$, such that for all $k > k_n$, $d(t, \theta_k \cdots \theta_1(u)) \leq 2^{-n}$. This implies that $d(t, \theta_k \cdots \theta_1(u)) \to 0$ as $k \to \infty$. □

Corollary 3.34. *Let $\theta_j (j \geq 1)$ be the unifiers of an infinite SLD derivation with respect to an initial goal u and a logic program that is both universal and observationally productive. Then, there exists $t \in \boldsymbol{Atom}^\infty(\Sigma)$ SLD computable at infinity.*

Corollary 3.35. *Let P be a logic program that is both universal and observationally productive. Goal u has an infinite structural derivation with respect to P iff there exists $t \in \boldsymbol{Atom}^\infty(\Sigma)$ SLD computable at infinity with respect to u and P.*

After knowing productivity conditions for logic programs, we may make use of unification (without occur-check) to get the same infinite tree as computed by infinite derivation. Such unification happens between one goal in the derivation with one of its ancestor goals. This approach is called *loop detection*.

3.3 Loop Detection

We detect loops in proof trees. A proof tree [19, §1.6 p.21] is not a search tree [19, §1.5 p.18]. Nodes in a proof tree are atomic formulae, and for each node t with children $\mathbf{t}$, the Horn clause $(t, \mathbf{t})$ is a substitution instance of some clause from a given program. For a node t in a proof tree, its *ancestor-set* consists of all atoms on the path from t to the root of the tree.

Every atom in a goal that is from an SLD (or structural) derivation can be annotated with its ancestor set. For instance, after annotation, a goal $t_1, \ldots, t_n$ becomes $(t_1, S_1), \ldots, (t_n, S_n)$ that is a list of pairs. For the sake of a more compact notation, we can zip this list of pairs into a pair of lists, as $(\mathbf{t}, \mathbf{S})$, where $\mathbf{t}$ stands for $t_1, \ldots, t_n$ while $\mathbf{S}$ stands for $S_1, \ldots, S_n$.

Definition 3.36. An *ancestor-set* is a finite subset of $\mathbf{Atom}(\Sigma)$, associated to an atom t in a goal, recording atoms from previous goals which are logically depending on t.

Definition 3.37. An *annotated goal* is a pair $(\mathbf{t}, \mathbf{S})$, where $\mathbf{t}$ is a goal, and $\mathbf{S}$ is a finite list of ancestor-sets, of the same length as $\mathbf{t}$, such that the i-th member of $\mathbf{S}$ is the ancestor-set of the i-th member of $\mathbf{t}$.

Notation 3.38. Given an annotated goal $(\mathbf{t}, \mathbf{S})$, we use t_i (S_i) to denote the i-th member of $\mathbf{t}$ (resp. $\mathbf{S}$), so S_i is the ancestor-set of t_i. We use $\boldsymbol{\emptyset}$ to denote a list $\emptyset, \ldots, \emptyset$ of empty ancestor-sets in the annotated goal $(\mathbf{t}, \boldsymbol{\emptyset})$.

Definition 3.39. An *annotated reduction* is a pair $((\mathbf{t}, \mathbf{S}), (\mathbf{t}', \mathbf{S}'))$ of annotated goals, and we say $(\mathbf{t}, \mathbf{S})$ is *reduced* to $(\mathbf{t}', \mathbf{S}')$.

Given an annotated goal $(\mathbf{t}, \mathbf{S})$ where $\mathbf{t}$ has the form $(\mathbf{t}_1, t_i, \mathbf{t}_2)$, and $\mathbf{S}$ has the form $(\mathbf{S}_1, S_i, \mathbf{S}_2)$, if $\mathbf{t} \rightarrow \mathbf{t}'$ with respect to some clause $(u, \mathbf{u})$ such that $u \sqsupseteq_\theta t_i$, we know that $\mathbf{t}'$ has the form $(\mathbf{t}_1, \theta(\mathbf{u}), \mathbf{t}_2)$. The ancestor-set for each atom in $\mathbf{t}'$ is as follows: ancestor-sets of atoms in sub-lists $\mathbf{t}_1, \mathbf{t}_2$ are not changed, while every atom in $\theta(\mathbf{u})$ has $S'_i = S_i \cup \{t_i\}$ as its ancestor-set. In other words, the list $\mathbf{S}'$ of ancestor-sets, that corresponds to the goal $\mathbf{t}'$, has the form $(\mathbf{S}_1, \mathbf{S}_3, \mathbf{S}_2)$ where the length of $\mathbf{S}_3$ equals the length of $\theta(\mathbf{u})$, and every member of $\mathbf{S}_3$ is S'_i.

Definition 3.40. An annotated reduction $((\mathbf{t}, \mathbf{S}), (\mathbf{t}', \mathbf{S}'))$ is an *annotated rewriting reduction* with respect to a logic program P, iff $\mathbf{t} \rightarrow \mathbf{t}'$ with respect to some $(u, \mathbf{u}) \in P$ and some $t_i \in \mathbf{t}$ such that $u \sqsupseteq_\theta t_i$, and, $\mathbf{S}'$ is obtained from $\mathbf{S}$ by first removing S_i, then adding back a list of n copies of S'_i, where n is the length of $\theta(\mathbf{u})$, and S'_i is $S_i \cup \{t_i\}$.

Notation 3.41. Let $\mathbf{S}$ be a list of ancestor-sets, i.e., $\mathbf{S}$ has the form $S_1, \ldots, S_n$ where each ancestor-set S_i has the form $\{t_1, \ldots, t_{m_i}\}$. The result of applying θ to all atoms in $\mathbf{S}$, denoted $\theta(\mathbf{S})$, has the form $\theta(S_1), \ldots, \theta(S_n)$ where each $\theta(S_i)$ has the form $\{\theta(t_1), \ldots, \theta(t_{m_i})\}$.

Definition 3.42. An annotated reduction $((\mathbf{t}, \mathbf{S}), (\mathbf{t}', \mathbf{S}'))$ is an *annotated substitution reduction* with respect to a logic program P, iff $\mathbf{t} \hookrightarrow_\theta \mathbf{t}'$ with respect to P, and, $\mathbf{S}'$ is $\theta(\mathbf{S})$.

Definition 3.43. An annotated reduction $((\mathbf{t}, \mathbf{S}), (\mathbf{t}', \mathbf{S}'))$ is an *annotated structural resolution reduction* with respect to a logic program P, iff $(\mathbf{t}, \mathbf{t}')$ is a structural resolution reduction with respect to some $(u, \mathbf{u}) \in P$ and some $t_i \in \mathbf{t}$, and, given $u \sim_\theta t_i$, $\mathbf{S}'$ is obtained from $\theta(\mathbf{S})$ by first removing $\theta(S_i)$, then adding back a list of n copies of $\theta(S'_i)$, where n is the length of $\theta(\mathbf{u})$, and S'_i is $S_i \cup \{t_i\}$.

Definition 3.44. A possibly infinite sequence $(\mathbf{t}_n, \mathbf{S}_n)$ $(n \geq 1)$ of annotated goals is an *annotated structural derivation* iff the sequence $\mathbf{t}_n$ $(n \geq 1)$ of goals is a structural derivation and all the ancestor-sets $\mathbf{S}_n$ $(n \geq 1)$ are so constructed that every pair of adjacent annotated goals constitutes an annotated structural resolution reduction, with $\mathbf{S}_1 = \boldsymbol{\emptyset}$.

Definition 3.45. A renaming substitution θ is *fresh* for $\mathbf{t}$ iff $\theta(\mathbf{t})$ is variable-disjoint from $\mathbf{t}$.

Definition 3.46. An infinite sequence γ_n $(n \geq 0)$ of renaming substitutions is *cumulatively fresh* for $\mathbf{t}$ iff:

- γ_0 is the identity substitution.
- For all $m \geq 1$, γ_m is fresh for $\gamma_{m-1} \cdots \gamma_0(\mathbf{t})$.[3]
- For all m and n (both ≥ 0), if $m \neq n$ then $\gamma_m \cdots \gamma_0(\mathbf{t})$ is variable-disjoint from $\gamma_n \cdots \gamma_0(\mathbf{t})$.[4]

Lemma 3.47. *If a sequence γ_n $(n \geq 0)$ of renaming substitutions is cumulatively fresh for $\mathbf{u}$, and $FV(\mathbf{u}) \supseteq FV(\mathbf{t})$, then γ_n $(n \geq 0)$ is also cumulatively fresh for $\mathbf{t}$.*

Proof. First consider a special case where $FV(\mathbf{u})$ is $\{x, y, z\}$ and $FV(\mathbf{t})$ is $\{y, z\}$. Suppose the effect of cumulative freshness of γ_n $(n \geq 0)$ on $\mathbf{u}$ is given by:

$$\begin{array}{ccccccc}
 & \gamma_1 & & \gamma_2 & & \cdots \\
x & \mapsto & x_1 & \mapsto & x_2 & \cdots \\
\hline
y & \mapsto & y_1 & \mapsto & y_2 & \cdots \\
z & \mapsto & z_1 & \mapsto & z_2 & \cdots
\end{array}$$

[3]In other words, γ_1 is fresh for $\gamma_0(\mathbf{t})$. γ_2 is fresh for $\gamma_1\gamma_0(\mathbf{t})$. γ_3 is fresh for $\gamma_2\gamma_1\gamma_0(\mathbf{t})$. And so on.

[4]In other words, members of the infinite list $\gamma_0(\mathbf{t})$, $\gamma_1\gamma_0(\mathbf{t})$, $\gamma_2\gamma_1\gamma_0(\mathbf{t})$, ... are pairwise variable-disjoint.

We can see the effect of cumulative freshness of γ_n $(n \geq 0)$ on $\mathbf{t}$.

In all cases other than the special one considered above, the reasoning is similar, and leads to the same conclusion. □

Notation 3.48. If we use $\{\mathbf{y} \mapsto \mathbf{u}\}$ to abbreviate a (pre-)substitution expression $\{y_1 \mapsto u_1, \ldots, y_n \mapsto u_n\}$ then $\mathbf{y}$ and $\mathbf{u}$ respectively denote the lists $y_1, \ldots, y_n$ and $u_1, \ldots, u_n$. Moreover, $\{\vec{\theta_1}(\mathbf{y}) \mapsto \vec{\theta_2}(\mathbf{u})\}$, where $\vec{\theta_1}, \vec{\theta_2}$ are (compositions of) renaming substitutions, abbreviates $\{\vec{\theta_1}(y_1) \mapsto \vec{\theta_2}(u_1), \ldots, \vec{\theta_1}(y_n) \mapsto \vec{\theta_2}(u_n)\}$.

Definition 3.49. A circular pre-substitution σ is *grounding* iff σ has the form $\{y_1 \mapsto u_1, \ldots, y_n \mapsto u_n\}$, abbreviated $\{\mathbf{y} \mapsto \mathbf{u}\}$, such that $FV(\mathbf{u}) = FV(\mathbf{y})$[5] and for all $u \in \mathbf{u}$, $u \notin \mathit{Var}$.

Definition 3.50. Let σ, of the form $\{\mathbf{y} \mapsto \mathbf{u}\}$, be grounding and let γ_n $(n \geq 0)$ be cumulatively fresh for $\mathbf{u}$. The *unfolding* of σ with respect to γ_n $(n \geq 0)$, is an infinite sequence α_n $(n \geq 0)$ of substitutions, where α_n is $\{\gamma_n \cdots \gamma_0(\mathbf{y}) \mapsto \gamma_{n+1} \cdots \gamma_0(\mathbf{u})\}$.

Proposition 3.51. *Assume a grounding circular pre-substitution σ of the form $\{\mathbf{y} \mapsto \mathbf{u}\}$, and a term $t \in \boldsymbol{Term}(\Sigma)$ such that $FV(t) = FV(\mathbf{u})$. Let α_n $(n \geq 0)$ and β_n $(n \geq 0)$ be any two unfolding*[6] *of σ. Then, there is a unique $t' \in \boldsymbol{GTerm}^{\infty}(\Sigma)$, such that*

$$\lim_{n \to \infty} d(t', \alpha_n \cdots \alpha_0(t)) = 0 \qquad \lim_{n \to \infty} d(t', \beta_n \cdots \beta_0(t)) = 0$$

Proof. Obvious. □

Proposition 3.51 justifies Definition 3.52.

Definition 3.52. Assume a grounding circular pre-substitution σ of the form $\{\mathbf{y} \mapsto \mathbf{u}\}$, and a term $t \in \mathbf{Term}(\Sigma)$ such that $FV(t) = FV(\mathbf{u})$. Let α_n $(n \geq 0)$ be any unfolding of σ. Then, the *application* of σ to t results in the $t' \in \mathbf{GTerm}^{\infty}(\Sigma)$ such that $d(t', \alpha_n \cdots \alpha_0(t)) \to 0$ as $n \to \infty$.

Notation 3.53. We use $\sigma(t)$ to denote the infinite term t' that results from applying a grounding circular pre-substitution σ to a term t.

[5] In other words, $FV(\mathbf{u}) = \{y_1, \ldots, y_n\}$

[6] For example, both $\{x \mapsto s(x_1)\}, \{x_1 \mapsto s(x_2)\}, \ldots$ and $\{x \mapsto s(y_1)\}, \{y_1 \mapsto s(y_2)\}, \ldots$ are unfolding of $\{x \mapsto s(x)\}$.

Theorem 3.54. *Let $(\mathbf{t}, \mathbf{S})$ be an annotated goal from an annotated structural derivation with respect to some* universal and observationally productive *logic program P. Let $\mathbf{t} \ni t$, $\mathbf{S} \ni S$, and S be the ancestor-set of t. Furthermore, let S has the form $\{u_1, \ldots, u_n\}$ where $n \geq 1$ and without loss of generality assume that for all $u_k \in S$ $(1 \leq k < n)$, u_k is added to S later than*[7] *u_{k+1} in the course of construction of S. The following statements hold.*

1. *$FV(u_k) \supseteq FV(u_j)$ for $1 \leq j < k \leq n$. $FV(u_1) \supseteq FV(t)$.*[8]
2. *If there exists some $u_i \in S$ such that $t \approx_\sigma u_i$ with pre-substitution σ, then*
   (a) *σ is circular, and,*
   (b) *if u'_i is a variant of u_i, and u'_i is variable-disjoint from t, and $t \sqsupseteq u'_i$, then*
      i. *there exists an infinite structural derivation starting with the goal $\mathbf{t}$, and,*
      ii. *if σ is grounding, then $\sigma(t)$ is structurally computable at infinity with respect to P and t.*

Proof. We display a clause $(t, \mathbf{t})$ in the form $t \Leftarrow \mathbf{t}$, and write $t \Leftarrow \ldots\ t_i\ \ldots$ to highlight $t_i \in \mathbf{t}$. Observe that there exist some clauses in P, whose instances are:

$$u_n \Leftarrow \ldots\ u_{n-1}\ \ldots$$
$$\ldots$$
$$u_2 \Leftarrow \ldots\ u_1\ \ldots$$
$$u_1 \Leftarrow \ldots\ t\ \ldots$$

Therefore, we have $u_{k+1} \to \ldots\ u_k\ \ldots$ for $1 \leq k < n$, and $u_1 \to \ldots\ t\ \ldots$. Now conclusion 1 follows using Lemma 3.27.

If σ is not circular, then by Proposition 2.42, σ is a substitution. This, together with $t \approx_\sigma u_i$, implies that $t \sim_\sigma u_i$. Now we have the following set of instances of clauses of P:

$$\sigma(u_i) \Leftarrow \ldots\ \sigma(u_{i-1})\ \ldots$$
$$\ldots$$
$$\sigma(u_2) \Leftarrow \ldots\ \sigma(u_1)\ \ldots$$
$$\sigma(u_1) \Leftarrow \ldots\ \sigma(t)\ \ldots$$

[7] In other words, u_k is a child of u_{k+1} in the corresponding proof tree.

[8] This is a minor conclusion used as a local lemma. Conclusion 2 is our major observation.

where $\sigma(t)$ is identical to $\sigma(u_i)$ because of $t \sim_\sigma u_i$. This gives a non-terminating sequence of rewriting reduction steps, whose repeating pattern is:

$$\begin{array}{rl} & \boxed{\ldots\ \sigma(u_i)\ \ldots} \\ \rightarrow & \ldots\ \sigma(u_{i-1})\ \ldots \\ \rightarrow & \cdots \\ \rightarrow & \ldots\ \sigma(u_2)\ \ldots \\ \rightarrow & \ldots\ \sigma(u_1)\ \ldots \\ \rightarrow & \ldots\ \sigma(t)\ \ldots \\ \text{i.e.} & \boxed{\ldots\ \sigma(u_i)\ \ldots} \end{array}$$

This breaks the *observational productivity* condition. Therefore σ is circular. Now we have proved conclusion 2a by contradiction.

In order to construct the infinite structural derivation starting with **t**, we shall need an infinite sequence γ_n $(n \geq 0)$ of renaming substitutions that is cumulatively fresh for u_i. Then, by conclusion 1 and Lemma 3.47, the sequence γ_n $(n \geq 0)$ is also cumulatively fresh for t and all u_j $(j < i)$.

- Since $t \sqsupseteq u'_i$, we have that for all $n \geq 0$, $\gamma_n \cdots \gamma_0(t) \sqsupseteq_{\sigma_{n+1}} \gamma_{n+1} \cdots \gamma_0(u_i)$ with matcher σ_{n+1}. In other words, for all $n \geq 0$, $\sigma_{n+1}\gamma_n \ldots \gamma_0(t)$ is identical to $\gamma_{n+1} \ldots \gamma_0(u_i)$.

- There is a set of program clause instances for every $n \geq 0$:

$$\begin{array}{lcl} \gamma_{n+1} \ldots \gamma_0(u_i) & \Leftarrow & \ldots \quad \gamma_{n+1} \ldots \gamma_0(u_{i-1}) \quad \ldots \\ \ldots & & \\ \gamma_{n+1} \ldots \gamma_0(u_2) & \Leftarrow & \ldots \quad \gamma_{n+1} \ldots \gamma_0(u_1) \quad \ldots \\ \gamma_{n+1} \ldots \gamma_0(u_1) & \Leftarrow & \ldots \quad \gamma_{n+1} \ldots \gamma_0(t) \quad \ldots \end{array}$$

Because of the two facts above, we know that, for all $n \geq 0$, there exists a repeating structural derivation pattern[9]:

$$[\ldots\ \gamma_n \cdots \gamma_0(t)\ \ldots] \hookrightarrow [\ldots\ \sigma_{n+1}\gamma_n \cdots \gamma_0(t)\ \ldots] \rightarrow^i [\ldots\ \gamma_{n+1} \cdots \gamma_0(t)\ \ldots]$$

This pattern dictates that the infinite structural derivation starting from goal **t** has the form

[9] We shall enclose goals between square brackets, to help them be more visually distinguishable.

$[\dots\ \gamma_0(t)\ \dots] \hookrightarrow [\dots\ \sigma_1\gamma_0(t)\ \dots] \to^i$
$[\dots\ \gamma_1\gamma_0(t)\ \dots] \hookrightarrow [\dots\ \sigma_2\gamma_1\gamma_0(t)\ \dots] \to^i$
$[\dots\ \gamma_2\gamma_1\gamma_0(t)\ \dots] \hookrightarrow \quad \cdots$

Now we have proved conclusion 2(b)i. If we restrict the initial goal to t (i.e., $\gamma_0(t)$) then using Theorem 3.33 the above infinite structural derivation computes an infinite term t' at infinity, characterized by

$$\lim_{n\to\infty} d(t', \sigma_n \cdots \sigma_1(t)) = 0$$

If σ is grounding then the sequence α_n $(n \geq 0)$ where $\alpha_n = \sigma_{n+1}$, is an unfolding of σ. Then $\sigma(t)$ and t' coincide. Hence conclusion 2(b)ii. □

The theorem is revised and is not the same as its counterpart in the published paper, because a counterexample was discovered during the course of reviewing the results and adding them to this thesis. Here we add the extra condition that the circular unifier is *grounding* in order for it to capture the infinite tree. Otherwise, we have the clause

$$h(s(x, y), g(z)) \Leftarrow h(x, y)$$

where only x, y, z are variables. Given a goal $h(x, y)$, the answer given by the circular unifier is not exactly the same as that by the infinite derivation, since the latter involves an infinite amount of distinct free variables while in the former the number of free variables is finite.

All the work in this section is built upon Gupta et al's work on CoLP [1], where they showed coinductive soundness of the system and explored its application. Moving the knowledge one step further, the author explains why CoLP can in some cases return the same answer as the perpetual computation. It is because of the definition that the result of applying a (grounding circular) unifier is the same as applying the unfolding of the unifier, and the observation that the unfolding is also given by the perpetual computation.

So far we as well as the coinductive logic programming research community have been working only with regular trees and cyclic derivations. Next we will move beyond this limitation and see a coinductively sound logic for Horn clauses that can prove some propositions involving irregular trees and non-cyclic derivations.

Chapter 4

Coinductive Uniform Proof

The author was deeply involved in the development of a coinductively sound and constructive logical framework, called *Coinductive Uniform Proof* (CUP) [20], for coinduction involving first-order Horn clauses and (possibly irregular) infinite trees. This framework reveals the common logical pattern that underlies the existing coinductive reasoning schemes (i.e., [1,2]), and shows the boundary of coinductive reasoning power of certain formal languages (such as first-order Horn clause and first-order hereditary Harrop formula), and promises new coinductive algorithms that can go beyond the functionality of the existing ones.

There are three versions of CUP that appeared naturally as the concept was created and revised.

In the summer of 2017, Dr Komendantskaya handed the author a sketch of the CUP system that embodies her idea of extending Uniform Proof [15] with a rule that asserts the goal as a new program clause. The author then designed the details of this system, including a guarding mechanism to safeguard coinductive soundness, and proved that the design is indeed sound. This is the version one of CUP, and was published in the abstract[1] submitted to the 2018 workshop Programming And Reasoning on Infinite Structures (PARIS).

Later on, Dr Basold joined the team. He revised the guarding mechanism from the author's cumbersome version to an equivalent but succinct one, while on the other hand he complicated the notion of fixed-point terms[2] in order to support a

[1] Available at `http://www.macs.hw.ac.uk/~yl55/Publication/WorkshopPB/KL4PARIS.pdf`

[2] A *fxied-point term* is a finite representation of a (possibly irregular) first-order term. A term is *regular* iff it has a finite amount of distinct sub-terms. All finite terms and some infinite terms are regular. All irregular terms are infinite.

wider range of infinite data structure. This is version two of CUP and was published in the 28th European Symposium on Programming (ESOP'19).

Version three is the author's simplification of version two by turning the notion of fixed-point terms back to the simple one used in version one, while keeping the improved guarding mechanism used in version two. This makes version three a minimum proof of concept for CUP. It first appeared in his technical report [21], then with some notational revision, presented by him in the PARIS 2018 workshop. It is version three that is presented here.

Section 4.1 presents the rules of CUP.

Section 4.2 lists some basic observations about CUP that are mainly used later on to support the model-theoretic soundness proof.

4.1 Coinductive Uniform Proof

The rules of CUP can be sorted into three groups:

1. the Uniform Proof rules,
2. the coinductive fixed point (co-fix) rule, and
3. the auxiliary rules.

These are all sequent manipulating rules of intuitionistic fashion, involving a signature and a finite set of formulae (i.e., assumptions) on the left side of each sequent, and a single formula (i.e., goal) on the right side.

The *Uniform Proof rules* consist of right-focusing rules, left-focusing rules and an assumption-choosing rule called Decide. The *right-focusing rules*, as the name suggests, focus on the right side of a sequent, recursively breaking the goal into its sub-formulae according to the top level logic connective, until the goal is atomic. In other words, the right-focusing rules are responsible for breaking down a composite goal into several atomic goals. Once right-focusing rules have done their job, and given a resulting sequent, the *Decide* rule chooses an assumption from the left side of the sequent. This chosen assumption is highlighted by moving it onto the top of the sequent arrow, while leaving a copy on the left side. The *left-focusing rules* focus on the assumption chosen by the Decide rule, breaking it down recursively

into an atomic formula that may unify with the atomic goal. If they do unify then the atomic goal is proved, otherwise not proved.

The *auxiliary rules* are the *auxiliary right-focusing rules*, together with the *auxiliary Decide rule* named *Decide*$\langle\rangle$[3]. Although basically doing what their counterparts (in the Uniform Proof rules) do, the auxiliary rules have the extra responsibility to restrict the selection of the very first of all the involved assumptions, given that selection of an assumption usually occurs at several junctures in a successful proof built within the CUP framework.

When we use CUP to construct a proof for a formula φ with respect to a logic program (i.e., a set of assumptions) P, we first use the *co-fix rule* to assert φ as a special assumption called *coinductive hypothesis*. This means that a copy of φ is added to the left side of the sequent. So now we have two instances of φ in the sequent: one on the left, called coinductive hypothesis, and the other on the right, called the *coinductive goal*, enclosed by the angles mark $\langle\rangle$.

The coinductive goal is then broken down into an atomic goal[4] using the auxiliary right-focusing rules. In this process, all intermediate goals, as well as the final atomic goal, are enclosed by the angles mark $\langle\rangle$. In other words, the angles mark $\langle\rangle$ is passed from one goal to its subsequent goal during the breakdown of the coinductive goal. Such is the designed effect of the auxiliary right-focusing rules.

Then, for the sequent with the angles-mark-enclosed atomic goal on the right, *Decide*$\langle\rangle$ selects an assumption that can be any clause from P but *not* the coinductive hypothesis φ, and at the mean time it peels the angles mark $\langle\rangle$ off the atomic goal. From now on the Uniform Proof rules apply, and any clause from P, as well as the coinductive hypothesis φ, can be freely chosen by the Decide rule.

Definition 4.1. P is an *H-program* and φ is an *H-goal* iff $P \subset H_\Sigma \ni \varphi$ and P is a finite set.

Example 4.2. Let Σ include $\{q : \iota \to o, g : \iota \to \iota, s : \iota \to \iota, z : \iota\}$. Then the subset of H_Σ which consists of (4.1.0.1), (4.1.0.2) and (4.1.0.3) is an H-program. An H-goal

[3] Read "decide angles".

[4] Assume that the coinductive goal is a single and typical H-formula, then it will finally be reduced to a single atomic goal, not plural atomic goals.

is (4.1.0.4).

$$\forall x.\, q\,(s\,(g\,x)) \wedge q\,(s\,(g\,(s\,x))) \wedge (q\,x) \to q\,(s\;x) \tag{4.1.0.1}$$

$$\forall x.\, q\,x \to q\,(g\,x) \tag{4.1.0.2}$$

$$q\,z \tag{4.1.0.3}$$

$$\forall x.\, q\,x \to q\,(s\,x) \tag{4.1.0.4}$$

Let Σ also include $\{p\colon \iota \to \iota \to o, h\colon \iota \to \iota \to \iota, f : \iota \to \iota\}$, and let N denote the guarded fixed-point[5] $(\mathbf{fix}\, y.\, \lambda x.\, h\, x\,(y\,(f\;x)))$. Then the subset of H_Σ which consists of (4.1.0.5) is an H-program. A related H-goal is (4.1.0.6).

$$\forall x.\, \forall y.\, p\,(f\;x)\,y \to p\,x\,(h\,x\,y) \tag{4.1.0.5}$$

$$\forall x.\, p\,x\,(N\;x) \tag{4.1.0.6}$$

Definition 4.3. Let P, Δ be two H-programs and φ be an H-goal. The relation $\Sigma; P; \Delta \Longrightarrow \varphi$ is given by Figure 4.1a.

Definition 4.4. Let P, Δ be two H-programs and φ be an H-goal. The relation $\Sigma; P; \Delta \Longrightarrow \langle\varphi\rangle$ is given by Figure 4.1b.

Definition 4.5. The relation $\Sigma; P \looparrowright \varphi$ between an H-goal φ and an H-program P, both on Σ, is given by Figure 4.1c.

Definition 4.6. A *sequent* is an expression in one of the forms:

- $\Sigma; P \looparrowright \varphi$
- $\Sigma; P; \Delta \Longrightarrow \langle\varphi\rangle$
- $\Sigma; P; \Delta \overset{D}{\Rightarrow} A$
- $\Sigma; P; \Delta \Longrightarrow \varphi$

Definition 4.7. If there is an expression of the form $\Sigma; P; \Delta$ on the left side of a sequent arrow, then any formula in Δ is called a *coinductive hypothesis*.

[5]See also Example 2.105

Definition 4.8. A *proof* of a sequent Q is a finite tree of sequents built using the rules of Figure 4.1, with Q as the root, and to every leaf sequent the INITIAL rule is applicable. In particular, a proof is a *coinductive uniform proof* iff its root has the form $\Sigma; P \looparrowright \varphi$. A proof is an *inductive uniform proof* iff its root has the form $\Sigma; P; \varnothing \Longrightarrow \varphi$.

Definition 4.9. The *coinductive uniform proof system* on Σ consists of H_Σ and the rules of Figure 4.1 (on page 55).

The reader may want to go directly from here to Example 4.10 and 5.14 for a complete worked-out proof construction in CUP.

4.2 Basic Observations about CUP

Inheriting goal-directed search from Uniform Proof, CUP is uniform in one additional sense: all proofs start with the co-fix rule, followed by using the auxiliary rules, then the Uniform Proof rules. We see this in detail next.

4.2.1 The Common Opening Pattern

Example 4.10. Let P denote the H-program that consists of (4.1.0.1), (4.1.0.2) and (4.1.0.3). Let φ denote the H-goal (4.1.0.4). The proof of $\Sigma; P \looparrowright \forall x.\, q\, x \to q\, (s\, x)$ starts with the steps given in Figure 4.2 (on page 56).

Example 4.11. Let P denote the H-program that consists of (4.1.0.5). Let φ denote the H-goal (4.1.0.6). The proof of $\Sigma; P \looparrowright \varphi$ starts with the steps given in Figure 4.3 (on page 56).

We now formalize the observation that all proof construction in the CUP framework share a common opening pattern, featuring an immediate use of the co-fix rule, then the auxiliary rules, and finally Uniform Proof rules.

Notation 4.12. We may use $\vec{A}$ to abbreviate $A_1 \wedge \cdots \wedge A_n$, and depict an H-formula as $(\forall \vec{x}.\, \vec{A} \to A)$. The notation $A_i \in \vec{A}$ means A_i is a conjunct of $\vec{A}$.

Notation 4.13. A list $[N_1/x_1], \ldots, [N_m/x_m]$ of substitutions can be more neatly denoted by $[N_1/x_1, \ldots, N_m/x_m]$, and can be further abbreviated using $[\vec{N}/\vec{x}]$, where $\vec{N}$ and $\vec{x}$ can then be used to refer to $N_1, \ldots, N_m$ and $x_1, \ldots, x_m$ respectively.

$$\frac{c:\iota,\Sigma;P;\Delta \Longrightarrow G\,[c/x] \qquad c:\iota\notin\Sigma}{\Sigma;P;\Delta \Longrightarrow \forall x:\iota.\,G}\ \forall R$$

$$\frac{\Sigma;P,D;\Delta \Longrightarrow G}{\Sigma;P;\Delta \Longrightarrow D\to G}\ {\to}R$$

$$\frac{\Sigma;P;\Delta \Longrightarrow G_1 \qquad \Sigma;P;\Delta \Longrightarrow G_2}{\Sigma;P;\Delta \Longrightarrow G_1\wedge G_2}\ \wedge R$$

$$\frac{\Sigma;P;\Delta \overset{D}{\Rightarrow} A \qquad D\in P\cup\Delta}{\Sigma;P;\Delta \Longrightarrow A}\ \text{DECIDE}$$

$$\frac{\Sigma;P;\Delta \xrightarrow{D[N/x]} A \qquad \begin{bmatrix}\Sigma;\varnothing\vdash_g N:\iota\\ D\,[N/x]\in H_\Sigma\end{bmatrix}}{\Sigma;P;\Delta \xrightarrow{\forall x:\iota.\,D} A}\ \forall L$$

$$\frac{\Sigma;P;\Delta \overset{D}{\Rightarrow} A \qquad \Sigma;P;\Delta \Longrightarrow G}{\Sigma;P;\Delta \xrightarrow{G\to D} A}\ {\to}L$$

$$\frac{\Sigma;P;\Delta \overset{D_x}{\Longrightarrow} A \qquad x\in\{1,2\}}{\Sigma;P;\Delta \xrightarrow{D_1\wedge D_2} A}\ \wedge L$$

$$\frac{A\equiv A'}{\Sigma;P;\Delta \overset{A'}{\Rightarrow} A}\ \text{INITIAL}$$

(a) The *Uniform Proof* Subsystem of CUP.

$$\frac{c:\iota,\Sigma;P;\Delta \Longrightarrow \langle M\,[c/x]\rangle \qquad c:\iota\notin\Sigma}{\Sigma;P;\Delta \Longrightarrow \langle\forall x:\iota.\,M\rangle}\ \forall R\langle\rangle$$

$$\frac{\Sigma;P,M_1;\Delta \Longrightarrow \langle M_2\rangle}{\Sigma;P;\Delta \Longrightarrow \langle M_1\to M_2\rangle}\ {\to}R\langle\rangle$$

$$\frac{\Sigma;P;\Delta \overset{D}{\Rightarrow} A \qquad P\ni D\notin\Delta}{\Sigma;P;\Delta \Longrightarrow \langle A\rangle}\ \text{DECIDE}\langle\rangle$$

(b) The *Guarded Uniform Proof* Subsystem of CUP.

$$\frac{\Sigma;P;\varphi \Longrightarrow \langle\varphi\rangle}{\Sigma;P \looparrowright \varphi}\ \text{CO-FIX}$$

(c) The *Coinduction* Subsystem of CUP.

Figure 4.1: The *Coinductive Uniform Proof* (CUP) System.

$$
\cfrac{\cfrac{\cfrac{\cfrac{c, \Sigma; P, (q\ c); \varphi \xRightarrow{(4.1.0.1)} q\ (s\ c)}{c, \Sigma; P, (q\ c); \varphi \Longrightarrow \langle q\ (s\ c) \rangle}\ \textsc{decide}\langle\rangle}{c, \Sigma; P; \varphi \Longrightarrow \langle q\ c \rightarrow q\ (s\ c) \rangle}\ {\rightarrow}R\langle\rangle}{\Sigma; P; \varphi \Longrightarrow \langle \forall x.\, q\ x \rightarrow q\ (s\ x) \rangle}\ \forall R\langle\rangle}{\Sigma; P \looparrowright \forall x.\, q\ x \rightarrow q\ (s\ x)}\ \textsc{co-fix}
$$

Figure 4.2: The proof steps referred to by Example 4.10.

$$
\cfrac{\cfrac{\cfrac{c, \Sigma; P; \varphi \xRightarrow{(4.1.0.5)} p\ c\ (N\ c)}{c, \Sigma; P; \varphi \Longrightarrow \langle p\ c\ (N\ c) \rangle}\ \textsc{decide}\langle\rangle}{\Sigma; P; \varphi \Longrightarrow \langle \forall x.\, p\ x\ (N\ x) \rangle}\ \forall R\langle\rangle}{\Sigma; P \looparrowright \forall x.\, p\ x\ (N\ x)}\ \textsc{co-fix}
$$

Figure 4.3: The proof steps referred to by Example 4.11.

Notation 4.14. $\vec{c}, \Sigma$ denotes a signature obtained by extending Σ by members of $\vec{c}$. Then, since a signature is a set, there is no collision regarding what "$\vec{c}$" stands for in the two expressions "$\vec{c}, \Sigma$" and "$[\vec{c}/\vec{x}]$".

Proposition 4.15. *All coinductive uniform proofs have the same opening pattern given by Figure 4.4 (on page 56).*

Corollary 4.16. *Every coinductive uniform proof has one and only one coinductive hypothesis.*

Next we identify some trivial proofs in CUP which does not actually concern coinduction.

$$
\cfrac{\cfrac{\cfrac{\cfrac{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \forall \vec{x}.\, \vec{A} \rightarrow A \overset{D}{\Rightarrow} A[\vec{c}/\vec{x}]}{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \forall \vec{x}.\, \vec{A} \rightarrow A \Longrightarrow \langle A[\vec{c}/\vec{x}] \rangle}\ \textsc{decide}\langle\rangle}{\vec{c}, \Sigma; P; \forall \vec{x}.\, \vec{A} \rightarrow A \Longrightarrow \langle (\vec{A} \rightarrow A)[\vec{c}/\vec{x}] \rangle}\ {\rightarrow}R\langle\rangle}{\Sigma; P; \forall \vec{x}.\, \vec{A} \rightarrow A \Longrightarrow \langle \forall \vec{x}.\, \vec{A} \rightarrow A \rangle}\ \forall R\langle\rangle \ldots \forall R\langle\rangle}{\Sigma; P \looparrowright \forall \vec{x}.\, \vec{A} \rightarrow A}\ \textsc{co-fix}
$$

Figure 4.4: The opening pattern of coinductive uniform proofs.

4.2.2 Trivial Coinductive Uniform Proofs

Definition 4.17. An H-formula is a *rule* iff it has the form $\forall \vec{x}.\, \vec{A} \to A$ where $\vec{A}$ is nonempty. Otherwise, it has the form $\forall \vec{x}.\, A$ and is called a *fact*.

Proposition 4.18. *Speaking with respect to Figure 4.4:*

1. *If the coinductive hypothesis is never selected throughout the coinductive uniform proof, then* $\Sigma; P; \varnothing \Longrightarrow \forall \vec{x}.\, \vec{A} \to A$.

2. *If* D *is* $\vec{A}[\vec{c}/\vec{x}]$*, then* $\Sigma; P; \varnothing \Longrightarrow \forall \vec{x}.\, \vec{A} \to A$ *and* $\forall \vec{x}.\, \vec{A} \to A$ *is a tautology.*

3. *If* D *is a fact in* P*, then* $\Sigma; P; \varnothing \Longrightarrow \forall \vec{x}.\, \vec{A} \to A$.

Proof. First we prove conclusion 1. If the coinductive hypothesis is never selected throughout the coinductive uniform proof, then the proof of

$$\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \forall \vec{x}.\, \vec{A} \to A \overset{D}{\Rightarrow} A[\vec{c}/\vec{x}]$$

which is implicit in Figure 4.4, can be converted into a proof of

$$\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \varnothing \overset{D}{\Rightarrow} A[\vec{c}/\vec{x}]$$

by systematically set the coinductive hypothesis $\forall \vec{x}.\, \vec{A} \to A$ to $\varnothing$. This enables us to construct a proof of $\Sigma; P; \varnothing \Longrightarrow \forall \vec{x}.\, \vec{A} \to A$, as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(Provable)}}{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \varnothing \overset{D}{\Rightarrow} A[\vec{c}/\vec{x}]}}{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \varnothing \Longrightarrow A[\vec{c}/\vec{x}]}\,\text{DECIDE}}{\vec{c}, \Sigma; P; \varnothing \Longrightarrow (\vec{A} \to A)[\vec{c}/\vec{x}]}\,{\to}R}{\Sigma; P; \varnothing \Longrightarrow \forall \vec{x}.\, \vec{A} \to A}\,\forall R \ldots \forall R$$

Conclusion 2 follows conclusion 1. If D is $\vec{A}[\vec{c}/\vec{x}]$, then the pattern in Figure 4.4 proceeds as follows:

$$\dfrac{\dfrac{A[\vec{c}/\vec{x}] \equiv A_i[\vec{c}/\vec{x}]}{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \forall \vec{x}.\, \vec{A} \to A \xRightarrow{A_i[\vec{c}/\vec{x}]} A[\vec{c}/\vec{x}]}\,\text{INITIAL}}{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \forall \vec{x}.\, \vec{A} \to A \xRightarrow{\vec{A}[\vec{c}/\vec{x}]} A[\vec{c}/\vec{x}]}\,\wedge L \ldots \wedge L$$

Obviously there is no involvement of the coinductive hypothesis throughout, hence conclusion 1 is applicable. Also note that the relation $A[\vec{c}/\vec{x}] \equiv A_i[\vec{c}/\vec{x}]$ implies that $A \equiv A_i$ for some $A_i \in \vec{A}$. So $\forall \vec{x}.\, \vec{A} \to A$ is a tautology.

Conclusion 3 follows conclusion 1. If D a fact in P, then assuming D has the form $(\forall \vec{y}.\, B)$, the pattern in Figure 4.4 proceeds as follows:

$$\dfrac{\dfrac{A[\vec{c}/\vec{x}] \equiv B[\vec{N}/\vec{y}]}{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \forall \vec{x}.\, \vec{A} \to A \xrightarrow{B[\vec{N}/\vec{y}]} A[\vec{c}/\vec{x}]}\ \text{INITIAL}}{\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \forall \vec{x}.\, \vec{A} \to A \xrightarrow{\forall \vec{y}.\, B} A[\vec{c}/\vec{x}]}\ \forall L \ldots \forall L$$

Obviously there is no involvement of the coinductive hypothesis throughout, hence conclusion 1 is applicable.

□

The definition below is motivated by Proposition 4.18.

Definition 4.19. Speaking with respect to Figure 4.4: a coinductive uniform proof is *trivial*, iff

- D is $\vec{A}[\vec{c}/\vec{x}]$, or
- D is a fact in P, or
- D is a rule in P and the coinductive hypothesis is never selected throughout the proof.

Otherwise, the coinductive uniform proof is *non-trivial*, in which case D is a rule in P and the coinductive hypothesis is selected for at least once in the proof.

So far we have presented the CUP framework, including its language and rules. We have also seen that all proofs in CUP share the same opening pattern, and that some proofs in CUP are trivial, being equivalent to inductive proofs. These basic observations are helpful when we formally establish the soundness of CUP, which we do next.

Chapter 5

Model-theoretic Soundness of Coinductive Uniform Proof

The relevance of CUP to logic programming depends on its model-theoretic soundness. The proof is interesting because it involves explicit extraction of a coinductive invariant (post-fixed-point) from a CUP tree. The full proof was first given in the author's annual report [21] phrased in traditional logic programming terminology, called the *report version*. The argument in the report version is verbose and not modularized. The ESOP paper [20] highlights the post-fixed-point extraction step by stating it as a lemma, while casting all technical concepts into category-theoretic notions. This proof is called the *conference version*, which shows an attempt to modularize and condense the argument. But due to page limitation, the conference version does not include a proof for the post-fixed-point extraction lemma. The proof that is presented here, the *thesis version*, is different from both earlier versions: it is modularized, condensed, complete, and in traditional logic programming terminology. It also highlights the top-level argument structure that was followed by the report version in an unduly quiet way.

The top-level structure of the soundness argument is given by Figure 5.1. We rely on a definition of Horn$^\omega$ clauses, which are Horn clauses involving explicit infinite trees. The concepts involved in the proof are introduced incrementally.

Section 5.1 defines the language of Horn$^\omega$ clauses and the greatest fixed-point model.

Section 5.2 draws our attention to a certain kind of sequent in a CUP tree, which is of particular interest in the soundness proof.

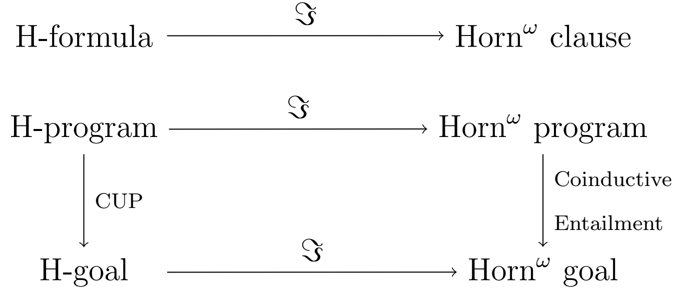


Figure 5.1: Top-level structure of CUP's model-theoretic soundness proof.

Section 5.3 highlights a certain kind of substitution in a CUP tree, which is a basic building block in the construction process of the proof.

Section 5.4 is the key constructive step in the proof, and is the part that was included in the conference version of the proof. See Definition 5.32.

Section 5.5 gives the important argument that is omitted from the conference version, which justifies the constructive step. See particularly Lemma 5.37.

Section 5.6 mainly harvests the results from earlier sections and states the soundness properties of CUP. See particularly Theorems 5.42 and 5.46.

5.1 Horn$^\omega$ Clause

Due to the way guarded lambda terms are defined, an infinite tree encoded by a guarded lambda term can only have a finite number of distinct free variables. So we define a *Horn$^\omega$ clause* as a formula built using such infinite trees and first-order predicates.

Definition 5.1. A *Horn$^\omega$ clause* on Σ is a pair $(t, \mathbf{t})$ where

- $t \in \mathbf{Atom}^\omega(\Sigma)$, $FV(t)$ is finite, and
- $\forall u \in \mathbf{t}$, $u \in \mathbf{Atom}^\omega(\Sigma)$ and $FV(u)$ is finite.

t is the *head*, and $\mathbf{t}$ is the *body*.[1]

Proposition 5.2. *If $\forall \vec{x}.\, A_1 \wedge \cdots \wedge A_n \to A_0$ is an H-formula, then $(t, \mathbf{t})$ is a Horn$^\omega$ clause, where $t = \Im(A_0)$ and $\mathbf{t} = \Im(A_1), \ldots, \Im(A_n)$.*

[1]Compare with Def. 2.59. Note that a variable may occur infinitely often in a Horn$^\omega$ clause.

Proof. Use Definitions 2.93 and 2.113. □

Notation 5.3. If φ is an H-formula of the form $\forall\vec{x}. A_1 \wedge \cdots \wedge A_n \rightarrow A_0$, then we use $\Im(\varphi)$ to denote the Horn$^\omega$ clause $(t, \mathbf{t})$ where $t = \Im(A_0)$ and $\mathbf{t} = \Im(A_1), \ldots, \Im(A_n)$.

Definition 5.4. A *Horn$^\omega$ program* on Σ is a finite set of Horn$^\omega$ clauses on Σ.

Corollary 5.5. *If $\{\varphi_1, \ldots, \varphi_n\}$ is an H-program, then $\{\Im(\varphi_1), \ldots, \Im(\varphi_n)\}$ is a Horn$^\omega$ program.*

Proof. Use Proposition 5.2. □

Notation 5.6. If $P = \{\varphi_1, \ldots, \varphi_n\}$ is an H-program, then we use $\Im(P)$ to denote the Horn$^\omega$ program $\{\Im(\varphi_1), \ldots, \Im(\varphi_n)\}$.

The standard fixed-point model theory [13, §5, §6, §25] for Horn clauses can be adapted for Horn$^\omega$ clauses. This will involve a definition of a monotonic immediate consequence operator for a typical Horn$^\omega$ program, and checking that this operator is working on a complete lattice. Then, by Tarski's fixed-point theorem, the operator is guaranteed to have fixed-points which are models of the logic program underlying the operator.

5.1.1 Horn$^\omega$ Immediate Consequence Operator

Since it is a standard result that the set of all complete Herbrand interpretations is a complete lattice [13, §25], we only need to define the customized immediate consequence operator and show that it is monotonic.

Definition 5.7. A *ground ω-substitution* on Σ is a partial function from *Var* to $1st\mathbf{GTerm}^\omega(\Sigma)$.

Definition 5.8. A *ground instance* of a Horn$^\omega$ clause $(t, \mathbf{t})$ on Σ is $(\theta(t), \theta(\mathbf{t}))$ where θ is a ground ω-substitution, and $FV(t) \cup FV(\mathbf{t})$ is a subset of the domain of θ.

Definition 5.9. The *immediate consequence operator* T_P of a Horn$^\omega$ program P on Σ is a mapping from and to the set of all complete Herbrand interpretations on Σ. We require that $t \in T_P(I)$ iff there exists a ground instance $(t, \mathbf{u})$ of some Horn$^\omega$ clause in P, such that $\forall u \in \mathbf{u}, u \in I$.

Lemma 5.10. *Let T_P be the* immediate consequence operator *of a Horn$^\omega$ program P on Σ. Then, T_P is monotonic. In other words, if $I \subseteq I'$ then $T_P(I) \subseteq T_P(I')$.*

Proof. By Definition 5.9. □

With the monotonic operator on a complete lattice defined, we can now define the complete coinductive model for Horn$^\omega$ programs.

5.1.2 Horn$^\omega$ Greatest Complete Herbrand Model

With the definition of the coinductive model comes a coinductive proof principle: to show that a set S of atomic formulae is included by the model, we could instead show that there exists a post-fixed-point of the operator, such that all atoms in S are included in this post-fixed-point. Since the coinductive model is the union of all post-fixed-points, it follows that the atoms in S must be in the model. We also stretch this principle a bit further, and show that its converse is also true: if a set of atoms are in the coinductive model, then there must be a post-fixed-point that contains all these atoms. This bidirectional proposition will be useful later when we show soundness of CUP, and the additional fact that if a formula is provable by CUP, we can use it as a lemma to prove further conclusions coinductively.

Proposition 5.11. *Let T_P be the immediate consequence operator of a Horn$^\omega$ program P on Σ. T_P has a greatest fixed-point $gfp(T_P)$, given by:*

$$gfp(T_P) = \bigcup\{I \mid I \subseteq T_P(I)\} \tag{5.1.2.1}$$

Proof. Apply Tarski's fixed-point theorem. Cf. the proof of Proposition 2.67. □

Definition 5.12. Let T_P be the immediate consequence operator of a Horn$^\omega$ program P on Σ.

- If $I \subseteq T_P(I)$, we call I a *post-fixed-point* of T_P.
- We refer to $gfp(T_P)$ as the *greatest complete Herbrand model* of P.

Lemma 5.13. *$S \subseteq gfp(T_P)$ iff there exists a post-fixed-point I of T_P and $S \subseteq I$.*

Proof. ($\Leftarrow$) This follows by Proposition 5.11. $gfp(T_P)$ is the union of all post-fixed-points of T_P. We have $S \subseteq I \subseteq gfp(T_P)$, so $S \subseteq gfp(T_P)$.

($\Rightarrow$) We construct a set I and show that $S \subseteq I$ and I is a post-fixed-point of T_P.

$$
\begin{aligned}
& S \subseteq gfp(T_P) \\
\text{iff} \quad & \forall x \in S,\ x \in gfp(T_P) \\
\text{iff} \quad & \forall x \in S, \text{ there exists a } T_P \text{ post-fixed-point } I_x \ni x.
\end{aligned}
$$

Let

$$I = \bigcup_{x \in S} I_x$$

Then $S \subseteq I$. Further,

$$
\begin{aligned}
\text{if} \quad & \forall x \in S,\ I_x \subseteq I \\
\text{then} \quad & \forall x \in S,\ T_P(I_x) \subseteq T_P(I) \quad \text{(Lem. 5.10)} \\
\text{then} \quad & \forall x \in S,\ I_x \subseteq T_P(I_x) \subseteq T_P(I) \\
\text{then} \quad & \forall x \in S,\ I_x \subseteq T_P(I) \\
\text{then} \quad & I \subseteq T_P(I).
\end{aligned}
$$

□

We now refer to a proof that is built within the CUP framework as a *sequent-tree*. With the coinductive model and the coinductive proof principle defined, we can now inspect a sequent-tree, and see that it provides sufficient information for us to construct a post-fixed-point that is the key to show the coinductive soundness of CUP.

Example 5.14. Building upon Figure 4.2 (on page 56), the rest of the sequent-tree is given by Figure 5.2 (on page 64).

5.2 Principal Back-chaining Sequent

A sequent-tree yields a tree-shaped dependency structure among atoms, called an *atom-tree*, that is basically the same as a standard finite success inductive proof

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(s\,c)} q\,(s\,c)}\ \text{INITIAL} \qquad \dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,c} q\,c}\ \text{INITIAL}}{\mathcal{LHS} \Longrightarrow q\,c}\ \text{DECIDE}}{\mathcal{LHS} \xRightarrow{q\,c \to q\,(s\,c)} q\,(s\,c)}\ \to L}{\mathcal{LHS} \xRightarrow{\varphi} q\,(s\,c)}\ \forall L \quad [c/x]}{\heartsuit : 4}\ \text{DECIDE}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q(s(g(s\,c)))} q(s(g(s\,c)))}\ \text{INIT.} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q(g(s\,c))} q(g(s\,c))}\ \text{INIT.} \qquad \dfrac{\heartsuit : 4}{\mathcal{LHS} \Longrightarrow q\,(s\,c)}}{\mathcal{LHS} \xRightarrow{q\,(s\,c) \to q\,(g\,(s\,c))} q\,(g\,(s\,c))}\ \to L}{\mathcal{LHS} \xRightarrow{(4.1.0.2)} q\,(g\,(s\,c))}\ \forall L}{\mathcal{LHS} \Longrightarrow q\,(g\,(s\,c))}\ \text{DEC.}}{\mathcal{LHS} \xRightarrow{q\,(g\,(s\,c)) \to q\,(s\,(g\,(s\,c)))} q\,(s\,(g\,(s\,c)))}\ \to L}{\mathcal{LHS} \xRightarrow{\varphi} q\,(s\,(g\,(s\,c)))}\ \forall L \quad [(g\,(s\,c))/x]}{\heartsuit : 3}\ \text{DECIDE}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(s\,(g\,c))} q\,(s\,(g\,c))}\ \text{INIT.} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(g\,c)} q\,(g\,c)}\ \text{INIT.} \qquad \dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,c} q\,c}\ \text{INIT.}}{\mathcal{LHS} \Longrightarrow q\,c}\ \text{DEC.}}{\mathcal{LHS} \xRightarrow{q\,c \to q\,(g\,c)} q\,(g\,c)}\ \to L}{\mathcal{LHS} \xRightarrow{(4.1.0.2)} q\,(g\,c)}\ \forall L}{\mathcal{LHS} \Longrightarrow q\,(g\,c)}\ \text{DECIDE}}{\mathcal{LHS} \xRightarrow{q\,(g\,c) \to q\,(s\,(g\,c))} q\,(s\,(g\,c))}\ \to L}{\mathcal{LHS} \xRightarrow{\varphi} q\,(s\,(g\,c))}\ \forall L \quad [(g\,c)/x]}{\heartsuit : 2}\ \text{DECIDE}$$

$$\dfrac{\dfrac{\heartsuit : 2}{\mathcal{LHS} \Longrightarrow q\,(s\,(g\,c))} \qquad \dfrac{\heartsuit : 3}{\mathcal{LHS} \Longrightarrow q\,(s\,(g\,(s\,c)))} \qquad \dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,c} q\,c}\ \text{INITIAL}}{\mathcal{LHS} \Longrightarrow q\,c}\ \text{DECIDE}}{\heartsuit : 1}\ \wedge R$$

$$\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(s\,c)} q\,(s\,c)}\ \text{INITIAL} \qquad \dfrac{\heartsuit : 1}{\mathcal{LHS} \Longrightarrow q\,(s\,(g\,c)) \wedge q\,(s\,(g\,(s\,c))) \wedge (q\,c)}}{\mathcal{LHS} \xRightarrow{q\,(s\,(g\,c)) \wedge q\,(s\,(g\,(s\,c))) \wedge (q\,c) \to q\,(s\,c)} q\,(s\,c)}\ \to L}{c, \Sigma; P, (q\,c); \varphi \xRightarrow{(4.1.0.1)} q\,(s\,c)}\ \forall L$$

Figure 5.2: The proof steps that build on Example 4.10 (i.e., Figure 4.2 on page 56) to complete the sequent-tree. Read bottom-up. $\mathcal{LHS}$ denotes the sequent left hand side $c, \Sigma; P, (q\,c); \varphi$.

tree[2] ("standard proof tree " for short). The difference between a standard proof tree and an atom-tree is as follows.

In a standard proof tree, every atom is immediately supported by a (possibly empty) set of atoms with which it forms a Horn clause that is a substitution instance of some clause from the logic program. In other words, in the standard scenario, it is only from the logic program that we select clauses to resolve atoms in goals. However, in a typical sequent-tree, atoms are resolved (or back-chained) not only against clauses from the logic program, but also against the coinductive hypothesis and the *implication hypothesis*, that is, the premise of the coinductive goal, provided that the goal contains implication ($\rightarrow$). These differences are translated into different kinds of edges and leaves when we draw the trees. If we use solid lines to connect a node with all its children to mean that the node is resolved by a clause from the program, then all edges in a standard proof tree are solid. Furthermore, if we use dashed lines to connect a node with all its children to mean that the node is resolved by the coinductive hypothesis, then a typical atom-tree would contain both solid and dashed edges. Moreover, if we underline a leaf node to mean that the atom inhabiting this leaf node is resolved against the implication hypothesis, then an atom-tree may have several of its leaf nodes underlined, but no underlined leaf node would appear in a standard proof tree.

Construction of the post-fixed-point starts with collecting the nodes of an atom-tree. When we try to extract an atom-tree from a sequent-tree, we notice that back-chaining of an atom in the latter may involve a consecutive steps of using left-focusing rules, and all sequents involved in these steps have the same atom on the right side. The definition of a *principal back-chaining sequent* we give now is an arbitrarily chosen one, among several equivalent ways, to extract atoms from a sequent-tree in order to form an atom-tree.

Definition 5.15. A sequent of the form $\Sigma; P; \Delta \xRightarrow{D} A$ is called a *back-chaining sequent*, where D is the *back-chaining formula*, and A the *back-chained atom*. In a coinductive uniform proof, a back-chaining sequent immediately on top of the horizontal line of an instance of the DECIDE or DECIDE$\langle\rangle$ rule is called a *principal back-chaining sequent*, and in which the back-chained atom is called a *principal back-chained atom*.

[2] For the notion of a proof tree, see [19, §1.6 p.21].

Example 5.16. Figure 5.3 (on page 67) is a reproduction of Figure 5.2 (on page 64) with all principal back-chaining sequents highlighted.

Although the concept of an atom-tree is helpful for the soundness proof development, we do not formally define it here, because it is not necessary for a formal presentation of the soundness proof — the nodes in an atom-tree are captured by the definition of principal back-chained atoms, while the dependency among these nodes are clear from the sequent-tree.

Example 5.17. Figure 5.4 (on page 68) shows the atom-tree extracted from the sequent-tree of Figure 5.3 (on page 67).

The auxiliary right-focusing rule for the universal quantifier ($\forall R\langle\rangle$) introduces a new function symbol of arity 0 to the signature. This function symbol, known as an *eigen-variable*, is used to replace the universally quantified variable in the goal. The basic discovery of the soundness proof is as follows. Using a system of eigen-variable substitutions to instantiate eigen-variables in an atom-tree, we can obtain an infinite amount of distinct *substitution instances of the atom-tree* ("atom-tree instances" for short), which do not contain eigen-variables. Recall that solid edges and dashed edges are used in an atom-tree to distinguish a node resolved by a program clause from a node resolved by the coinductive hypothesis. In the former case we say that the node is *firmly supported*, while in the latter case, *nominally supported*. The underlined leaf nodes in an atom-tree, i.e., those nodes that are resolved against the implication hypothesis, are also regarded as being nominally supported. The interesting observation, referred to as the *node-hopping effect*, is that nodes in all of these atom-tree instances are interconnected in such a way that every node is either firmly supported or if it is nominally supported, there must be an identical node in some other atom-tree instance which is firmly supported.

The building blocks of the system of eigen-variable substitutions are those ordinary substitutions that are involved in junctures of a sequent-tree where the coinductive hypothesis is selected by the Decide rule. We call them *δ-substitutions*.

5.3 δ-substitution

Definition 5.18. We call the list $[\vec{N}/\vec{x}]$ a *δ-substitution* in a given coinductive uniform proof iff:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(s\,c)} q\,(s\,c)}\,\text{INITIAL} \qquad \dfrac{\dfrac{}{\mathcal{LHS} \overset{q\,c}{\Rightarrow} q\,c}\,\text{INITIAL}}{\mathcal{LHS} \Longrightarrow q\,c}\,\text{DECIDE}}{\mathcal{LHS} \xRightarrow{q\,c \to q\,(s\,c)} q\,(s\,c)}\,{\to}L}{\mathcal{LHS} \overset{\varphi}{\Rightarrow} q\,(s\,c)}\,\forall L \quad [c/x]}{\heartsuit : 4}\,\text{DECIDE}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q(s(g(s\,c)))} q(s(g(s\,c)))}\,\text{INIT.} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q(g(s\,c))} q(g(s\,c))}\,\text{INIT.} \qquad \dfrac{\heartsuit : 4}{\mathcal{LHS} \Longrightarrow q\,(s\,c)}}{\mathcal{LHS} \xRightarrow{q\,(s\,c) \to q\,(g\,(s\,c))} q\,(g\,(s\,c))}\,{\to}L}{\mathcal{LHS} \xRightarrow{(4.1.0.2)} q\,(g\,(s\,c))}\,\forall L}{\mathcal{LHS} \Longrightarrow q\,(g\,(s\,c))}\,\text{DEC.}}{\mathcal{LHS} \xRightarrow{q\,(g\,(s\,c)) \to q\,(s\,(g\,(s\,c)))} q\,(s\,(g\,(s\,c)))}\,{\to}L}{\mathcal{LHS} \overset{\varphi}{\Rightarrow} q\,(s\,(g\,(s\,c)))}\,\forall L \quad [(g\,(s\,c))/x]}{\heartsuit : 3}\,\text{DECIDE}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(s\,(g\,c))} q\,(s\,(g\,c))}\,\text{INIT.} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(g\,c)} q\,(g\,c)}\,\text{INIT.} \qquad \dfrac{\dfrac{}{\mathcal{LHS} \overset{q\,c}{\Rightarrow} q\,c}\,\text{INIT.}}{\mathcal{LHS} \Longrightarrow q\,c}\,\text{DEC.}}{\mathcal{LHS} \xRightarrow{q\,c \to q\,(g\,c)} q\,(g\,c)}\,{\to}L}{\mathcal{LHS} \xRightarrow{(4.1.0.2)} q\,(g\,c)}\,\forall L}{\mathcal{LHS} \Longrightarrow q\,(g\,c)}\,\text{DECIDE}}{\mathcal{LHS} \xRightarrow{q\,(g\,c) \to q\,(s\,(g\,c))} q\,(s\,(g\,c))}\,{\to}L}{\mathcal{LHS} \overset{\varphi}{\Rightarrow} q\,(s\,(g\,c))}\,\forall L \quad [(g\,c)/x]}{\heartsuit : 2}\,\text{DECIDE}
$$

$$
\dfrac{\dfrac{\heartsuit : 2}{\mathcal{LHS} \Longrightarrow q\,(s\,(g\,c))} \qquad \dfrac{\heartsuit : 3}{\mathcal{LHS} \Longrightarrow q\,(s\,(g\,(s\,c)))} \qquad \dfrac{\dfrac{}{\mathcal{LHS} \overset{q\,c}{\Rightarrow} q\,c}\,\text{INITIAL}}{\mathcal{LHS} \Longrightarrow q\,c}\,\text{DECIDE}}{\heartsuit : 1}\,\wedge R
$$

$$
\dfrac{\dfrac{\dfrac{}{\mathcal{LHS} \xRightarrow{q\,(s\,c)} q\,(s\,c)}\,\text{INITIAL} \qquad \dfrac{\heartsuit : 1}{\mathcal{LHS} \Longrightarrow q\,(s\,(g\,c)) \wedge q\,(s\,(g\,(s\,c))) \wedge (q\,c)}}{\mathcal{LHS} \xRightarrow{q\,(s\,(g\,c)) \wedge q\,(s\,(g\,(s\,c))) \wedge (q\,c) \to q\,(s\,c)} q\,(s\,c)}\,{\to}L}{c, \Sigma; P, (q\,c); \varphi \xRightarrow{(4.1.0.1)} q\,(s\,c)}\,\forall L
$$

Figure 5.3: A reproduction of Figure 5.2 (on page 64) with all principal back-chaining sequents highlighted in green boxes, and all δ-substitutions highlighted in red boxes.

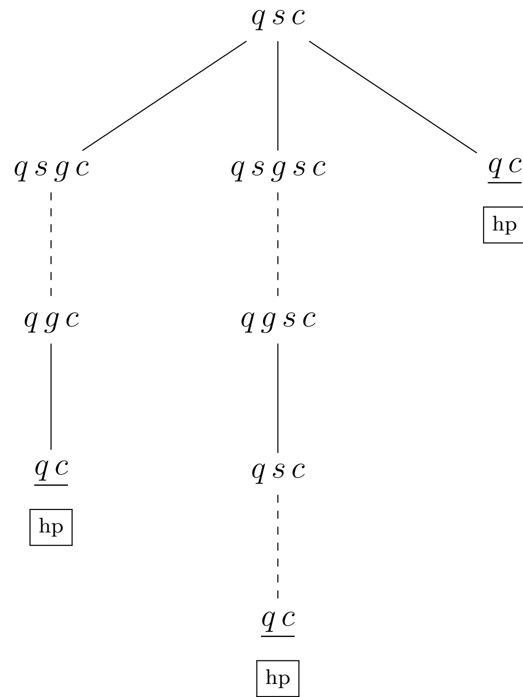


Figure 5.4: The atom-tree extracted from Figure 5.3 (on page 67). Parentheses in the atoms have been omitted. Dashed edges are from nodes proved using the coinductive hypothesis φ. Solid edges are from nodes proved using clauses from the program P. Underlined nodes with label "hp" are proved using the implication hypothesis $(q\,c)$.

1. there exists a principal back-chaining sequent $\Sigma; P; \Delta \overset{D}{\Rightarrow} A$, and

2. the back-chaining formula D is the coinductive hypothesis, and

3. $[\vec{N}/\vec{x}]$ is the list of all substitutions involved in the succession of $\forall L$ steps that instantiate all the $\forall$-quantified variables in D.

Example 5.19. In Figure 5.3 (on page 67), all (three) δ-substitutions are highlighted whilst all other substitutions are not displayed.

The δ-substitutions are interesting because they have the *composition effect* that refers to the existence of two constructive processes that both result in Figure 5.5 (on page 70). The first view is that Figure 5.5 shows the initial fragment of some infinite paths in the (infinite) standard proof tree for $q\ (s\ z)$ due to (4.1.0.1), which rules that a node of the form $q\,s\,x$ has children of the forms $q\,s\,g\,x$ (rendered as the left child), $q\,s\,g\,s\,x$ (rendered as the right child) and $q\ x$ (omitted). The second view is that if we extract from the δ-substitution $[(g\ c)/x]$ ($[(g\ (s\ c))/x]$) the prefix "g-" (resp. "gs-"), then Figure 5.5 is the result of 1) systematically composing the two prefixes, then 2) separately applying these compositions to the boxed letter z in the root atom, and 3) organizing the resulting atoms into a structure that agrees with the way in which the prefixes are systematically composed. Based on this (Figure 5.5) example we are not able to decide whether the composition effect is a coincidence or a general principle — our formal approach later shows that the latter is the case.

Proposition 5.20. *The following numbers counted with respect to a coinductive uniform proof are equal:*

- *The number of occasions on which the coinductive hypothesis is selected.*
- *The number of principal back-chaining sequents where the back-chaining formula is the coinductive hypothesis.*
- *The number of δ-substitutions.*

Proof. The occasion on which the coinductive hypothesis is selected, is when the coinductive hypothesis occurs as the back-chaining formula of some principal back-chaining sequent. A δ-substitution created in one occasion is distinguished from δ-substitutions created in other occasions if the coinductive hypothesis is selected on more than one occasion within a coinductive uniform proof. □

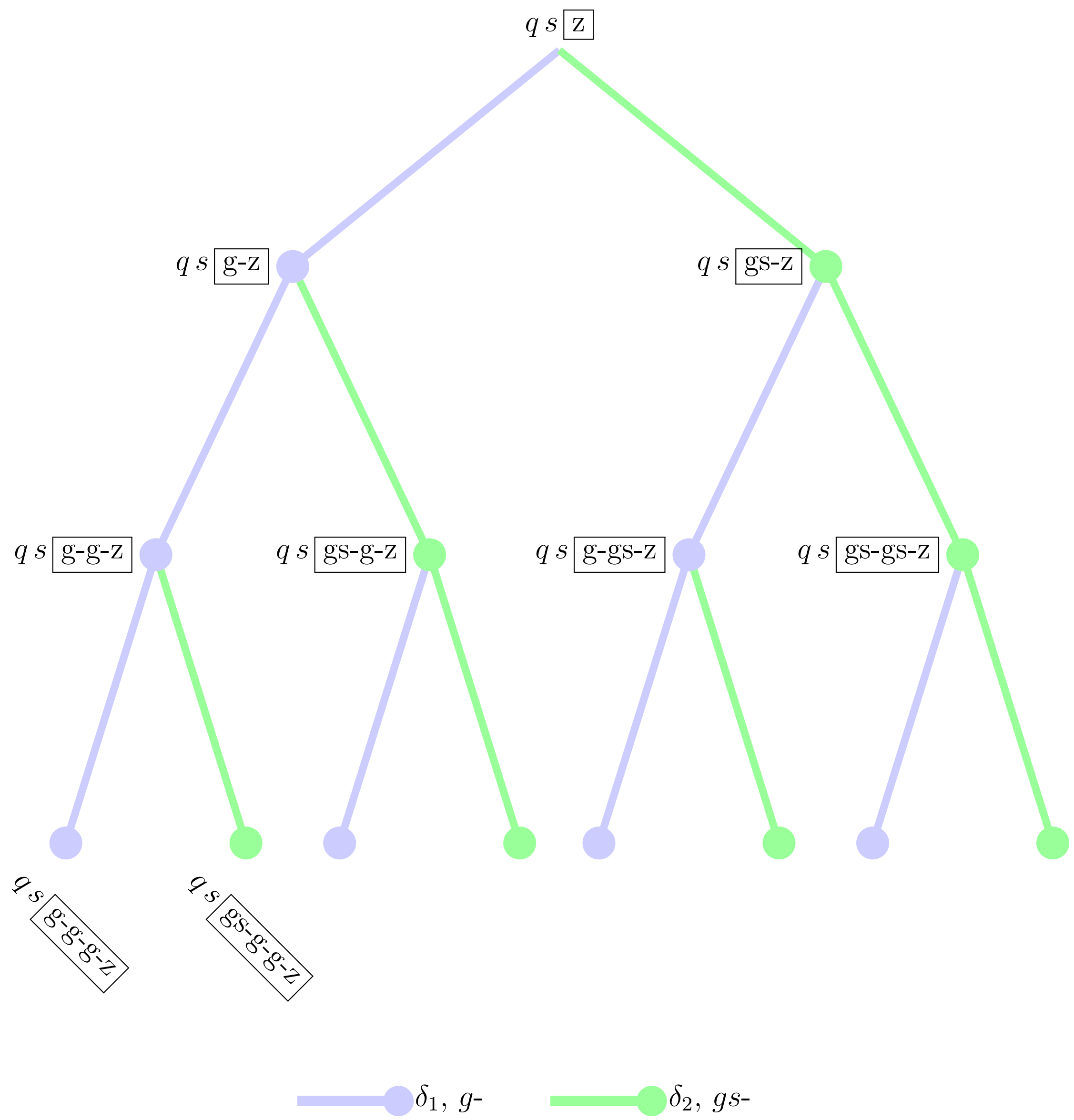


Figure 5.5: Composition effect.

Corollary 5.21. *A non-trivial coinductive uniform proof involves at least one δ-substitution.*

Notation 5.22. When the total number of δ-substitutions of a coinductive uniform proof is $r \geq 1$, we may denote the first δ-substitution by $[\vec{N_1}/\vec{x}]$, and denote the second δ-substitution by $[\vec{N_2}/\vec{x}]$, and so on, and denote the collection of all δ-substitutions by $[\vec{N_i}/\vec{x}]$ $(1 \leq i \leq r)$.

The composition effect motivates the definition of an *eigen-variable substitution system.*

5.4 Eigen-variable Substitution System

Definition 5.23. An *eigen-variable* is the new constant symbol added to the signature by $\forall R$ or $\forall R\langle\rangle$.[3]

Proposition 5.24. *In a coinductive uniform proof the number of eigen-variables in the signature of any principal back-chaining sequent equals the number of $\forall$-quantified variables in the coinductive hypothesis.*

Proof. Note that only $\forall R$ and $\forall R\langle\rangle$ can change the number of eigen-variables, but all coinductive uniform proofs do not involve the $\forall R$ rule. Therefore, for a coinductive uniform proof, $\forall R\langle\rangle$ is the only rule that can change the number of eigen-variables. As indicated by Figure 4.4, all instances of $\forall R\langle\rangle$ are below the first principal back-chaining sequent, so the number of eigen-variables in the signature of any principal back-chaining sequent is the same as that in the first principal back-chaining sequent.

On the other hand, the $\forall$-quantified variables removed by $\forall R\langle\rangle$ (read the rule bottom-up) are just copies of those in the coinductive hypothesis, and removal of one $\forall$-quantified variable results in addition of one eigen-variable. □

The definition below is motivated by Proposition 5.24.

Definition 5.25. The *eigen-variable list* of a coinductive uniform proof, is a list of all and only eigen-variables in the signature of a principal back-chaining sequent.

Notation 5.26 introduces more assumptions about an eigen-variable list.

[3] An eigen-variable is denoted c in $\forall R$ and $\forall R\langle\rangle$ in Figure 4.1.

Notation 5.26. We use $\vec{c}$ to denote an eigen-variable list. In order to make this notation consistent with existing ways of using " $\vec{c}$ " in notation like " $[\vec{c}/\vec{x}]$ " and " $\vec{c}, \Sigma$ ",[4] we always assume that an eigen-variable list is *well ordered and has no member-repetition*, in the following sense. Say the root sequent is $\Sigma; P \looparrowright \forall x_1. \forall x_2. \psi$ where ψ has no $\forall$-quantifier. Then we know that the $\forall R\langle\rangle$ steps (cf. Fig¿4.4) create a list of the form $[c_1/x_1\,,\, c_2/x_2]$, and that every δ-substitution has the form $[M_1/x_1\,,\, M_2/x_2]$. Then, the eigen-variable list, denoted $\vec{c}$, can only be c_1, c_2 and cannot be c_2, c_1 nor c_1, c_1, c_2. This agrees with abbreviating $\forall x_1. \forall x_2. \psi$, $[c_1/x_1\,,\, c_2/x_2]$ and $[M_1/x_1\,,\, M_2/x_2]$ by $\forall \vec{x}. \psi$, $[\vec{c}/\vec{x}]$ and $[\vec{M}/\vec{x}]$ respectively.

Definition 5.27. Given a coinductive uniform proof, whose eigen-variable list is $\vec{c}$, and whose root sequent signature is Σ, an *eigen-variable substitution* is a mapping from members of $\vec{c}$ to $1st\mathbf{GTerm}^{\omega}(\Sigma)$.

Notation 5.28. Let $\vec{c} = c_1, \ldots, c_m$ be an eigen-variable list, and let $\vec{t} = t_1, \ldots, t_m$ be a list of terms. We use $\vec{c} \mapsto \vec{t}$ to denote an eigen-variable substitution that maps c_i to t_i for all $1 \leq i \leq m$.

Definition 5.29. Applying an eigen-variable substitution $\vec{c} \mapsto \vec{t}$ ($\vec{t}$ on Σ) to a term $u \in 1st\mathbf{GTerm}^{\omega}(\vec{c}, \Sigma)$ means that, for each pair of $c_i \in \vec{c}$ and $t_i \in \vec{t}$, we substitute t_i for all occurrences of c_i in u.

Definition 5.30. Assume a coinductive uniform proof whose total number of δ-substitutions is $r \geq 1$. An *eigen-variable substitution index* (*EVS-index*, for short) is a word on $\{1, \ldots, r\}$.

Example 5.31. For a coinductive uniform proof with only 1 δ-substitution, its set of all EVS-indexes is infinite, consisting of all words on $\{1\}$ such as ϵ, $[1]$,$[1, 1]$ and $[1, 1, 1]$.

Definition 5.32, together with Notation 5.33 and 5.34, defines an eigen-variable substitution system.

Definition 5.32. Assume a coinductive uniform proof, with root sequent signature Σ, eigen-variable list $\vec{c}$, and δ-substitutions $[\vec{N_i}/\vec{x}]$ $(1 \leq i \leq r)$. An *eigen-variable substitution system* is a mapping Θ, from the set of all EVS-indexes, to the set of all

[4]Cf. Notation 4.13 and 4.14.

eigen-variable substitutions, defined as follows:

$$\Theta(\epsilon) \quad \text{is} \quad \vec{c} \mapsto \vec{t}$$

$$\Theta(wi) \quad \text{is} \quad \vec{c} \mapsto \Theta(w) \odot \Im(\vec{N_i})$$

where members of $\vec{t}$ are arbitrary, and $\vec{t}$ is called the *parameter* of Θ.

Notation 5.33. If S is a set or a list, then $\Im(S)$ denotes the result of applying the term/atom-value function $\Im$ to all members of S, and without disturbing the order if S is a list— i.e., if S is the list $M_1, \ldots, M_k$, then $\Im(S)$ denotes $\Im(M_1), \ldots, \Im(M_k)$.

Notation 5.34. If S is a set or a list, then $\Theta(w) \odot S$ denotes the result of applying the eigen-variable substitution $\Theta(w)$ to all members of S, and without disturbing the order if S is a list.

Example 5.35. Following Example 5.19, there are three δ-substitutions where:

$$\delta_1 \text{ is } [g\ c/x] \qquad \delta_2 \text{ is } [g\ (s\ c)/x] \qquad \delta_3 \text{ is } [c/x]$$

We shall have a countably infinite set of EVS-indexes on $\{1, 2, 3\}$. Below is a sample of the EVS-indexes.

$$\begin{pmatrix} \epsilon & [1] & [2] & [3] \\ [1,1] & [1,2] & [1,3] & [2,1] \\ [2,2] & [2,3] & [3,1] & [3,2] \\ [3,3] & [1,1,1] & [1,1,2] & [1,1,3] \\ [1,2,1] & [1,2,2] & [1,2,3] & \ldots \end{pmatrix}$$

Let $\Theta(\epsilon)$ be $c \mapsto z$. Then, some sample values of Θ[5] are given by:

$$\Theta(1) \quad \text{is} \quad c \mapsto \Theta(\epsilon) \odot \Im(g\ c) \quad \text{i.e.,} \quad c \mapsto g\ z$$

$$\Theta(2) \quad \text{is} \quad c \mapsto \Theta(\epsilon) \odot \Im(g\ (s\ c)) \quad \text{i.e.,} \quad c \mapsto g\ (s\ z)$$

$$\Theta(1,1) \quad \text{is} \quad c \mapsto \Theta(1) \odot \Im(g\ c) \quad \text{i.e.,} \quad c \mapsto g\ (g\ z)$$

The node-hopping effect can now be observed in Figure 5.6 on page 75. We use $\mathcal{A}_p$ to denote the atom-tree given by Figure 5.4 (on page 68) and use $\Theta(w) \odot \mathcal{A}_p$ to denote the atom-tree instance obtained by applying $\Theta(w)$ to $\mathcal{A}_p$. Figure 5.6

[5] We write $\Theta(1,1)$ as a shorthand for $\Theta([1,1])$.

involves three atom-tree instances: $\Theta(\epsilon) \odot \mathcal{A}_p$, $\Theta(1) \odot \mathcal{A}_p$ and $\Theta(2) \odot \mathcal{A}_p$ aligned in a column where a nominally supported node finds a firmly supported copy of itself by hoping downwards (upwards) if its corresponding node in $\mathcal{A}_p$ is resolved against the coinductive (resp. implication) hypothesis.

The node-hopping effect generalizes the composition effect. The latter focuses on nodes resolved against the coinductive hypothesis whilst the former additionally shows that nodes resolved against the implication hypothesis are also firmly supported. We now formalize the node-hopping effect in terms that the collection of all atoms from all atom-tree instances form a post-fixed-point of the related logic program.

5.5 Post-fixed-point Construction

Notation 5.36. Let $\mathbf{t}$ be the body of a Horn$^\omega$ clause, which is a list, and let S be a set. We overload the symbol $\subseteq$ and write $\mathbf{t} \subseteq S$ to mean that $\forall t \in \mathbf{t}$, $t \in S$.

Lemma 5.37. *Assume a non-trivial coinductive uniform proof, where:*

1. *The root sequent is* $\Sigma; P \looparrowright \varphi$*, and* φ *is* $\forall \vec{x}. \vec{A} \to A$.
2. *The immediate consequence operator of the Horn*$^\omega$ *program* $\Im(P)$ *is* T.
3. *The set of all atom-values of the principal back-chained atoms is* $\mathcal{A}_p$*, i.e.,* $\Im(B) \in \mathcal{A}_p$ *iff* B *is a principal back-chained atom.*
4. *The eigen-variable list is* $\vec{c}$*, and the eigen-variable substitution system with parameter* $\vec{t}$ *is* Θ.
5. *The ground instance of* $\Im(\varphi)$ *due to applying* $[\vec{x} \mapsto \vec{t}]$ *is* $(u, \mathbf{u})$.
6. $\mathbf{u} \subseteq gfp(T)$.
7. I_1 *is a post-fixed-point of* T*, with* $\mathbf{u} \subseteq I_1$.[6]
8. I_2 *is the union of all* $\Theta(w) \odot \mathcal{A}_p$ *with* w *ranges across the domain of* Θ.

Then,

[6] Lemma 5.13 guarantees existence of I_1.

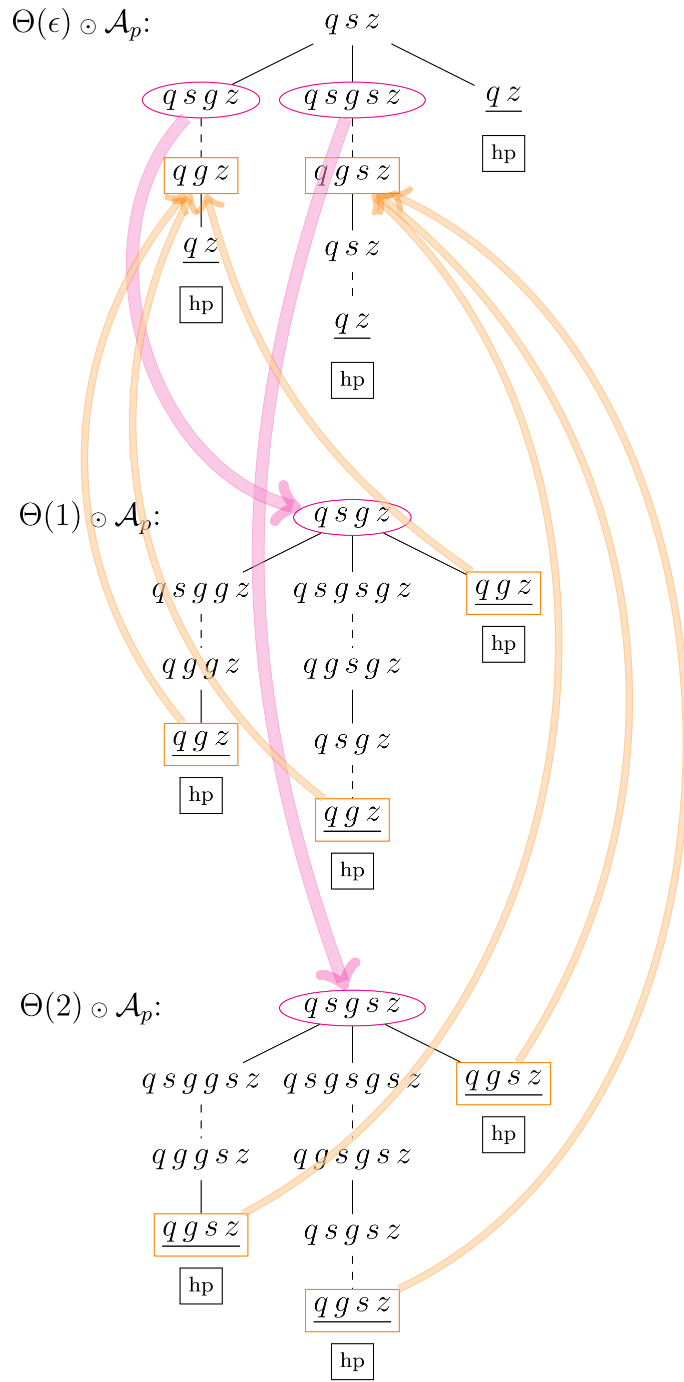


Figure 5.6: Node-hopping effect.

- $I = I_1 \cup I_2$ *is a post-fixed-point of* T*, i.e.,* $I \subseteq T(I)$*.*

- $u \in I$*.*

Proof. We first prove $u \in I$. Figure 4.4 shows that $A[\vec{c}/\vec{x}]$ is the back-chained atom of the very first principal back-chaining sequent. So $\Im(A[\vec{c}/\vec{x}]) \in \mathcal{A}_p$ and then $\Theta(\epsilon) \odot \Im(A[\vec{c}/\vec{x}]) \in \Theta(\epsilon) \odot \mathcal{A}_p \subseteq I$. Also note that

$$u = \Im(A)[\vec{x} \mapsto \vec{t}] = \Theta(\epsilon) \odot \Im(A[\vec{c}/\vec{x}])$$

so $u \in I$.

Next, we prove $I \subseteq T(I)$, i.e., $\forall x \in I$, $x \in T(I)$, by two cases:

Case 1 $\forall x \in I_1$, $x \in T(I)$.

Case 2 $\forall x \in I_2$, $x \in T(I)$.

[**Case 1**] If $x \in I_1$, then $x \in T(I_1)$. Since $I_1 \subseteq I$, and T is monotonic, we have $T(I_1) \subseteq T(I)$. So $x \in T(I)$. Now we have shown that $\forall x \in I_1$, $x \in T(I)$.

[**Case 2**] If $x \in I_2$, then 1) there exists an index w, such that $x \in \Theta(w) \odot \mathcal{A}_p$, and 2) there exists a principal back-chaining sequent

$$\vec{c}, \Sigma; \vec{A}[\vec{c}/\vec{x}], P; \varphi \overset{D}{\Rightarrow} B$$

such that x is $\Theta(w) \odot \Im(B)$. We observe three further cases over the back-chaining formula D below:

Case 2.1 $D \in P$.

Case 2.2 D is φ, the coinductive hypothesis.

Case 2.3 D is the implication hypothesis $\vec{A}[\vec{c}/\vec{x}]$ added by the $\rightarrow R\langle\rangle$ step.

[**Case 2.1**] If $D \in P$, then there exists a (possibly empty) list $\vec{B}$ of principal back-chained atoms, such that $\Theta(w) \odot \Im(\vec{B})$ is the body of a ground instance of some clause in $\Im(P)$, and the head of which is x. Since $\Theta(w) \odot \Im(\vec{B})) \subseteq \Theta(w) \odot \mathcal{A}_p \subseteq I$, we conclude that $x \in T(I)$.

[**Case 2.2**] If D is φ (i.e. $\forall \vec{x}. \vec{A} \rightarrow A$), then there exists a δ-substitution $[\vec{N_i}/\vec{x}]$, such that $A[\vec{N_i}/\vec{x}] \equiv B$, implying that $\Im(A[\vec{N_i}/\vec{x}]) = \Im(B)$, and then $\Theta(w) \odot \Im(A[\vec{N_i}/\vec{x}]) = \Theta(w) \odot \Im(B) = x$. Note that

$$\Theta(w) \odot \Im(A[\vec{N_i}/\vec{x}]) = \Theta(wi) \odot \Im(A[\vec{c}/\vec{x}])$$

by definition of Θ. Now it has been revealed that x is not only $\Theta(w) \odot \Im(B)$, but also $\Theta(wi) \odot \Im(A[\vec{c}/\vec{x}])$. It is shown by Figure 4.4 that $A[\vec{c}/\vec{x}]$ is the back-chained atom of the very first principal back-chaining sequent. Since the coinductive uniform proof is non-trivial, the back-chaining formula for $A[\vec{c}/\vec{x}]$ is a rule from P. Thus [**Case 2.2**] is reduced to [**Case 2.1**], so we can conclude that $x \in T(I)$.

[**Case 2.3**] If D is $\vec{A}[\vec{c}/\vec{x}]$, we can infer that there is some $A_k \in \vec{A}$ such that $A_k[\vec{c}/\vec{x}] \equiv B$, implying $\Im(A_k[\vec{c}/\vec{x}]) = \Im(B)$ and so $\Theta(w) \odot \Im(A_k[\vec{c}/\vec{x}]) = \Theta(w) \odot \Im(B) = x$. We see two further cases over the shape of w:

Case 2.3.1 w is ϵ.

Case 2.3.2 w is vj.

[**Case 2.3.1**] If w is ϵ, then note that

$$x = \Theta(\epsilon) \odot \Im(A_k[\vec{c}/\vec{x}]) = \Im(A_k)[\vec{x} \mapsto \vec{t}] = u_k \in \mathbf{u}$$

so $x \in \mathbf{u} \subseteq I_1$. Thus [**Case 2.3.1**] is reduced to [**Case 1**], so we can conclude that $x \in T(I)$.

[**Case 2.3.2**] If w is vj, then we rewrite $x = \Theta(w) \odot \Im(A_k[\vec{c}/\vec{x}])$ as $x = \Theta(vj) \odot \Im(A_k[\vec{c}/\vec{x}])$, and observe that

$$\Theta(vj) \odot \Im(A_k[\vec{c}/\vec{x}]) = \Theta(v) \odot \Im(A_k[\vec{N_j}/\vec{x}])$$

where $[\vec{N_j}/\vec{x}]$ is a δ-substitution, and $A_k[\vec{N_j}/\vec{x}]$ is a principal back-chained atom. Now it has been revealed that x is not only $\Theta(w) \odot \Im(B)$, but also $\Theta(v) \odot \Im(B')$ where *length*$(v) + 1 =$ *length*(w), and both B, B' are principal back-chained atoms. This brings us from [**Case 2.3.2**] back to [**Case 2**] but with a strictly smaller index — v instead of w. So we are guaranteed to be able to break out of the looping case analysis characterized by

[**Case 2**] $\to$ [**Case 2.3**] $\to$ [**Case 2.3.2**] $\to$ [**Case 2**]

and we will finally reach one of the cases (2.1 or 2.2 or 2.3.1) where we can conclude that $x \in T(I)$. $\square$

Now we can formally state the coinductive soundness of CUP.

5.6 Theorems on Model-theoretic Soundness

Notation 5.38. Let P be a Horn$^\omega$ program and $(t, \mathbf{t})$ be a Horn$^\omega$ clause. We write $P \approx\!\!\!\!\models (t, \mathbf{t})$, iff, for all ground instances $(t', \mathbf{t}')$ of $(t, \mathbf{t})$, if $\mathbf{t}' \subseteq gfp(T_P)$ then $t' \in gfp(T_P)$.

Corollary 5.39. *If $\Sigma; P \looparrowright \varphi$ is the root of a non-trivial coinductive uniform proof, then $\Im(P) \approx\!\!\!\!\models \Im(\varphi)$.*

Proof. This follows Lemma 5.37. □

Lemma 5.40. *If $\Sigma; P; \varnothing \Longrightarrow \varphi$, then $\Im(P) \approx\!\!\!\!\models \Im(\varphi)$.*

Proof. Trivial. □

Corollary 5.41. *If $\Sigma; P \looparrowright \varphi$ is the root of a trivial coinductive uniform proof, then $\Im(P) \approx\!\!\!\!\models \Im(\varphi)$.*

Proof. By Proposition 4.18 and Lemma 5.40. □

Theorem 5.42. $\Sigma; P \looparrowright \varphi$ *implies* $\Im(P) \approx\!\!\!\!\models \Im(\varphi)$.

Proof. By Corollaries 5.39 and 5.41. □

Definition 5.43. A Horn$^\omega$ clause $(u, \mathbf{u})$ is *GI-disjoint*[7] from a Horn$^\omega$ program P iff the set of all ground instances of $(u, \mathbf{u})$ is disjoint from the set of all ground instances of all clauses in P.

Definition 5.44. A Horn$^\omega$ clause $(u, \mathbf{u})$ is a *co-lemma* for a Horn$^\omega$ program P iff $(u, \mathbf{u})$ is GI-disjoint from P, and $P \approx\!\!\!\!\models (u, \mathbf{u})$, and for all ground instances $(u', \mathbf{u}')$ of $(u, \mathbf{u})$, $\mathbf{u}' \subseteq gfp(T_P)$.[8]

Lemma 5.45. *Let P be a Horn$^\omega$ program with immediate consequence operator T_P. Let $(u, \mathbf{u})$ be a co-lemma for P, and Let $Q = P \cup \{(u, \mathbf{u})\}$, with immediate consequence operator T_Q. Then, $gfp(T_P) = gfp(T_Q)$.*

Proof. 1) A post-fixed-point of T_P must be a post-fixed-point of T_Q, so $gfp(T_P) \subseteq gfp(T_Q)$.

2) If $I \subseteq T_Q(I)$ and $I \nsubseteq T_P(I)$, then $\exists x \in I$, such that a) $x \in T_Q(I)$ and b) $x \notin T_P(I)$. Statement a) holds iff there is $(t, \mathbf{t})$ that is a ground instance of some

[7] "GI" abbreviates "Ground Instance".

[8] "co-" in "co-lemma" stands for "coinductive".

clause in Q, such that $x = t$ and $\mathbf{t} \subseteq I$. Statement b) holds iff for all $(t, \mathbf{t})$ that is a ground instance of some clause in P, if $x = t$ then $\mathbf{t} \nsubseteq I$. Thus, combining a) and b), we know that there is $(t, \mathbf{t})$ that is a ground instance of the co-lemma $(u, \mathbf{u})$, such that $x = t$ and $\mathbf{t} \subseteq I$. Let X denote the set of all and only members of I for which a) and b) holds. Since $(u, \mathbf{u})$ is a co-lemma for P, we infer that $X \subseteq \mathit{gfp}(T_P)$. By Lemma 5.13 there is a post-fixed-point J of T_P such that $X \subseteq J$. Then $I \cup J$ is a post-fixed-point of T_P. Therefore, we conclude that if I is a post-fixed-point of T_Q but not a post-fixed-point of T_P, then I is a subset of some post-fixed-point of T_P. Then $\mathit{gfp}(T_Q) \subseteq \mathit{gfp}(T_P)$.

Combining 1) and 2) we have $\mathit{gfp}(T_P) = \mathit{gfp}(T_Q)$. □

Theorem 5.46. *If $\Sigma; P \looparrowright \varphi$, and $\Im(\varphi)$ is a co-lemma of $\Im(P)$, and $\Sigma; Q \looparrowright \psi$, where $Q = P \cup \{\varphi\}$, then $\Im(P) \approx\models \Im(\psi)$.*

Proof. By Theorem 5.42, $\Im(Q) \approx\models \Im(\psi)$. By Lemma 5.45, $\mathit{gfp}(T_P) = \mathit{gfp}(T_Q)$. Then $\Im P \approx\models \Im(\psi)$. □

We have established two theorems about the soundness of CUP. Theorem 5.42 says that if an H-formula is provable in CUP with respect to an H-program, then in the Horn$^\omega$ domain the corresponding Horn$^\omega$ clause is true with respect to the coinductive model of the related Horn$^\omega$ program. Theorem 5.46 allows us to break nested coinduction into basic coinduction — there is no Cut rule for CUP, but the function of a Cut rule is to some extent provided by this theorem.

Next we show the constructive nature of CUP from another aspect: its soundness with respect to coinductive intuitionist logic.

Chapter 6

Proof-theoretic Soundness of Coinductive Uniform Proof

The logic $\mathbf{iFOL}_{\blacktriangleright}$ is an extension of first-order intuitionistic logic with the modality $\blacktriangleright$ (read "later"), created by the author's collaborator Dr Henning Basold [22]. The constructiveness of CUP is backed by its soundness with respect to $\mathbf{iFOL}_{\blacktriangleright}$, referred to as *proof-theoretic soundness* of CUP. The $\mathbf{iFOL}_{\blacktriangleright}$ rules used here are modified from the original version in order to match the technical background of CUP. Dr Basold provides a soundness theorem for $\mathbf{iFOL}_{\blacktriangleright}$ relative to the Herbrand model [20]. This means that there are two ways to establish model-theoretic soundness of CUP: one is a direct proof that allows extraction of a coinductive invariant, the other is indirect — via $\mathbf{iFOL}_{\blacktriangleright}$, which shows the constructiveness of CUP. Figure 6.1 summarizes all the CUP soundness results we have so far. This chapter presents the author's full proof of soundness for CUP relative to $\mathbf{iFOL}_{\blacktriangleright}$.

Section 6.1 defines $\mathbf{iFOL}_{\blacktriangleright}$ and gives some derivable rules in $\mathbf{iFOL}_{\blacktriangleright}$ which we will use next.

Section 6.2 formulates the soundness theorem and provides a proof.

6.1 Coinductive Intuitionistic Logic

Definition 6.1. Let Σ be a signature. The formulae of the logic $\mathbf{iFOL}_{\blacktriangleright}$ over Σ are those defined in Figure. 2.4 (to be found on p.30) extended with the following rule.

$$\frac{\Sigma; \Gamma \Vdash \varphi}{\Sigma; \Gamma \Vdash \blacktriangleright \varphi}$$

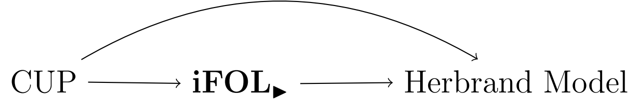


Figure 6.1: Summary of Soundness Results of CUP. → reads "is sound with respect to".

Conversion ($\equiv$) extends to these formulae in the obvious way.

Definition 6.2. Let φ be a formula and Δ a set of formulae. We say φ is *provable in context* Γ *under the assumptions* Δ, iff $\Gamma \mid \Delta \vdash \varphi$ holds. The *provability relation* $\vdash$ is given inductively by the rules in Figure 6.2.

Lemma 6.3. *The following* $\blacktriangleright$*-preservation rules are derivable in the logic* $\mathbf{iFOL}_\blacktriangleright$.

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright(\forall x_1 \cdots x_m.\, \varphi)}{\Gamma \mid \Delta \vdash \forall x_1 \cdots x_m.\, \blacktriangleright \varphi}\ (\blacktriangleright\textbf{-Pres-}\forall_r)$$

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright \varphi_1 \quad \cdots \quad \Gamma \mid \Delta \vdash \blacktriangleright \varphi_n}{\Gamma \mid \Delta \vdash \blacktriangleright(\varphi_1 \wedge \cdots \wedge \varphi_n)}\ (\blacktriangleright\textbf{-Pres-}\wedge_r)$$

Proof. We use the *distribution lemma* [22, p.132], which says that *if* an inference rule **(X)** is derivable for arbitrary Δ', *then* we can derive $(\blacktriangleright\textbf{-Pres-X})$.

$$\frac{\Gamma \mid \Delta, \Delta' \vdash \varphi_1 \qquad \cdots \qquad \Gamma \mid \Delta, \Delta' \vdash \varphi_n}{\Gamma \mid \Delta, \Delta' \vdash \psi}\ \textbf{(X)}$$

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright \varphi_1 \qquad \cdots \qquad \Gamma \mid \Delta \vdash \blacktriangleright \varphi_n}{\Gamma \mid \Delta \vdash \blacktriangleright \psi}\ (\blacktriangleright\textbf{-Pres-X})$$

We also use the *weakening lemma* [22, p.131], which is

$$\frac{\Gamma \mid \Delta \vdash \varphi}{\Gamma, x \mid \Delta \vdash \varphi}\ \textbf{(Weak)}$$

- We first prove $(\blacktriangleright\textbf{-Pres-}\forall_r)$.

$$\dfrac{\dfrac{\dfrac{\Gamma \mid \Delta \vdash \blacktriangleright \forall x_1 \cdots x_m.\, \varphi}{\Gamma, x_1, \cdots, x_m \mid \Delta \vdash \blacktriangleright \forall x_1 \cdots x_m.\, \varphi}\ \textbf{(Weak)}}{\Gamma, x_1, \cdots, x_m \mid \Delta \vdash \blacktriangleright \varphi}\ (*)}{\Gamma \mid \Delta \vdash \forall x_1 \cdots x_m.\, \blacktriangleright \varphi}\ (\forall\textbf{-I})$$

where $(*)$ is the rule obtained using the distribution lemma after instantiating **(X)** with $(\forall\textbf{-E})$.

$$\frac{\varphi \in \Delta}{\Gamma \mid \Delta \vdash \varphi}\ \textbf{(Proj)} \qquad \frac{\Gamma \mid \Delta \vdash \varphi' \qquad \varphi \equiv \varphi'}{\Gamma \mid \Delta \vdash \varphi}\ \textbf{(Conv)}$$

$$\frac{\Gamma \mid \Delta \vdash \varphi \qquad \Gamma \mid \Delta \vdash \psi}{\Gamma \mid \Delta \vdash \varphi \wedge \psi}\ (\wedge\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \varphi_1 \wedge \varphi_2 \qquad i \in \{1, 2\}}{\Gamma \mid \Delta \vdash \varphi_i}\ (\wedge_i\textbf{-E})$$

$$\frac{\Gamma \mid \Delta \vdash \varphi_i \qquad \Sigma; \Gamma \Vdash \varphi_j \qquad j \neq i}{\Gamma \mid \Delta \vdash \varphi_1 \vee \varphi_2}\ (\vee_i\textbf{-I})$$

$$\frac{\Gamma \mid \Delta, \varphi_1 \vdash \psi \qquad \Gamma \mid \Delta, \varphi_2 \vdash \psi}{\Gamma \mid \Delta, \varphi_1 \vee \varphi_2 \vdash \psi}\ (\vee\textbf{-E})$$

$$\frac{\Gamma \mid \Delta, \varphi \vdash \psi}{\Gamma \mid \Delta \vdash \varphi \to \psi}\ (\to\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \varphi \to \psi \qquad \Gamma \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \psi}\ (\to\textbf{-E})$$

$$\frac{\Gamma, x : \iota \mid \Delta \vdash \varphi \qquad x : \iota \notin \Gamma}{\Gamma \mid \Delta \vdash \forall x : \iota.\, \varphi}\ (\forall\textbf{-I})$$

$$\frac{\Gamma \mid \Delta \vdash \forall x : \iota.\, \varphi \qquad \left[\begin{array}{l} \Sigma; \Gamma \vdash_g N : \iota \\ \Sigma; \Gamma \Vdash \varphi\,[N/x] \end{array}\right]}{\Gamma \mid \Delta \vdash \varphi\,[N/x]}\ (\forall\textbf{-E})$$

$$\frac{\Sigma; \Gamma \vdash_g M : \iota \qquad \Gamma \mid \Delta \vdash \varphi\,[M/x]}{\Gamma \mid \Delta \vdash \exists x : \iota.\, \varphi}\ (\exists\textbf{-I})$$

$$\frac{\Sigma; \Gamma \Vdash \psi \qquad \Gamma, x : \iota \mid \Delta, \varphi \vdash \psi \qquad x : \iota \notin \Gamma}{\Gamma \mid \Delta, \exists x : \iota.\, \varphi \vdash \psi}\ (\exists\textbf{-E})$$

(a) Intuitionistic Rules for Standard Connectives

$$\frac{\Gamma \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \blacktriangleright \varphi}\ \textbf{(Next)} \qquad \frac{\Gamma \mid \Delta \vdash \blacktriangleright(\varphi \to \psi)}{\Gamma \mid \Delta \vdash \blacktriangleright \varphi \to \blacktriangleright \psi}\ \textbf{(Mon)}$$

$$\frac{\Gamma \mid \Delta, \blacktriangleright \varphi \vdash \varphi}{\Gamma \mid \Delta \vdash \varphi}\ \textbf{(Löb)}$$

(b) Rules for the Modality $\blacktriangleright$. **(Mon)** stands for *monotone*

Figure 6.2: Rules for the Logic $\textbf{iFOL}_{\blacktriangleright}$

$$\frac{\Gamma, x_1, \cdots, x_m \mid \Delta, \Delta' \vdash \forall x_1 \cdots x_m. \varphi}{\Gamma, x_1, \cdots, x_m \mid \Delta, \Delta' \vdash \varphi} \textbf{ (}\forall\textbf{-E)}$$

- We then prove **($\blacktriangleright$-Pres-$\wedge_r$)**. It results from the distribution lemma by instantiating **(X)** with **($\wedge$-I)**.

$$\frac{\Gamma \mid \Delta, \Delta' \vdash \varphi_1 \quad \cdots \quad \Gamma \mid \Delta, \Delta' \vdash \varphi_n}{\Gamma \mid \Delta, \Delta' \vdash \varphi_1 \wedge \cdots \wedge \varphi_n} \textbf{ (}\wedge\textbf{-I)}$$

□

6.2 CUP Proof-theoretic Soundness

The guarding of an H-formula is obtained by adding the symbol $\blacktriangleright$ (read "later") to every atom in the body of the H-formula. The guarding of an H-program is obtained by guarding every H-formula in it. Then, using case analysis and induction, we can show that if an H-program P entails a formula φ in CUP, then the guarding of P entails φ in $\mathbf{iFOL}_{\blacktriangleright}$. Guarding is necessitated by the fact that the co-fix rule of CUP introduces the angles mark $\langle\rangle$ on the right side of a sequent to protect coinductive soundness, while the Löb rule of $\mathbf{iFOL}_{\blacktriangleright}$ uses the later modality on the left to do the same.

Definition 6.4. Given an H-formula φ of the shape $\forall \vec{x}. (A_1 \wedge \cdots \wedge A_n) \rightarrow A$, we define its *guarding* $\overline{\varphi}$ to be $\forall \vec{x}. (\blacktriangleright A_1 \wedge \cdots \wedge \blacktriangleright A_n) \rightarrow A$. For a collection P of H-formulae, we define its guarding $\overline{P}$ by guarding each formula in P.

Notation 6.5. When we abbreviate an H-formula as $\forall \vec{x}. \vec{A} \rightarrow A$, its guarding is denoted by $\forall \vec{x}. \blacktriangleright \vec{A} \rightarrow A$, where $\blacktriangleright \vec{A}$ denotes $\blacktriangleright A_1 \wedge \cdots \wedge \blacktriangleright A_n$.

Theorem 6.6. *Given a signature Σ, an H-formula φ and an H-program P, if $\Sigma; P \looparrowright \varphi$ then $\varnothing \mid \overline{P} \vdash \varphi$.*

Proof. Our proof method is to discuss about the shape of the CUP sequent-tree in different cases, and *construct* an $\mathbf{iFOL}_{\blacktriangleright}$ proof for each case. For a less straight forward case, we use induction. The structure of the argument is given in Figure 6.3.

1. For a typical H-formula φ of the form $\forall \vec{x}. \vec{A} \rightarrow A$, the opening pattern of the CUP sequent-tree is given by Figure 4.4, reproduced below.

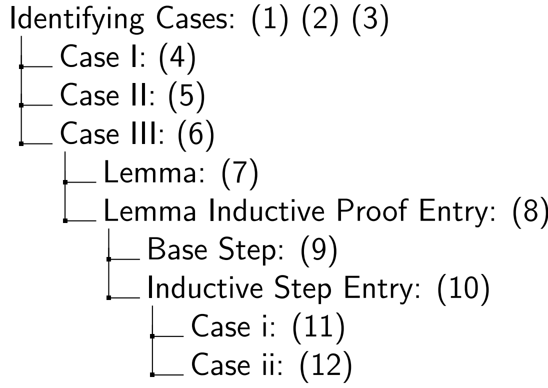


Figure 6.3: Structure of CUP proof-theoretic soundness proof.

$$\dfrac{\dfrac{\dfrac{\vec{c},\Sigma;\vec{A}[\vec{c}/\vec{x}],P;\forall\vec{x}.\,\vec{A}\to A \Longrightarrow \langle A[\vec{c}/\vec{x}]\rangle}{\vec{c},\Sigma;P;\forall\vec{x}.\,\vec{A}\to A\Longrightarrow\langle(\vec{A}\to A)[\vec{c}/\vec{x}]\rangle}\to R\langle\rangle}{\Sigma;P;\forall\vec{x}.\,\vec{A}\to A\Longrightarrow\langle\forall\vec{x}.\,\vec{A}\to A\rangle}\forall R\langle\rangle\ldots\forall R\langle\rangle}{\Sigma;P\looparrowright\forall\vec{x}.\,\vec{A}\to A}\text{CO-FIX}$$

Correspondingly, we build $\mathbf{iFOL}_{\blacktriangleright}$ proof steps, as follows.

$$\dfrac{\dfrac{\dfrac{\vec{x}\mid\overline{P},\blacktriangleright(\forall\vec{x}.\,\vec{A}\to A),\vec{A}\vdash A}{\vec{x}\mid\overline{P},\blacktriangleright(\forall\vec{x}.\,\vec{A}\to A)\vdash\vec{A}\to A}(\mathbf{\to\text{-}I})}{\varnothing\mid\overline{P},\blacktriangleright(\forall\vec{x}.\,\vec{A}\to A)\vdash\forall\vec{x}.\,\vec{A}\to A}(\mathbf{\forall\text{-}I})\ldots(\mathbf{\forall\text{-}I})}{\varnothing\mid\overline{P}\vdash\forall\vec{x}.\,\vec{A}\to A}(\textbf{Löb})$$

2. To save space, we will adopt the following abbreviations[1].

$$\Sigma';P';\Pi\quad\text{abbr.}\quad\vec{c},\Sigma;\vec{A}[\vec{c}/\vec{x}],P;\forall\vec{x}.\,\vec{A}\to A$$

$$\Gamma\mid\Delta\quad\text{abbr.}\quad\vec{x}\mid\overline{P},\blacktriangleright(\forall\vec{x}.\,\vec{A}\to A),\vec{A}$$

It remains to show that if $\Sigma';P';\Pi\Longrightarrow\langle A[\vec{c}/\vec{x}]\rangle$ then $\Gamma\mid\Delta\vdash A$.

3. For CUP, the next step must be using DECIDE$\langle\rangle$, and there are three cases of possible selection, which we analyze separately. DECIDE$\langle\rangle$ can only select one of the following items.

    (a) $\vec{A}[\vec{c}/\vec{x}]$.

    (b) A *fact* clause F from P, which is an H-formula *without* the connective $\to$.

    (c) A *rule* clause R from P, which is an H-formula *with* the connective $\to$.

[1] The letter Π is used earlier in the thesis to denote the subset of Σ involving predicates. Here we use Π to denote the coinductive hypothesis, and does not understand Π in the former way.

4. In case (3a), CUP proceeds as follows.

$$\dfrac{\dfrac{\dfrac{A_k[\vec{c}/\vec{x}] \equiv A[\vec{c}/\vec{x}]}{\Sigma'; P'; \Pi \xrightarrow{A_k[\vec{c}/\vec{x}]} A[\vec{c}/\vec{x}]}\ \textsc{initial}}{\Sigma'; P'; \Pi \xrightarrow{\vec{A}[\vec{c}/\vec{x}]} A[\vec{c}/\vec{x}]}\ \wedge L \ldots \wedge L}{\Sigma'; P'; \Pi \Longrightarrow \langle A[\vec{c}/\vec{x}]\rangle}\ \textsc{decide}\langle\rangle$$

Correspondingly, we build $\mathbf{iFOL}_{\blacktriangleright}$ proof steps, as follows.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash \vec{A}}\ \textbf{(Proj)}}{\Gamma \mid \Delta \vdash A_k}\ (\wedge\textbf{-E})\ldots(\wedge\textbf{-E}) \qquad A_k \equiv A}{\Gamma \mid \Delta \vdash A}\ \textbf{(Conv)}$$

5. In case (3b), CUP proceeds as follows.

$$\dfrac{\dfrac{\dfrac{F' \equiv A[\vec{c}/\vec{x}]}{\Sigma'; P'; \Pi \xRightarrow{F'} A[\vec{c}/\vec{x}]}\ \textsc{initial}}{\Sigma'; P'; \Pi \xRightarrow{F} A[\vec{c}/\vec{x}]}\ \forall L \ldots \forall L}{\Sigma'; P'; \Pi \Longrightarrow \langle A[\vec{c}/\vec{x}]\rangle}\ \textsc{decide}\langle\rangle$$

Correspondingly, we build $\mathbf{iFOL}_{\blacktriangleright}$ proof steps, as follows.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash F}\ \textbf{(Proj)}}{\Gamma \mid \Delta \vdash F'[\vec{x}/\vec{c}]}\ (\forall\textbf{-E})\ldots(\forall\textbf{-E}) \qquad F'[\vec{x}/\vec{c}] \equiv A}{\Gamma \mid \Delta \vdash A}\ \textbf{(Conv)}$$

6. We now study case (3c). For clarity, let us assume that the rule R, its ground instance R' and guarding $\overline{R}$ respectively have the typical forms

$$\forall \vec{y}.\, \vec{B} \to B \tag{R}$$

$$\vec{B}' \to B' \tag{R'}$$

$$\forall \vec{y}.\, \blacktriangleright \vec{B} \to B \tag{$\overline{R}$}$$

CUP proceeds as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{B' \equiv A[\vec{c}/\vec{x}]}{\Sigma'; P'; \Pi \xRightarrow{B'} A[\vec{c}/\vec{x}]}\ \textsc{initial} \qquad \Sigma'; P'; \Pi \Longrightarrow \vec{B}'}{\Sigma'; P'; \Pi \xRightarrow{\vec{B}' \to B'} A[\vec{c}/\vec{x}]}\ \to L}{\Sigma'; P'; \Pi \xRightarrow{\forall \vec{y}.\, \vec{B} \to B} A[\vec{c}/\vec{x}]}\ \forall L \ldots \forall L}{\Sigma'; P'; \Pi \Longrightarrow \langle A[\vec{c}/\vec{x}]\rangle}\ \textsc{decide}\langle\rangle$$

Correspondingly, we build $\mathbf{iFOL}_{\blacktriangleright}$ proof steps,

$$\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash \overline{R}}\ \textbf{(Proj)}}{\Gamma \mid \Delta \vdash \blacktriangleright(\vec{B}'[\vec{x}/\vec{c}]) \to B'[\vec{x}/\vec{c}]}\ (\forall\text{-}\mathbf{E})\ldots(\forall\text{-}\mathbf{E})$$
$$\heartsuit$$

$$\dfrac{\dfrac{\heartsuit \qquad \Gamma \mid \Delta \vdash \blacktriangleright(\vec{B}'[\vec{x}/\vec{c}])}{\Gamma \mid \Delta \vdash B'[\vec{x}/\vec{c}]}\ (\to\text{-}\mathbf{E}) \qquad B'[\vec{x}/\vec{c}] \equiv A}{\Gamma \mid \Delta \vdash A}\ \textbf{(Conv)}$$

7. To complete case (3c), and in light of the $(\blacktriangleright$**-Pres-**$\wedge_r)$ rule of Lemma 6.3, it remains to show Lemma A below. We generalize Lemma A into Lemma B, and see that Lemma A is indeed a special case of Lemma B, when C_j in (SQT) ranges over members of $\vec{B}'$.

Lemma A Let B'_j denote the j-th conjunct of $\vec{B}'$, which corresponds to $\blacktriangleright(B'_j[\vec{x}/\vec{c}])$ in $\blacktriangleright(\vec{B}'[\vec{x}/\vec{c}])$. If for all j, $\Sigma'; P'; \Pi \Longrightarrow B'_j$ has a proof, then for all j, $\Gamma \mid \Delta \vdash \blacktriangleright(B'_j[\vec{x}/\vec{c}])$ has a proof.

Lemma B Let $\{C_j \mid j \geq 1\}$ be an enumerable set of arbitrary ground atoms on Σ'. For all $\eta \geq 1$, if every member of the set (SQT) has a proof, then every member of the set (IFN) has a proof.

$$\{\ \Sigma'; P'; \Pi \Longrightarrow C_j \ \mid\ 1 \leq j \leq \eta\ \} \tag{SQT}$$

$$\{\ \Gamma \mid \Delta \vdash \blacktriangleright(C_j[\vec{x}/\vec{c}]) \ \mid\ 1 \leq j \leq \eta\ \} \tag{IFN}$$

8. To prove Lemma B, we use an inductive argument on the total number $\nu \geq 0$ of $\to L$ steps involved in the proofs of all sequents in (SQT). We show that *Lemma B holds for all* $\nu \geq 0$. The use of induction and the choice of the induction parameter ν is motivated by the following fact. To each sequent in (SQT), only the DECIDE rule can be applied, and there are several cases depending on what is selected by DECIDE. For instance, DECIDE can only select one of the following items.

   (a) $\vec{A}[\vec{c}/\vec{x}]$ from P'.

   (b) A fact F from P.

   (c) A rule R from P.

(d) $\forall \vec{x}. \vec{A} \to A$ from Π.

i. $|\vec{A}| = 0$

ii. $|\vec{A}| > 0$

We shall see that the cases (8a), (8b) and (8(d)i) are terminal, while the cases (8(d)ii) and (8c), whose necessary consequence is later use of $\to L$, alludes to induction.

9. The base of induction is that Lemma B holds when $\nu = 0$. Note that $\nu = 0$ iff every member $\Sigma'; P'; \Pi \Longrightarrow C$ of (SQT) is reduced in one of the cases (8a), (8b) and (8(d)i). In each case, the proof for the counterpart $\Gamma \mid \Delta \vdash \blacktriangleright(C[\vec{x}/\vec{c}])$ in (IFN) can be constructed. For instance, Case (8a) is similar to case (3a), but involves the **(Next)** rule. Case (8b) and (8(d)i) are both similar to case (3b), but the former of which involves the **(Next)** rule, while the latter of which involves the **(▶-Pres-$\forall_r$)** rule of Lemma 6.3.

10. We provide the inductive step, showing that if *Lemma B holds for* $\nu = n$ (the inductive hypothesis), then *Lemma B holds for* $\nu = n{+}1$. Without loss of generality, we assume that the first-time (out of $n + 1$ times) use of $\to L$ happens immediately after the DECIDE and $\forall L$ steps on $\Sigma'; P'; \Pi \Longrightarrow C_1$ of (SQT). Concerning the back-chaining formula associated with this first-time use of $\to L$, there are two cases, (8c) and (8(d)ii), discussed in (11) and (12) respectively.

11. In this case, CUP proceeds as follows. We reuse the typical forms of the selected rule R, its ground instance R', and guarding $\overline{R}$ given in (6), but without any interference with the usage of these symbols there.

$$\dfrac{\dfrac{\dfrac{\dfrac{C_1 \equiv B'}{\Sigma'; P'; \Pi \overset{B'}{\Longrightarrow} C_1}\text{INITIAL} \qquad \Sigma'; P'; \Pi \Longrightarrow \vec{B'}}{\Sigma'; P'; \Pi \overset{R'}{\Longrightarrow} C_1}\to L}{\Sigma'; P'; \Pi \overset{R}{\Longrightarrow} C_1}\forall L \ldots \forall L}{\Sigma'; P'; \Pi \Longrightarrow C_1}\text{DECIDE}$$

Since the total number of use of $\to L$ to prove all members of (SQT) is $n + 1$, we know that the total number of use of $\to L$ to prove members of (SQT$'$) is n. By the inductive hypothesis, we know that there is a proof for every member of (IFN$'$).

$$\{\Sigma'; P'; \Pi \Longrightarrow B'_i \mid B'_i \in \vec{B'}\} \cup \{\Sigma'; P'; \Pi \Longrightarrow C_j \mid 2 \le j \le \eta\} \qquad \text{(SQT}'\text{)}$$

$$\{\Gamma \mid \Delta \vdash \blacktriangleright(B'_i[\vec{x}/\vec{c}]) \mid B'_i \in \vec{B'}\} \cup \{\Gamma \mid \Delta \vdash \blacktriangleright(C_j[\vec{x}/\vec{c}]) \mid 2 \le j \le \eta\} \qquad \text{(IFN}'\text{)}$$

This implies that $\Gamma \mid \Delta \vdash \blacktriangleright(C_1[\vec{x}/\vec{c}])$ in (IFN) can also be proved, as follows.

$$\cfrac{\cfrac{\cfrac{}{\Gamma \mid \Delta \vdash \overline{R}}\,\textbf{(Proj)}}{\Gamma \mid \Delta \vdash \blacktriangleright(\vec{B}'[\vec{x}/\vec{c}]) \to B'[\vec{x}/\vec{c}]}\,(\forall\textbf{-E})\ldots(\forall\textbf{-E}) \qquad \Gamma \mid \Delta \vdash \blacktriangleright(\vec{B}'[\vec{x}/\vec{c}])}{\Gamma \mid \Delta \vdash B'[\vec{x}/\vec{c}]}\,(\to\textbf{-E})$$
$$\heartsuit$$

$$\cfrac{\cfrac{\heartsuit \qquad B'[\vec{x}/\vec{c}] \equiv C_1[\vec{x}/\vec{c}]}{\Gamma \mid \Delta \vdash C_1[\vec{x}/\vec{c}]}\,\textbf{(Conv)}}{\Gamma \mid \Delta \vdash \blacktriangleright(C_1[\vec{x}/\vec{c}])}\,\textbf{(Next)}$$

We now have shown all members of (IFN) can be proved.

12. In this case, CUP proceeds as follows.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{A' \equiv C_1}{\Sigma'; P'; \Pi \overset{A'}{\Longrightarrow} C_1}\,\text{INITIAL} \qquad \Sigma'; P'; \Pi \Longrightarrow \vec{A}'}{\Sigma'; P'; \Pi \overset{\vec{A}' \to A'}{\Longrightarrow} C_1}\,\to L}{\Sigma'; P'; \Pi \overset{\forall \vec{x}.\, \vec{A} \to A}{\Longrightarrow} C_1}\,\forall L \ldots \forall L}{\Sigma'; P'; \Pi \Longrightarrow C_1}\,\text{DECIDE}}{}$$

Since the total number of use of $\to L$ to prove all members of (SQT) is $n+1$, we know that the total number of use of $\to L$ to prove members of (SQT$''$) is n. By the inductive hypothesis, we know that there is a proof for every member of (IFN$''$).

$$\{\Sigma'; P'; \Pi \Longrightarrow A'_i \mid A'_i \in \vec{A}'\} \cup \{\Sigma'; P'; \Pi \Longrightarrow C_j \mid 2 \le j \le \eta\} \qquad \text{(SQT}''\text{)}$$

$$\{\Gamma \mid \Delta \vdash \blacktriangleright(A'_i[\vec{x}/\vec{c}]) \mid A'_i \in \vec{A}'\} \cup \{\Gamma \mid \Delta \vdash \blacktriangleright(C_j[\vec{x}/\vec{c}]) \mid 2 \le j \le \eta\} \qquad \text{(IFN}''\text{)}$$

This implies that $\Gamma \mid \Delta \vdash \blacktriangleright(C_1[\vec{x}/\vec{c}])$ in (IFN) can also be proved, as follows.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma \mid \Delta \vdash \blacktriangleright(\forall \vec{x}.\, \vec{A} \to A)}\,\textbf{(Proj)}}{\Gamma \mid \Delta \vdash \forall \vec{x}.\, \blacktriangleright(\vec{A} \to A)}\,(\blacktriangleright\textbf{-Pres-}\forall_r)}{\Gamma \mid \Delta \vdash \blacktriangleright(\vec{A}'[\vec{x}/\vec{c}] \to A'[\vec{x}/\vec{c}])}\,(\forall\textbf{-E})\ldots(\forall\textbf{-E})}{\Gamma \mid \Delta \vdash \blacktriangleright(\vec{A}'[\vec{x}/\vec{c}]) \to \blacktriangleright(A'[\vec{x}/\vec{c}])}\,\textbf{(Mon)}$$
$$\heartsuit$$

$$\cfrac{\cfrac{\heartsuit \qquad \Gamma \mid \Delta \vdash \blacktriangleright(\vec{A}'[\vec{x}/\vec{c}])}{\Gamma \mid \Delta \vdash \blacktriangleright(A'[\vec{x}/\vec{c}])}\,(\to\textbf{-E}) \qquad \blacktriangleright(A'[\vec{x}/\vec{c}]) \equiv \blacktriangleright(C_1[\vec{x}/\vec{c}])}{\Gamma \mid \Delta \vdash \blacktriangleright(C_1[\vec{x}/\vec{c}])}\,\textbf{(Conv)}$$

We now have shown that all members of (IFN) can be proved.

The proof is complete. □

We have now established the soundness of CUP with respect to $\mathbf{iFOL}_{\blacktriangleright}$. So far, we have introduced the following about CUP. CUP proves a Horn-clause-like formula with respect to a program of the same kind of formulae. First it asserts the goal as an extra assumption. Then, after breaking down the goal to an atom, and resolving it with an assumption from the program, CUP freely chooses any assumption until all sub-goals are resolved. The proven formula is true with respect to the complete greatest fixed-point of the program. The entire CUP proof is subsumed by a coinductive intuitionistic logic called $\mathbf{iFOL}_{\blacktriangleright}$. CUP provides a framework for coinductive first-order Horn clause logic programming.

Next, we compare CUP with related systems, such as μMALL and the Abella prover.

Chapter 7

Related Work

CUP, Cyclic Proof [23] and the Abella prover [24] form a closely related trio of systems. I regard CUP and Cyclic Proof as dual to each other: one for coinduction, the other for induction. Furthermore, I found that CUP and Cyclic Proof bear close technical resemblance to the implementation of, respectively, coinduction and induction, in Abella, though the difference is not trivial. μMALL and Abella share the same principle in their approaches to (co)induction, but I found that understanding μMALL is helpful for understanding Abella. So we first compare CUP with μMALL in §7.1. Then, we move on to compare CUP with Abella in §7.2. Cyclic Proof and its relation with CUP and Abella are discussed in §7.4.

7.1 μMALL and CUP

μMALL [25] is a proof system that supports induction and coinduction. It extends the Multiplicative and Additive fragment of Linear Logic (MALL), with 1) formula-level fixed-points, such as $\mu X.P$ and $\nu X.P$, where P is a first-order formula, and 2) inference rules for such fixed-points. Existing literature on μMALL [25–28] indicates that the notion of *formula-level fixed-point* originates from the concepts of definitional reflection and completion axiom [29], and that the related inference rules are based on Tarski's lattice-theoretic fixed-point theorem [30].

§7.1.1 gives details about how exactly formula-level fixed-points in μMALL are justified. §7.1.2 shows how the fixed-point inference rules in μMALL are derived from Tarski's theorem. These accounts arose as I tried to understand what is μMALL and how it is related to CUP, so these two sections can be omitted if the reader is already familiar with μMALL.

I noticed that there is a correspondence between formulae involved in CUP and those in μMALL. Such a correspondence could help discovering coinductive invariants for CUP proofs, since μMALL gives precise specification about what properties an invariant shall have. We discuss about these in §7.1.3. We finish by pointing out, in §7.1.4, about what future work can be done on comparing CUP with μMALL.

7.1.1 Formulating predicates as fixed-points

Suppose a predicate p is defined in a first-order Horn clause logic program [13] as:

$$p(t_{11}, \dots, t_{1m}) \Leftarrow C_1.$$
$$\vdots$$
$$p(t_{n1}, \dots, t_{nm}) \Leftarrow C_n.$$

where C denotes the body of a Horn clause. The *completion axiom* [29, 31] says:

$$\begin{aligned} &(\forall x_1 \dots x_m)\ p(x_1, \dots, x_m) \text{ iff} \\ &((\exists \vec{y_1})\ x_1 = t_{11} \otimes \dots \otimes x_m = t_{1m} \otimes C_1) \\ &\oplus \\ &\vdots \\ &\oplus \\ &((\exists \vec{y_n})\ x_1 = t_{n1} \otimes \dots \otimes x_m = t_{nm} \otimes C_n) \end{aligned} \tag{CA}$$

where $\vec{y_i}$ denotes all variables that are free in i-th clause of p, and $\otimes$ and $\oplus$ are respectively multiplicative conjunction and additive disjunction in linear logic.

The completion axiom provides a way to define every predicate as a (least or greatest) fixed-point: we abbreviate (CA) as "$p(x_1, \dots, x_m)$ iff D", then, regarding D as a simply typed lambda term, we can express p as $\lambda\vec{x}.D$, and we can further abstract away all (possibly none) free occurrences of p from $\lambda\vec{x}.D$, so that p is now expressed as $((\lambda y \lambda\vec{x}.D)\ p)$, indicating that p is a fixed-point of $\lambda y \lambda\vec{x}.D$. We write

$p = \mu(\lambda y \lambda \vec{x}.D) = \mu y.\lambda \vec{x}.D$ to mean p is the least fixed-point, and use ν instead of μ to mean the greatest fixed-point.

A logic program can have four fixed-points— least and greatest, each of which has two cases: infinite terms allowed or disallowed. Infinite terms are involved in productive perpetual (i.e. non-terminating) computation in logic programming [13, ch.6]. Coinductive uniform proof is designed to work with both finite and infinite terms but μMALL syntax does not allow infinite terms. So the μ and ν notation in μMALL should be understood as denoting fixed-points that do not involve infinite terms, corresponding to least and greatest Herbrand models on a universe of finite terms.

Example 7.1. Consider the program that defines the predicate q:

$$q(a).$$

$$q(s(X)) \Leftarrow q(X).$$

We can write q in the form of a fixed-point. First we instantiate the completion axiom using the definition of q:

$$\begin{aligned} (\forall Y)\ q(Y) \text{ iff} \\ & Y = a \\ & \oplus \\ & (\exists X)\ Y = s(X) \otimes q(X) \end{aligned}$$

We write the above expression in a single line, omitting the top level universal quantification ($\forall Y$), and we consider formulae on both sides of "iff" as simply typed lambda terms (cf. [25, §1.1]):

$$q\ Y \text{ iff } Y = a \oplus (\exists X.\ Y = s\,X \otimes q\,X)$$

We then abstract away both Y and q from the right side, so

$$q \text{ iff } \lambda Y.\ Y = a \oplus \exists X.\ Y = s\,X \otimes q\,X$$

then

$$q \text{ iff } (\lambda Q \lambda Y.\ Y = a \oplus \exists X.\ Y = s\,X \otimes Q\,X)\ q$$

Using B_q to denote

$$\lambda Q \lambda Y.\ Y = a \oplus \exists X.\ Y = s\,X \otimes Q\,X$$

and regarding "iff" as a sense of equality, we could see that "q iff $B_q\ q$", meaning that q is a fixed-point of B_q.

B_q is an operator on predicates: it takes a predicate then returns a predicate. If we provide some predicate r as input for B_q, then the returned predicate (call it r') would be

$$\lambda Y.\ Y = a \oplus \exists X.\ Y = s\,X \otimes r\,X$$

so that $r'(y)$ is, after β-reduction, the formula

$$y = a \oplus \exists X.\ y = s\,X \otimes r\,X$$

meaning that $r'(y)$ is true iff $y = a$ or there exist X such that $r(X)$ is true and $y = s(X)$.

For instance, if $r = \{b, c\}$, i.e. we define r to be true on (and only on) constants b and c, then $r' = \{a, s(b), s(c)\}$. We could see that since $B_q(r) = r' \neq r$, r is not a fixed-point of B_q.

In the particular case of B_q, its least fixed-point μB_q and greatest fixed-point νB_q coincide:

$$\mu B_q = \nu B_q = \{a, s(a), s(s(a)), s(s(s(a))), \ldots\}$$

We should note the ambiguities related to regarding q as a fixed-point of B_q.

1. The completion axiom allows us to regard q as a fixed-point of the operator B_p, but no clue is given about which fixed-point of B_q is q. Up to us, we can either decide that q is μB_q, or that is νB_q.

2. The process of expressing q in terms of a fixed-point of B_q loses the information about the name of q. Suppose we also have

$$r(a).$$

$$r(s(X)) \Leftarrow r(X).$$

   We cannot distinguish fixed-point representations of r and of q, which are the same modulo α-equivalence.

7.1.2 Using the fixed-point theorem for inference rules

We show a process in which the μMALL rules regarding least and greatest fixed-points are derived from Tarski's fixed-point theorem.

Let X be a set, then the power set 2^X of X equipped with the subset relation $\subseteq$ is a *complete lattice*. If $F : 2^X \mapsto 2^X$ is a *monotonic* function, then according to Tarski's theorem, F has a least fixed-point μF which is the intersection of all sets S such that $F(S) \subseteq S$ (such an S is called a *pre-fixed-point* of F), and F has a greatest fixed-point νF which is the union of all sets S such that $S \subseteq F(S)$ (such an S is called a *post-fixed-point* of F).

As a corollary of Tarski's theorem, we have

$$\begin{cases} F(S) \subseteq S & \Rightarrow \quad \mu F \subseteq S \\ S \subseteq F(S) & \Rightarrow \quad S \subseteq \nu F \end{cases} \tag{E1}$$

Using the definition of $\subseteq$, that $A \subseteq B$ iff $\forall x(A \ni x \Rightarrow B \ni x)$, we can rewrite (E1) as

$$\begin{cases} [\forall x(F(S) \ni x \Rightarrow S \ni x)] & \Rightarrow \quad [\forall y(\mu F \ni y \Rightarrow S \ni y)] \\ [\forall x(S \ni x \Rightarrow F(S) \ni x)] & \Rightarrow \quad [\forall y(S \ni y \Rightarrow \nu F \ni y)] \end{cases} \tag{E2}$$

Writing the two occurrences of top level $\Rightarrow$ in (E2) as rules, and omitting the quantifier $\forall$, we reshape (E2) as two rules:

$$\begin{cases} \dfrac{F(S) \ni x \Rightarrow S \ni x}{\mu F \ni y \Rightarrow S \ni y} \\ \\ \dfrac{S \ni x \Rightarrow F(S) \ni x}{S \ni y \Rightarrow \nu F \ni y} \end{cases} \tag{E3}$$

For every set S, there is an associated predicate P_S such that for all x, $P_S(x)$ is true iff $S \ni x$. Similarly, we can associate a *predicate operator* O_F to each *set operator* F, such that $O_F(P_A) = P_B$ iff $F(A) = B$. Given $F(A) = A$, meaning that set A is a fixed-point of F, we say $P_A = \mu O_F$ if A is the least fixed-point, and we say $P_A = \nu O_F$ if A is the greatest fixed-point. Using this notation and rewriting $\Rightarrow$ as $\vdash$, we can reshape (E3) as

$$\begin{cases} \dfrac{O_F\ P_S\ x \vdash P_S\ x}{\mu O_F\ y \vdash P_S\ y} \\ \\ \dfrac{P_S\ x \vdash O_F\ P_S\ x}{P_S\ y \vdash \nu O_F\ y} \end{cases} \tag{E4}$$

We can simplify the notation in (E4) by identifying the set name A with the predicate name P_A, just using A for both, and similarly we use F for the name O_F. This yields a pair of primitive fixed-point rules (E5), see also [26, p.15].

$$\begin{cases} \dfrac{F\ S\ x \vdash S\ x}{\mu F\ y \vdash S\ y}\ \mu' \\ \\ \dfrac{S\ x \vdash F\ S\ x}{S\ y \vdash \nu F\ y}\ \nu' \end{cases} \tag{E5}$$

We then build the proof-theoretically important *cut* rule into the fixed-point rules in (E5). In this process we will instantiate the universally bound variable y by a n-tuple of terms $\vec{t}$.

$$\dfrac{\dfrac{\dfrac{F\ S\ x \vdash S\ x}{\mu F\ y \vdash S\ y}\ \mu'}{\mu F\ \vec{t} \vdash S\ \vec{t}} \qquad \Gamma, S\ \vec{t} \vdash \Delta}{\Gamma, \mu F\ \vec{t} \vdash \Delta}\ \text{Cut}$$

$$\dfrac{\dfrac{\dfrac{S\ x \vdash F\ S\ x}{S\ y \vdash \nu F\ y}\ \nu'}{S\ \vec{t} \vdash \nu F\ \vec{t}} \qquad \Gamma \vdash S\ \vec{t}, \Delta}{\Gamma \vdash \nu F\ \vec{t}, \Delta}\ \text{Cut}$$

The fixed-point rules of interest derived from above are (E6), where x must be a new symbol, in order to express that the sequent $F\ S\ x \vdash S\ x$ (resp. $S\ x \vdash F\ S\ x$) represents the general property of the predicate S of being a pre-fixed (resp. post-fixed) point of F.

$$\begin{cases} \dfrac{F\ S\ x \vdash S\ x \qquad \Gamma, S\ \vec{t} \vdash \Delta}{\Gamma, \mu F\ \vec{t} \vdash \Delta}\ \mu L \\ \\ \dfrac{S\ x \vdash F\ S\ x \qquad \Gamma \vdash S\ \vec{t}, \Delta}{\Gamma \vdash \nu F\ \vec{t}, \Delta}\ \nu R \end{cases} \tag{E6}$$

The remaining fixed-point rules are based on the fact that if $F(A) = A$, then we can freely interchange the statement $A \ni x$ with $F(A) \ni x$ for any x.

$$\begin{cases} \dfrac{\Gamma \vdash F(\mu F)\vec{t}, \Delta}{\Gamma \vdash \mu F\ \vec{t}, \Delta}\ \mu R \\ \dfrac{\Gamma, F(\nu F)\vec{t} \vdash \Delta}{\Gamma, \nu F\ \vec{t} \vdash \Delta}\ \nu L \end{cases} \tag{E7}$$

7.1.3 Relating μMALL and CUP

There is a correspondence between the formulae involved in a CUP proof and those in a μMALL proof. Suppose we have $D_p \looparrowright D'_p$ where D_p is a set of Horn clauses coinductively defining some predicate p, and D'_p a conjunction of Horn clauses that *redefine* p. Correspondingly we could have a μMALL proof starting with

$$\dfrac{\vdash S\ \vec{t} \qquad S\ x \vdash B_p\ S\ x}{\vdash \nu B_p\ \vec{t}}\ \nu R$$

where νB_p is the greatest fixed-point notation of p derived from D_p, and S is the least fixed-point notation of (the redefined) p derived from D'_p. Under this correspondence, if $D_p \looparrowright D'_p$ then the least fixed-point of D'_p is also a post-fixed-point subsumed by the greatest fixed-point of D_p. This agrees with the soundness theorem of CUP. We could use μMALL as a clue when searching coinductive invariants for CUP.

Example 7.2. Consider the clause

$$r(X) \Leftarrow r(s(X))$$

which we understand coinductively (i.e. as the greatest fixed-point). The predicate r can then be reformulated as the greatest fixed-point νB_r of the predicate operator B_r:

$$\lambda z \lambda x. \exists y.(x = y \otimes z(s\ y))$$

Restricting constant symbols to a, s[1], then μB_r denotes any predicate that is true for no value, while νB_r represents any predicate that is true over the set $\{a, s(a), s(s(a)), \ldots\}$.

We observe a μMALL proof of $\vdash \nu B_r\ a$ and compare it with coinductive uniform proofs related to r.

[1]This restriction is solely for convenience, since our μMALL proof shown later still holds with an extended set of constants.

$$\frac{\vdash S\ a \qquad S\ x \vdash B_r\ S\ x}{\vdash \nu B_r\, a}\ \nu R \tag{P1}$$

The νR rule asks us to look for a predicate S, and the properties that S shall satisfy are given in terms of the two premises $\vdash S\ a$ and $S\ x \vdash B_r\ S\ x$. The meaning of $\vdash S\ a$ is that S holds specifically for a. Next, we have

$$B_r\ S := \lambda x.\exists y.(x = y \otimes S(s\ y))$$

so the predicate $(B_r\, S)$ is such that it takes an input term x, and decides whether there is a term y that *both* equals to the input x *and* makes $(s\ y)$ satisfy predicate S. In other words, $(B_r\ S\ x)$ holds iff $(S\ (s\ x))$ holds. So the second premise says that for all x, if $S\ x$ holds, then $S(s\ x)$ holds.

We have two groups of candidate predicates for S. The first group consists of all predicates that are defined by two Horn clauses as the predicate q in Example 7.1. For instance, we can choose from any one of $q_1, q_2, q_3, \ldots$ below to instantiate S in (P1).

$$q_1(a). \qquad q_2(a). \qquad q_3(a).$$

$$q_1(s(X)) \Leftarrow q_1(X). \qquad q_2(s(X)) \Leftarrow q_2(X). \qquad q_3(s(X)) \Leftarrow q_3(X).$$

All predicates $q_1, q_2, q_3 \ldots$ in the group can be represented as the same fixed-point μB_q. So we can let S in (P1) be μB_q, and it can be shown that the resulting premises $\vdash \mu B_q\ a$ and $\mu B_q\ x \vdash B_r\ (\mu B_q)\ x$ can be proved.

$$\cfrac{\cfrac{\cfrac{\vdash a = a}{\vdash a = a \oplus \exists X.\ a = s\,X \otimes (\mu B_q)\,X}\ \oplus R}{\vdash B_q(\mu B_q)\ a}\ \text{i.e.}}{\vdash \mu B_q\ a}\ \mu R$$

$$\cfrac{\cfrac{\mu B_q\ x \vdash x = x \otimes (\mu B_q)(s\ x)}{\mu B_q\ x \vdash \exists y.(x = y \otimes (\mu B_q)(s\ y))}\ \exists R}{\mu B_q\ x \vdash B_r\ (\mu B_q)\ x}\ \text{i.e.}$$

To prove $\mu B_q\ x \vdash (\mu B_q)(s\ x)$ we use the μR rule first.

Our second group of candidate predicates for S in (P1) consists of all predicates $p_1, p_2, \ldots$ defined in the form $\forall X.p(X)$, which are

$$\forall X.p_1(X). \quad \forall X.p_2(X). \quad \ldots$$

All these predicates $p_1, p_2, \ldots$ can be represented by the same fixed-point formula μB where B is $\lambda z \lambda x. \exists y. x = y$. So we can instantiate S in (P1) by μB, then the premises $\vdash \mu B\ a$ and $\mu B\ x \vdash B_r\ (\mu B)\ x$ can both be proved.

From the above analysis we could see that μMALL works with anonymous predicates, and two predicates are not distinguishable when they are defined in the same pattern and have the same (co)inductive interpretation. In this sense μMALL is *name insensitive*. In the contrary, we will see that coinductive uniform proof is *name sensitive*.

Corresponding to the μMALL proofs studied above, the following sequents are derivable by coinductive uniform proof [2]:

$$\forall X. r(X) \Leftarrow r(s(X)) \quad \looparrowright \quad r(a) \wedge \forall X. r(s(X)) \Leftarrow r(X)$$

$$\forall X. r(X) \Leftarrow r(s(X)) \quad \looparrowright \quad \forall X. r(X)$$

μMALL abstracts away the name r from both sides of $\looparrowright$, but CUP requires the predicate names be present and relevant on both sides of $\looparrowright$. For instance, in CUP we cannot prove

$$\forall X. r_1(X) \Leftarrow r_1(s(X)) \quad \looparrowright \quad \forall X. r_2(X)$$

but in μMALL, r_1 is mapped to the anonymous predicate νB_r and when r_2 is used to instantiate S in (P1), it is also mapped to the anonymous predicate μB.

Since μMALL does not allow infinite terms, a CUP proof may not have a related μMALL proof. For instance, on the one hand, we can derive in CUP

$$\forall X. r(X) \Leftarrow r(s(X)) \quad \looparrowright \quad \forall X. r(s(X)) \Leftarrow r(X) \tag{P2}$$

On the other hand, suppose there is a system called "μMALL II" obtained by extending μMALL by enabling the infinite term s^ω (which denotes $s(s(\cdots))$ so that $s^\omega = s(s^\omega)$), and using ν' to denote greatest fixed-points that contain infinite terms. Our CUP sequent (P2) is related to the following tentative "μMALL II" derivation of $\vdash \nu' B_r\ s^\omega$:

$$\frac{\vdash S\ s^\omega \qquad S\ x \vdash B_r\ S\ x}{\vdash \nu' B_r\ s^\omega}\ \nu' R \tag{P3}$$

[2]To prove the first sequent we have to extend CUP with an auxiliary right-focusing rule for $\wedge$. The coinductive soundness of the CUP system so extended is believed to hold by the author, but a formal proof of this is left as future work.

where rule $\nu' R$ is obtained from νR of μMALL by simply replacing ν by ν'. We can then use the infinite-term-allowed greatest fixed-point of $r(s(X)) \Leftarrow r(X)$ to instantiate S in (P3).

Going further with the μMALL II derivation (P3) is not without problems. After all, s^ω is not allowed in μMALL, so there is indeed some mismatch between CUP and μMALL caused by their different attitudes to infinite terms.

7.1.4 Conclusion

We made some informal but concrete comparison between μMALL and CUP, highlighting that the conclusion of a CUP sequent corresponds to the post-fixed-point in μMALL's greatest-fixed-point rule (νR). It should be of theoretical interest to formally establish this correspondence, which we leave as future work.

μMALL was successfully applied to encode finite state automata and to reason about automata inclusion [28]. Since automata can also be encoded in Horn clauses, we could investigate whether CUP can also be used for similar tasks.

7.2 Abella and CUP

Abella is an interactive theorem prover featuring higher-order abstract syntax, reasoning capacity for object-level binding structures, induction and coinduction. These features are developed and integrated through a series of intermediate systems. Coinduction implemented in Abella closely resembles CUP. Coinduction in both systems feature a COFIX-like rule, and a guarding scheme. But there is a difference in the guarding scheme, so that the same coinductive hypothesis simultaneously provided to Abella and CUP can sometimes be viewed as defining totally different post-fixed-points by the two systems. We review the history of technical development of Abella in §7.2.1, and discuss the similarities and differences between coinduction in Abella and that in CUP in §7.2.2.

7.2.1 Line of work leading to the Abella prover

The theorem prover Abella [24] is an implementation of the logic $\mathcal{G}$. The series of evolving logic systems that led to $\mathcal{G}$ is summarized in Figure 7.1.

All systems in Figure 7.1 have the cut elimination property. Features possessed

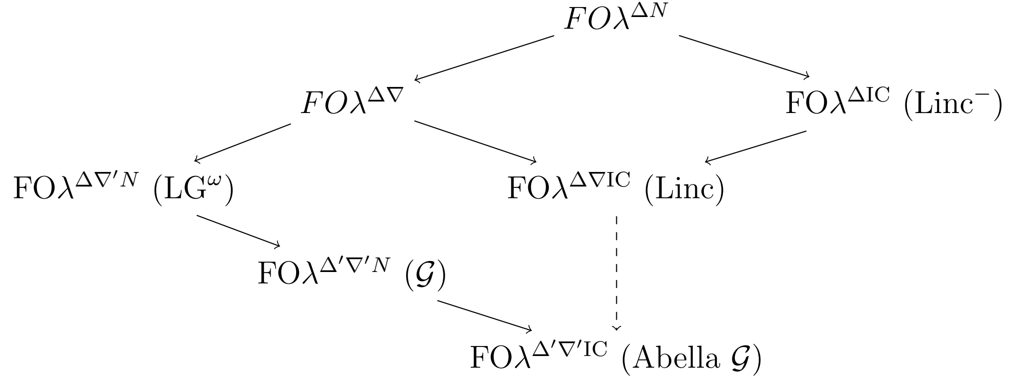


Figure 7.1: Dependency diagram for logic systems that lead to the logic $\mathcal{G}$ underlying the Abella Prover. To highlight differences among systems, following the naming convention as used in $FO\lambda^{\Delta N}$, Linc^{-} is renamed as $\mathrm{FO}\lambda^{\Delta \mathrm{IC}}$, and Linc is renamed as $\mathrm{FO}\lambda^{\Delta\nabla \mathrm{IC}}$, etc.

by a system are indicated by a combination of "feature codes" which constitutes the system name. Feature codes are drawn from the following pool.

FOλ — Intuitionistic logic with quantification over high-order non-predicate types.

N — Natural number induction.

IC — General principles of induction and coinduction.

∇ — Basic ∇ (read "nabla") quantifier.

∇' — Strengthened ∇ quantifier by extra axioms.

Δ — Basic definition, not involving the ∇ quantifier.

Δ' — Extended definition, involving the ∇ quantifier.

All systems share the same core, which is an extension of intuitionistic logic with 1) quantification over high-order non-predicate types, and 2) a proof-theoretic notion of definition.

$FO\lambda^{\Delta N}$(read "fold-n") [32–34] tackled the design challenge of making higher-order quantification and definition compatible with induction. Linc^{-} (read "link minus") [35, 36] is an extension of $FO\lambda^{\Delta N}$ with general principles of induction and coinduction. A second, independent extension of $FO\lambda^{\Delta N}$ is $FO\lambda^{\Delta\nabla}$(read "fold-nabla") [37] which introduces the ∇ quantifier to reason about dynamics of bindings in object systems. Linc [38] primitively combines the features ∇ and general

principles of induction and coinduction, suffering an inadequate interplay between (co)induction and the ∇ quantifier, causing some inductive properties involving binding to be not provable.

To address the problem exposed by Linc, the LG^{ω} system [39] successfully integrates ∇ quantifier with natural number induction. The key is additional axioms for ∇ which give this quantifier a natural semantics so that induction can now be used to prove specifications involving ∇. So far, a definition is still not allowed to involve the ∇ quantifier, thus limiting the usefulness of the systems. Extension in this direction is done in $\mathcal{G}$ [40], which allows definitions to contain ∇. Further extension of $\mathcal{G}$ [41, 42] replaces natural number induction by general principles of (co)induction, so now the features of ∇, *definition* and *(co)induction* are fully integrated with each other.

In $Linc^{-}$ [35] (where it is called Linc), coinduction has a side condition imposed by the proof technique that establishes cut elimination. This side condition can be removed, as shown in [36], by adopting an alternative proof technique. Stratification of definitions has been required in all systems until [36] shows that this is not necessary at least in the absence of ∇.

7.2.2 Comparing coinduction in Abella with CUP

Coinduction in Abella[3] is implemented via the `coinduction` tactic and a *guarding scheme* of +/# annotations. The `coinduction` tactic resembles the COFIX rule of CUP. The +/# guarding scheme resembles the $\langle\cdot\rangle$ guarding scheme of CUP. We identified two aspects of difference between the two systems concerning coinduction. The one that is obvious, and less important, is that Abella allows multiple applications of the `coinduction` tactic to construct a proof, while CUP only allows COFIX be applied once, and uses the conservative model extension theorem to simulate nested coinduction. Beyond this, there is *a deeper difference*, which concerns the guarding schemes.

We first give some examples where Abella faithfully mimics CUP which may give an inaccurate impression that Abella subsumes CUP. We then give an example that highlights their important difference on guarding schemes. This example is followed by a uniform (albeit incomplete) formalization of the guarding schemes

[3]The Abella system used in this review is version 2.0.5.

of both systems, *in the +/# terminology of Abella*, thus highlighting the difference. The conclusion is that, despite not allowing multiple uses of COFIX, the guarding scheme of CUP is more liberal than that of Abella.

We start with setting up the basic inductive definition of natural numbers.

```
Kind nat type.           | Define nt : nat -> prop by
Type z nat.              |  nt z;
Type s nat -> nat.       |  nt (s A) := nt A.
```

Here are two examples where Abella mimics CUP:

- ```
  CoDefine tom : nat -> prop by
  tom X := tom (s X) /\ nt X.

  Theorem tom_true : forall x, nt x -> tom x.
  coinduction. intros. unfold. backchain CH. search.
  ```
- ```
  CoDefine tree : nat -> prop by
  tree X := tree (s X).

  Theorem tree_true : forall y, tree y.
  coinduction. intros. unfold. backchain CH.
  ```

The guarding scheme difference between Abella and CUP is exposed by the following observation. We want to prove the theorem `omega` in Abella:

```
Theorem omega : forall x, tree x -> tree (s x).
```

We apply a series of tactics *that mimic the same way in which CUP approaches the theorem*, starting with `coinduction`:

```
coinduction. intros. unfold.
```

After `unfold`, the state of the proof is:

```
Variables: x
CH : forall x, tree x -> tree (s x) +
H1 : tree x
============================
tree (s (s x)) +
```

We now `backchain CH`, then the state of the proof becomes:

```
Variables: x
CH : forall x, tree x -> tree (s x) +
H1 : tree x
============================
tree (s x)
```

Now, the response of CUP to the above state is to `backchain CH` again, then complete the proof. We let Abella do so:

```
omega < backchain CH.
Error: Coinductive restriction violated
```

Abella indicates that we cannot use the hypothesis `CH` on the goal. This is where CUP and Abella differ. Actually, if we express COFIX using the annotation based guarding scheme of Abella, the coinductive hypothesis asserted by COFIX is:

```
COFIX  CH : forall x, tree x + -> tree (s x) +
```

which is different from that asserted by Abella:

```
ABELLA CH : forall x, tree x -> tree (s x) +
```

The difference lies in COFIX's extra `+` mark on the recursive call in the hypothesis body. Lack of this extra mark results in that Abella cannot follow CUP step-by-step to complete the proof.

In below we formulate the COFIX rule using Abella's annotation method, and contrast it with the `coinduction` tactic. We find that the annotation method of Abella can be more succinctly formulated if we place the mark closer to the predicate of the atom that is being marked, as a superscript, rather than as a superscript (or post-fix) at the end of the atom, which is done by Gacek [41, p.66]. For instance, instead of writing $p\ \vec{t}_+$ and $p\ \vec{t}_\#$, we now write $p^+\ \vec{t}$ and $p^\#\ \vec{t}$, resp.

$$\frac{\Sigma; \Gamma, \forall \vec{x}.\ B\ p^+\ \vec{x} \Rightarrow p^+\ \vec{t}\ \vdash\ \forall \vec{x}.\ B\ p\ \vec{x} \Rightarrow p^\#\ \vec{t}}{\Sigma; \Gamma\ \vdash\ \forall \vec{x}.\ B\ p\ \vec{x} \Rightarrow p\ \vec{t}}\ \text{COFIX}$$

In comparison, Abella's `coinduction` tactic is:

$$\frac{\Sigma; \Gamma, \forall \vec{x}.\ B\ p\ \vec{x} \Rightarrow p^+\ \vec{t}\ \vdash\ \forall \vec{x}.\ B\ p\ \vec{x} \Rightarrow p^\#\ \vec{t}}{\Sigma; \Gamma\ \vdash\ \forall \vec{x}.\ B\ p\ \vec{x} \Rightarrow p\ \vec{t}}\ \texttt{coinduction}$$

We see that when a formula $\forall \vec{x}.\ B\ p\ \vec{x} \Rightarrow p\ \vec{t}$ is asserted as a hypothesis, CUP annotates both the head predicate p and all (if any) recursive occurrences of p in the body $B\ p\ \vec{x}$, hence the hypothesis $\forall \vec{x}.\ B\ p^{+}\ \vec{x} \Rightarrow p^{+}\ \vec{t}$. Abella only annotates the head predicate p and left all predicates in the body $B\ p\ \vec{x}$ unmarked, hence the hypothesis $\forall \vec{x}.\ B\ p\ \vec{x} \Rightarrow p^{+}\ \vec{t}$.

Recall that the coinduction principle in $\mathcal{G}$ (which is the same as in μMALL [27]) says that to prove an atom $p\ \vec{t}$, where p is a coinductively defined predicate, we need to find a post-fixed-point S and show that $S\ \vec{t}$ is true. The difference in guarding scheme between Abella and CUP can be explained by understanding the formula that acts as the coinduction hypothesis, as a *definition* of a post-fixed-point. For instance, if we rename the predicate `tree +` as `r`, then the post-fixed-point `r` as seen by Abella is:

```
CoDefine r : nat -> prop by
r (s X) := tree X.
```

While the `r` as seen by CUP is:

```
CoDefine r : nat -> prop by
r (s X) := r X.
```

The `r` as seen by Abella is indeed a post-fixed-point of the predicate operator determined by the coinductive definition of `tree`. Abella can actually complete the proof by definitional reflection. The `r` as seen by CUP is also a post-fixed-point, which defines $\emptyset$, and which can also define $\{s^{\omega}\}$ when we allow infinite terms. We know that there is no infinite term in the syntax of $\mathcal{G}$, but CUP works with a metric-theoretically complete universe of terms that allows structures like s^{ω}. By different annotation/guarding schemes, Abella and CUP are soundly proving different post-fixed-points that arise from the same coinductive goal.

7.2.3 Conclusion

As a future task, it remains to establish a detailed and formal relationship between a μMALL style coinduction rule (as that of $\mathcal{G}$) and a CUP style coinduction rule (as that implemented in Abella). Andrew Gacek [41, p.66] claims that a CUP style proof amounts to show that a predicate is a post-fixed-point, fulfilling a premise that arises

from using the μMALL style coinduction rule in a $\mathcal{G}$ proof. His view is supported by our observations on comparing CUP proofs and μMALL proofs. However, important details are missing in his account in order to formally substantiate the claim.

7.3 Discussion: post-fixed-points vs. coinductive goals

Proving that an atom A is in the coinductive model of a logic program using CUP can be regarded as using the μMALL style coinduction rule at the meta-level: we first build a finite success sequent-tree in CUP with respect to some suitable coinductive goal, and then extract a post-fixed-point from it which contains A using the method provided by the formal soundness proof of CUP. For instance, in the setting of Example 5.35, if we want to use the νR rule of Equation E6 (on page 95) to show that $q\ (s\ z)$ is coinductively true, we would need to find a post-fixed-point of the logic program that contains $q\ (s\ z)$. Since we have a proof of $\forall x.\, q\ x \rightarrow q\ (s\ z)$ in CUP from which we can extract the desired post-fixed-point, we could say that the premise of the νR rule is satisfiable, and then establish the conclusion.

A tactic, call it `cup`, can therefore be suggested for implementation in the future in interactive theorem provers. The tactic `cup` shall behave according to the following description. To prove that some goal is coinductively true, the user can call the tactic and provide an argument to it which is a suitable coinductive goal. For example, imagine that there is only one goal ψ to be proved under a set S of assumptions, and the user types the command "`cup` φ". Now the system creates two sub-goals: one is φ and the other is ψ, both having the assumption $S \cup \{\varphi\}$. The system is expecting an interactive session of proving φ according to CUP rules and restrictions, followed by a session of proving ψ. If both sub-goals are provable, then the proof is completed. The `cup` tactic shall be justifiable by the soundness theorems of CUP (i.e., Theorem 5.42 and 5.46).

7.4 Cyclic Proof, CUP and Abella

We briefly introduce the notion of cyclic proof in §7.4.1. Then in §7.4.2 we see an example that suggests, that cyclic proof is a proof theoretic formulation, of the

`induction` tactic in the theorem prover Abella. This in turn suggests, that Abella's justification of its `induction` tactic, supports the *cyclic proof conjecture*, albeit restricted from a classical setting to an intuitionistic one. Finally, in §7.4.3, we suggest to regard Cyclic Proof as an inductive dual system of Coinductive Uniform Proof.

7.4.1 Introducing cyclic proof

Cyclic proof, is a proof theoretic generalization, of the infinite descent method, that is known as an alternative to mathematical induction. The core theory of cyclic proof was developed by James Brotherston and Alex Simpson in 2005 [23] and remains stable since then, and has been published in several formats later on [43–45]. More recent work on this system is about its implementation [46] and applications [47–50].

Technically, the rules of cyclic proof consist of classical first-order logic sequent rules, left and right introduction rules for inductive definitions (i.e. definitional reflection of [29]), and an extra structural rule of substitution:

$$\frac{\Gamma \vdash \Delta}{\Gamma[\Theta] \vdash \Delta[\Theta]}\ \text{(Subst.)}$$

Cyclic proof is interpreted w.r.t the least Herbrand model of inductive definitions. For example, given a definition $\forall y,\ p\ y := p\ (s\ y)$ ($:=$ reads "if"), we can prove $p\ x \vdash \bot$ by the following cyclic proof:

$$\dfrac{\dfrac{\dfrac{\underline{p\ x} \vdash \bot \quad (*)}{\underline{p\ (s\ y)} \vdash \bot}\ \text{(Subst.)}}{p\ y,\ \underline{p\ (s\ y)} \vdash \bot}\ \text{(Weakening)}}{\underline{p\ x} \vdash \bot \quad (*)}\ \text{(Case } p\text{)}$$

where the pair of $(*)$ indicates a cycle in the proof. The above cyclic proof is alternative to an inductive proof that involves an explicit pre-fixed-point (cf. [41, p.34]). Soundness of cyclic proofs depends on the existence of an infinitely progressing trace of atomic formulae, for every infinite path of sequents in the proof tree. Formulae in the trace are drawn from the LHS of $\vdash$ following a certain method. Progress only happens when a defined predicate is unfolded by its definition on the LHS of $\vdash$. The above cyclic proof only has one infinite path, where there is a periodical and infinitely progressing trace with the repeating pattern $p\ x$, $p\ (s\ y)$, $p\ (s\ y)$, $p\ x$ (i.e. the upward going series of atoms underlined in the proof) where progress is made at the step from $p\ x$ to $p\ (s\ y)$ which corresponds to the use of (Case p).

7.4.2 Relating cyclic proof to Abella's induction

Cyclic proof is closely related to the `induction` tactic in the theorem prover Abella [24]. Taking for example the cyclic proof in §7.4.1, the final steps of capping the derivation by using the substitution rule (Subst.) and then forming a back link to the root sequent, corresponds to the Abella operations of first declaring $p\ x \Rightarrow \bot$ as an inductive hypothesis and later applying it, as follows [4].

```
Kind i type.                  Define p : i -> prop by
Type s  i -> i.               p X := p (s X).
```

We define the inductive predicate p, such that it has an empty model, and we state this fact as a theorem `pFalse`.

```
Theorem pFalse : forall x, p x -> false.
```

The Abella proof has four steps (1) – (4).

(1)

```
pFalse < induction on 1.

IH : forall x, p x * -> false
============================
forall x, p x @ -> false
```

(2)

```
pFalse < intros.

Variables: x
IH : forall x, p x * -> false
H1 : p x @
============================
false
```

(3)

```
pFalse < case H1.

Variables: x
IH : forall x, p x * -> false
H2 : p (s x) *
============================
false
```

(4)

```
pFalse < apply IH to H2.
Proof completed.
```

Step (2) corresponds to the opening state of the cyclic proof. The use of (Case p) in the cyclic proof corresponds to step (3), while step (4) echoes (Subst.). Finally, formation of the back link corresponds to step (1).

[4] The Abella system used in this review is version 2.0.5.

There are more induction examples to be explored in order to more precisely formulate the relationship between cyclic proof and the `induction` tactic of Abella. However, such exploration would lead us out of the scope of this thesis, which focuses on coinduction. Therefore, we draw some conjectures based on what we have observed, without further attempt to verify them.

We conjecture, that the `induction` tactic of Abella implements cyclic proof in an intuitionistic setting. In other words, we conjecture that:

1. An inductive proof in Abella, using the `induction` tactic, is a cyclic proof.

2. An intuitionistic cyclic proof is trivially an inductive proof in Abella.

Following this view, we further suppose that, when using the `induction` tactic, checking for existence of progressing traces can be safely omitted, since the $*/@$-annotation guarantees existence of such traces.

Here is an application of our conjecture. Gacek justified the `induction` tactic, by translating it into inductive $\mathcal{G}$ proof steps [41, §5.2]. Given the connection between the notions of a *structural proof* and an *inductive $\mathcal{G}$ proof* [5], Gacek's account should be helpful for Brotherston and Simpson to prove their *cyclic proof conjecture* [23, Conjecture 4.16] [45, Conjecture 7.7], that if there is a cyclic proof of $\Gamma \vdash \Delta$ then there is a structural proof of $\Gamma \vdash \Delta$.

7.4.3 Conclusion

We use Table 7.1 as an agenda for our following discussion.

[45] gives two inductive proof systems that formalizes infinite descent: 1) LKID$^\omega$, where a proof could be non-terminating, to represent infinite descent in a straight forward way, and 2) CLKID$^\omega$ (a.k.a cyclic proof) where all proofs are finite, representing all terminating or periodical LKID$^\omega$ constructs. Therefore LKID$^\omega$ is conceptually more fundamental than CLKID$^\omega$. We add pre-fix "i-" to denote their intuitionistic fragment.

Based on some primitive observation, we have supposed that i-CLKID$^\omega$ resembles ($\approx$) the `induction` tactic of Abella. We have compared elsewhere, the connection between CUP and the `coinduction` tactic of Abella, both of which are intuitionistic. Further, we observe that the i-CLKID$^\omega$ and CUP are symmetric across the

[5] A structural proof [23] (An inductive $\mathcal{G}$ proof [41]) is a *classical* (resp. *intuitionistic*) proof with definitional reflection and induction.

Induction	**Coinduction**
i-CLKID$^\omega$ $\approx$ `induction` tactic	CUP $\approx$ `coinduction` tactic
i-LKID$^\omega$	"Infinite CUP"

Table 7.1: Supposed duality among (co)inductive systems.

sequent arrow ($\vdash$): a i-CLKID$^\omega$ proof exhibits infinite unfolding of (inductively) defined predicates on the LHS of $\vdash$, while a CUP proof encodes infinite unfolding of (coinductively) defined predicates on the RHS of $\vdash$. In these senses, we suggest to regard i-CLKID$^\omega$ as a dual to CUP, as indicted in Table 7.1.

Now that we have regarded CUP as a dual to i-CLKID$^\omega$, we may wonder, that there should be a coinductive dual for i-LKID$^\omega$. There is no such a system exist in the literature, and we tentatively call it "Infinite CUP" in Table 7.1. It could be studied in the future. However, given the already extensive studies of perpetual processes from the logic programming community, this proposed "Infinite CUP" may merely be a proof-theoretic rendering of what has been known about non-terminating logic programs.

Chapter 8

Conclusion and Future work

8.1 Summary

We summarize the thesis by providing short answers to the research questions asked in Section 1.1.

1. Re perpetual computation in logic programming:

   (a) What kind of logic programs can produce infinite data, whenever its SLD derivation is non-terminating ?

   Ans. *Logic programs that are both observationally productive and universal (OP&U).*

   (b) Under what circumstances can we use unification, between goals in a potentially non-terminating SLD derivation, to decide non-termination of that derivation ?

   Ans. *The circumstance where we are working with OP&U logic programs, and an atom t from a goal unifies an atom t' from a previous goal and t is no less general that t'.*

   (c) When can the infinite data generated by a non-terminating SLD derivation be the same as that computed by unification between goals in the derivation? And in such cases, why are the results the same ?

   Ans. *In the circumstances described above, and additionally when the circular unifier is grounding; because the infinite amount of answer substitutions involved in the SLD derivation coincide with the unfolding of the circular unifier.*

2. Re coinduction in Horn clause logic:

   (a) What could be a coinductively sound logic that could prove some irregular trees in the greatest complete Herbrand model of a first-order Horn clause logic program?

   Ans. *Coinductive Uniform Proof.*

8.2 Future work

Having answered the research questions of the thesis, here is a listing of some tasks of interest, which may deserve future effort.

1. The existing method of post-fixed point extraction for CUP suffers *the restriction* that the coinductive goal must be a single H-formula, although a *conjunction* of H-formulae is allowed by the definition of CUP. If the coinductive goal is a single H-formula, it can only use itself as the coinductive hypothesis, whereas if it is a conjunction of H-formulae, one conjunct may use other conjuncts as coinductive hypotheses, and such dynamics is out of scope for the existing post-fixed point extraction method. Solving this problem can put CUP at a stronger position in proof theory as a computational counterpart to Tarskian fixed-point approaches to coinduction such as μMALL. Soundness proof in the more complex case could generalize the existing proof in terms that an EVS-index is no longer a word but a list of triples, where a triple (i, j, k) denotes the δ-substitution created when selecting the i-th coinductive hypothesis for back-chaining in the sequent sub-tree of the j-th conjunct for the k-th time.

2. Implement CUP as a tactic for some theorem prover that has a relatively large user base.

3. Use an interactive theorem prover to certify the soundness proof for CUP.

4. Formally establish the relationship between a μMALL-style coinduction rule and a CUP-style coinduction rule.

5. Extend the language of CUP from H-formula to hereditary Harrop language.

Appendix A

Tarski's Fixed-Point Theorem

Tarski's fixed-point theorem [13, §5] and its proof are given here.

Definition A.1. A *binary relation* on a set S is any subset of $S \times S$.

Definition A.2. A binary relation R (on a set S) is *reflexive* if xRx for all $x \in S$. It is *anti-symmetric* if xRy and yRx implies $x = y$ for all $x, y \in S$. It is *transitive* if xRy and yRz implies xRz for all $x, y, z \in S$.

Definition A.3. A binary relation R is a *partial order* if it is reflexive, transitive and anti-symmetric.

Example A.4. On the power set 2^S of any set S, the subset relation $\subseteq$ is a partial order. The less-than-or-equal relation $\leq$ on $\mathbb{N}$ is a partial order.

Notation A.5. From now on we use the symbol $\leq$ to denote an arbitrary partial order.

Definition A.6. Let S be a set with a subset X, and $\leq$ a partial order on S. The element $a \in S$ is an *upper bound* of X if for all $x \in X$ we have $x \leq a$. The element $b \in S$ is a *lower bound* of X if for all $x \in X$ we have $b \leq x$. Furthermore, $a' \in S$ is the *least upper bound* of X if a' is an upper bound of X and, for all upper bounds a of X we have $a' \leq a$. Similarly, $b' \in S$ is the *greatest lower bound* of X if b' is a lower bound of X and, for all lower bounds b of X we have $b \leq b'$.

Notation A.7. We use $lub(X)$ and $glb(X)$ to denote respectively the least upper bound and greatest lower bound of X.

Definition A.8. Let $\leq$ be a partial order on S. We call S a *complete lattice* if for every subset X of S, there exist both $lub(X)$ and $glb(X)$.

Example A.9. The power set 2^S of any set S, equipped with the subset relation $\subseteq$, is a complete lattice. For every $X \subseteq 2^S$, $lub(X) = \bigcup X$ and $glb(X) = \bigcap X$.

Definition A.10. Let the set S equipped with a partial order $\leq$ be a complete lattice. A function T from S to S, is *monotonic* if $x \leq y$ implies $T(x) \leq T(y)$ for all $x, y \in S$.

Assumption A.11. Let the set S equipped with a partial order $\leq$ be a complete lattice. Let the function T from S to S, be monotonic.

Definition A.12. Assume A.11. The element $x \in S$ is a *fixed-point* of T if $T(x) = x$. It is a *pre-fixed-point* of T if $T(x) \leq x$. It is a *post-fixed-point* of T if $x \leq T(x)$.

- The element $y \in S$ is the *least fixed-point* of T if y is a fixed-point of T and $y \leq x$ for every fixed-point x of T.
- The element $y \in S$ is the *greatest fixed-point* of T if y is a fixed-point of T and $x \leq y$ for every fixed-point x of T.

Lemma A.13. *Assume A.11. Let* $G = \{x \in S \mid x \leq T(x)\}$ *and* $g = lub(G)$ *and* $g' = lub\{x \in S \mid T(x) = x\}$. *Then,*

1. *The least upper bound of all post-fixed-points is a post-fixed-point, i.e.,* $g \in G$.
2. *The least upper bound of all post-fixed-points is a fixed-point, i.e.,* $T(g) = g$.
3. *The least upper bound of all post-fixed-points equals the least upper bound of all fixed-points, i.e.,* $g = g'$.

Proof. 1. Since $x \leq g$ for all $x \in G$, by monotonicity of T, we have $T(x) \leq T(g)$ for all $x \in G$. Then, by transitivity of $\leq$ we have $x \leq T(g)$ for all $x \in G$. Therefore, $T(g)$ is an upper bound of G. Then $g \leq T(g)$ for g is the least upper bound. So g is a post-fixed-point and $g \in G$.

2. Given $g \leq T(g)$, it remains to show $T(g) \leq g$, so that we can have $T(g) = g$ by anti-symmetry of $\leq$. Starting with $g \leq T(g)$, we have $T(g) \leq T(T(g))$ by monotonicity of T, which means that $T(g)$ is a post-fixed-point of T. Therefore $T(g) \in G$. Since g is the least upper bound of G, we have $T(g) \leq g$. Then, $T(g) = g$.

3. Since g is a fixed-point of T, and g' is the least upper bound of all fixed-points of T, we have $g \leq g'$. On the other hand, since $\{x \in S \mid T(x) = x\} \subseteq \{x \in S \mid x \leq T(x)\}$, we have $g' \leq g$. From $g' \leq g$ and $g \leq g'$ we derive $g = g'$ by anti-symmetry of $\leq$.

□

Lemma A.14. *Assume A.11. Let $H = \{x \in S \mid T(x) \leq x\}$ and $h = glb(H)$ and $h' = glb\{x \in S \mid T(x) = x\}$. Then,*

1. *The greatest lower bound of all pre-fixed-points is a pre-fixed-point, i.e., $h \in H$.*
2. *The greatest lower bound of all pre-fixed-points is a fixed-point, i.e., $T(h) = h$.*
3. *The greatest lower bound of all pre-fixed-points equals the greatest lower bound of all fixed-points, i.e., $h = h'$.*

Proof. Similar to the proof of Lemma A.13. □

Theorem A.15 (Tarski)**.** *Assume A.11. Then, the least fixed-point of T equals the greatest lower bound of all fixed-points of T, which also equals the greatest lower bound of all pre-fixed-points of T. Moreover, the greatest fixed-point of T equals the least upper bound of all fixed-points of T, which also equals the least upper bound of all post-fixed-points of T. In equations:*

$$lfp(T) = glb\{x \in S \mid T(x) = x\} = glb\{x \in S \mid T(x) \leq x\}$$

$$gfp(T) = lub\{x \in S \mid T(x) = x\} = lub\{x \in S \mid x \leq T(x)\}$$

Proof.

1. By definition, the least fixed-point of T is a fixed-point of T as well as a lower bound of the set of all fixed-points of T. These requirements are satisfied by the greatest lower bound of all pre-fixed-points following Lemma A.14.

2. By definition, the greatest fixed-point of T is a fixed-point of T as well as an upper bound of the set of all fixed-points of T. These requirements are satisfied by the least upper bound of all post-fixed-points following Lemma A.13.

□

References

[1] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In Sandro Etalle and Mirosław Truszczyński, editors, *Logic Programming*, pages 330–345, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[2] Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers, and Andrew Pond. Proof relevant corecursive resolution. In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming*, pages 126–143, Cham, 2016. Springer International Publishing.

[3] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[4] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014.

[5] Xinyu Feng. Mechanized verification of preemptive os kernels (invited talk). In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 2–2, New York, NY, USA, 2017. ACM.

[6] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

[7] Yves Bertot and Ekaterina Komendantskaya. Inductive and coinductive components of corecursive functions in coq. *Electronic Notes in Theoretical Computer Science*, 203(5):25 – 47, 2008. Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).

[8] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for isabelle/hol. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 93–110, Cham, 2014. Springer International Publishing.

[9] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *Fundamentals of Computation Theory*, pages 274–287, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[10] Dean Voets and Danny De Schreye. Non-termination analysis of logic programs using types. In María Alpuente, editor, *Logic-Based Program Synthesis and Transformation*, pages 133–148, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[11] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, New York, NY, USA, 1st edition, 2014.

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[13] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[14] Alain Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A Tarnlund, editors, *Logic Programming*, volume 11, pages 231–251. Academic Press, Inc., London, UK, 1982.

[15] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1):125 – 157, 1991.

[16] James Harland. *On Hereditary Harrop Formulae as a basis for Logic Programming*. PhD thesis, The University of Edinburgh, 1991.

[17] Yue Li. Structural resolution with co-inductive loop detection. In *Proceedings of the First Workshop on Coalgebra, Horn Clause Logic Programming and Types, CoALP-Ty 2016, Edinburgh, UK, 28-29 November 2016.*, pages 52–67, 2016.

[18] Ekaterina Komendantskaya, Patricia Johann, and Martin Schmidt. A productivity checker for logic programming. In Manuel V Hermenegildo and Pedro Lopez-Garcia, editors, *Logic-Based Program Synthesis and Transformation*, pages 168–186, Cham, 2017. Springer International Publishing.

[19] Keith L. Clark. *Predicate Logic as a Computational Formalism*. PhD thesis, University of London, London, UK, 1980.

[20] Henning Basold, Ekaterina Komendantskaya, and Yue Li. Coinduction in uniform: Foundations for corecursive proof search with horn clauses. In Luís Caires, editor, *Programming Languages and Systems*, pages 783–813, Cham, 2019. Springer International Publishing.

[21] Yue Li. Coinductive uniform proof. Technical report, Heriot-Watt University, Available at `http://www.macs.hw.ac.uk/~yl55/UnPublished/CUP-TR18.pdf` 2018.

[22] Henning Basold. *Mixed Inductive-Coinductive Reasoning: Types, Programs and Logic*. PhD thesis, Radboud University Nijmegen, 2018.

[23] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proceedings of TABLEAUX-14*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

[24] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

[25] David Baelde. *A linear approach to the proof-theory of least and greatest fixed points*. PhD thesis, Ecole Polytechnique, France, 12 2008.

[26] Amina Doumane. *On the infinitary proof theory of logics with fixed points*. PhD thesis, Paris Diderot University, France, 2017.

[27] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In Nachum Dershowitz and Andrei Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'07)*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 92–106, Yerevan, Armenia, October 2007. Springer.

[28] David Baelde. On the proof theory of regular fixed points. In Martin Giese and Arild Waaler, editors, *Proceedings of the 18th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX'09)*, volume 5607 of *Lecture Notes in Artificial Intelligence*, pages 93–107, Oslo, Norway, July 2009. Springer.

[29] Peter Schroeder-Heister. Rules of definitional reflection. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 222–232, June 1993.

[30] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[31] Peter Schroeder-Heister. Definitional reflection and the completion. In Roy Dyckhoff, editor, *Extensions of Logic Programming*, pages 333–347, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[32] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 434–445, June 1997.

[33] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232(1):91 – 119, 2000.

[34] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. Comput. Logic*, 3(1):80–136, January 2002.

[35] Alberto Momigliano and Alwen Tiu. Induction and co-induction in sequent calculus. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 293–308, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[36] Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic*, 10(4):330 – 367, 2012. Selected papers from the 6th International Conference on Soft Computing Models in Industrial and Environmental Applications.

[37] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Logic*, 6(4):749–783, October 2005.

[38] Alwen Tiu. *A Logical Framework for Reasoning About Logical Specifications*. PhD thesis, The Pennsylvania State University, 2004. AAI3141024.

[39] Alwen Tiu. A logic for reasoning about generic judgments. *Electronic Notes in Theoretical Computer Science*, 174(5):3 – 18, 2007. Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006).

[40] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 33–44, 2008.

[41] Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.

[42] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48 – 73, 2011.

[43] James Brotherston. *Sequent Calculus Proof Systems for Inductive Definitions*. PhD thesis, University of Edinburgh, November 2006.

[44] James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *Proceedings of LICS-22*, pages 51–60. IEEE Computer Society, July 2007.

[45] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, December 2011.

[46] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. A generic cyclic theorem prover. In *Proceedings of APLAS-10*, LNCS, pages 350–367. Springer, 2012.

[47] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, pages 101–112. ACM, 2008.

[48] James Brotherston, Dino Distefano, and Rasmus L. Petersen. Automated cyclic entailment proofs in separation logic. In *Proceedings of CADE-23*, LNAI, pages 131–146. Springer, 2011.

[49] Reuben N. S. Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *Proceedings of CPP-6*, pages 53–65. ACM, 2016.

[50] Gadi Tellez Espinosa and James Brotherston. Automatically verifying temporal properties of programs with cyclic proof. In *Proceedings of CADE-26*, volume 10395 of *LNAI*, pages 491–508. Springer, 2017.

Index of Definitions

ancestor-set, 43

arity

  of function symbol, 12

  of higher-order variable, 21

  of type, 21

  of variable, 12

atom, 15

atom-value, 31

back-chained atom

  principal, 65

back-chaining formula, 65

back-chaining sequent

  principal, 65

β-reduction, 24

circular, 17

circular substitution

  application of, 46

clause, 19

closed formula, 31

co lemma, 78

coinductive hypothesis, 53

coinductive uniform proof, 54

combined reduction, 24

composition, 17

composition effect, 69, 71

computable at infinity, 38

context, 21

convertible, 24

CUP, 54

δ-substitution, 66

distance, 15

EVS-index, 72

eigen-variable, 71

  list of, 71

eigen-variable substitution, 72

  application of, 72

eigen-variable substitution index, 72

eigen-variable substitution system, 72

fact, 57

finite,infinite

  atom, 15

  term, 13

  tree language, 11

firmly supported, 66

first-order

  λ-term, 27

  formula, 31

  term, 15

fix-reduction, 24

fix-beta reduction, 28

fixed-point, 19

  least, greatest, 19

  post-, 19, 62

pre-, 19
formula, 31
free variable, 13
fresh, 45
cumulatively, 45
function symbol, 12
`nil`, `cons`, `suc`, 12

GI-disjoint, 78
goal, 35
annotated, 44
ground
atom, 15
instance, 19, 61
term, 13
grounding, 46
guarded λ-term, 26
guarded atom, 30
guarded fixed-point, 25
guarding, 83

H-formula, 33
H-goal, 52
H-program, 52
Herbrand base, 19
complete, 19
Herbrand interpretation, 19
complete, 19
Herbrand operator, 19
complete, 19
Herbrand universe, 19
complete, 19
higher-order
λ-term, 27
formula, 31
higher-order variable, 21
Horn$^{\omega}$ clause, 60
Horn$^{\omega}$ program, 61
Horn clause, 19
body, 19
head, 19

identity substitution, 17
immediate consequence operator, 61
instance, 19
ground, 19
intended infinite term, 29

λ-term, 23
level, 14
logic program, 19

matcher, 18
model, 20, 62

node-hopping effect, 66, 73
nominally supported, 66
non-trivial, 39, 58
strongly, 39

observationally productive, 38
occur-check, 18
ω-substitution, 61
order, 21

pre-substitution, 16
predicate, 12
proof, 54

reduction, 35
annotated, 44
renaming, 17
rewriting reduction, 36

annotated, 44

rule, 57

sequent, 53

signature, 12

SLD derivation, 38

SLD resolution reduction, 36

structural derivation, 38

annotated, 45

structural resolution reduction, 36

annotated, 45

substitution, 16

application of, 17

for λ-term, 24

of eigen-variable, 72

substitution reduction, 36

annotated, 44

term, 13

term-value, 30

approximate, 28

tree language, 11

trivial, 58

truncation, 14

type, 20

type-I guarded λ-term, 26

type-II guarded λ-term, 26

unfolding, 46

unification, 18

unifier, 18

of derivation, 38

universal, 40

variable, 12

variable-disjoint, 14, 35

variant

of clause, 19

of term, 17

word, 11

length, 11

Index of Notation

$wi\ wj$, 11

$\odot$, 72, 73

$\theta(\mathbf{t})$, 17

$arity(f)$, 12

$\Leftarrow$, 47

ar(), 21

A_Σ, 31

$\mathbf{GAtom}^*(\Sigma)$, 15

$\mathbf{Atom}^\omega(\Sigma)$, 15

$\mathbf{Atom}(\Sigma)$, 15

$\mathbf{Atom}^\infty(\Sigma)$, 15

$\vec{c}$, 72

$(t, \mathbf{t}) \in P$, 19

$\models\approx$, 78

$\Gamma\ \Gamma'\ \Gamma_1$, 21

$\Gamma_{\mathbb{T}}$, 22

$\Gamma^n_{\mathbb{T}}$, 22

$\vec{c} \mapsto \vec{t}$, 72

$\diamond$, 14

$d(t, u)$, 15

$domain(t)$, 13

$\equiv$, 24, 31

$1st\mathbf{GTerm}^\omega(\Sigma)$, 15

$1st\mathbf{Term}(\Sigma)$, 15

$\longrightarrow_{\mathbf{fix}\beta}$, 28

$\longrightarrow_{\mathrm{Fix}\beta}$, 28

FV, 14

$\vdash_g$, 26

$\vdash_{\mathrm{I}}$, 26

$\vdash_{\mathrm{II}}$, 26

$\Sigma; \varnothing \vdash_{\triangleright} M : \tau$, 25

$x : \pi$, 22

$\mathcal{T}_P$, 19

$\mathcal{T}_P^\omega$, 19

$\forall \vec{x}. \vec{A} \to A$, 54

H_Σ, 33

$\mathbf{iFOL}_{\blacktriangleright}$, 80

$\Im(P)$, 61

$\Im(\varphi)$, 61

$\dot{\Im}$, 28

$\Im(S)$, 73

$\Im$, 30, 31

$A_i \in \vec{A}$, 54

Λ_Σ, 23

Λ_Σ^G, 27

$length(w)$, 11

$[n_1, \ldots, n_k]$, 11

$t \sqsupseteq_\theta u$, 18

$\mathbb{N}$, 10

$\mathbb{N}^*$, 11

ϵ, 11

ord(), 21

Π, 12

$\Gamma \mid \Delta \vdash \varphi$, 81

$\vdash$, 81

$\mathbf{t} \to_\theta \mathbf{t}'$, 36

$\mathbf{t} \to \mathbf{t}'$, 36

$\mathbf{t}_0 \to^n \mathbf{t}_n$, 36

$\mathbf{t} \hookrightarrow_\theta \mathbf{t}'$, 36

$\longrightarrow_\beta$, 24

$\longrightarrow$, 24

$\longrightarrow_{\mathbf{fix}}$, 24

$\Sigma; P \looparrowright \varphi$, 53

$\Sigma; P; \Delta \Longrightarrow \langle\varphi\rangle$, 53

$\Sigma; P; \Delta \Longrightarrow \varphi$, 53

$\Sigma\ \Sigma'\ \Sigma_1$, 12

$\theta\ \sigma$, 16

$\Sigma_{\mathbb{P}}$, 22

$\Sigma^n_{\mathbb{P}}$, 22

$\Sigma_{\mathbb{T}}$, 22

$\Sigma^n_{\mathbb{T}}$, 22

$\mathbf{t} \subseteq S$, 74

$[N/x]$, 24

$\{\mathbf{y} \mapsto \mathbf{u}\}$, 46

$\{\vec{\theta_1}(\mathbf{y}) \mapsto \vec{\theta_2}(\mathbf{u})\}$, 46

$M\,[N/x]$, 24

$\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, 17

$[N_1/x_1, \ldots, N_m/x_m]$, 54

$[\vec{N}/\vec{x}]$, 54

$\theta_k \cdots \theta_1(\mathbf{t})$, 17

$t\ u$, 13

$\mathbf{Term}^\omega$, 13

$\mathbf{Term}$, 13

$\mathbf{Term}^\infty$, 13

$t \in \mathbf{t}$, 13

$\mathbf{t}\ \mathbf{u}$, 13

$(\mathbf{t}_1, \mathbf{u}, \mathbf{t}_2)$, 13

$(\mathbf{t}_1, t, \mathbf{t}_2)$, 13

$(t_1, \ldots, t_n)$, 13

T_P, 61

$t|_n$, 14

$\to$, 20

$\mathbb{B}$, 20

ι, 20

$\tau\ \pi\ \rho$, 21

$\mathbb{P}$, 20

o, 20

$\mathbb{T}$, 20

$\Sigma; \Gamma \vdash_{(m;n)} M : \tau$, 23

$\Sigma; \Gamma \vdash^*_{(m;n)} M : \tau$, 23

$t \approx_\sigma u$, 18

$t \sim_\theta u$, 18

$\Sigma; \Gamma \Vdash_a \varphi$, 30

$\Sigma; \Gamma \Vdash \varphi$, 30, 80

Var, 12

Var', 21

$\vec{A}$, 54

$\blacktriangleright \vec{A}$, 83

$|\vec{x}|$, 24

$(\vec{c}, \Sigma)$, 56

$\lambda\vec{x} : \iota.\, N$, 24

$\vec{N}$, 24

$\vec{x}$, 24