Cronfa
Swansea University's Research Repository

Enhanced Realizability Interpretation for Program Extraction

Olga Petrovska

Submitted to Swansea University in partial fulfilment
of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

2021

Copyright: The author, Olga Petrovska, 2021.

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..[redacted]....... (candidate)

Date ....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..[redacted]. (candidate)

Date ....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..[redacted]. (candidate)

Date ....

Abstract

This thesis presents Intuitionistic Fixed Point Logic (IFP), a schema for formal systems aimed to work with program extraction from proofs. IFP in its basic form allows proof construction based on natural deduction inference rules, extended by induction and coinduction. The corresponding system RIFP (IFP with realizers) enables transforming logical proofs into programs utilizing the enhanced realizability interpretation. The theoretical research is put into practice in PRAWF[1], a Haskell-based proof assistant for program extraction.

[1] pronounced /prau̯v/, Welsh for 'Proof'

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Ulrich Berger, for all his support throughout the recent years. This thesis would not have been possible without his guidance, patience, and encouragement, which inspired me to strive forward.

I am thankful to my examiners, Prof. Achim Jung and Dr. Paulo Oliva, for our discussions during the viva and their words of encouragement. Their detailed and constructive feedback on my thesis was very appreciated.

My gratitude also goes to all the members of the Swansea University Computer Science Department, especially Monika Seisenberger and Anton Setzer for all their support. Our discussions were instrumental in my progress. I would also like to thank all my fellow PhD students and wish you the best of luck for your futures.

I was lucky to have met many interesting and inspiring researchers on my journey. I am grateful to Prof. Hideki Tsuiki for inviting us to Kyoto University, where I worked with him, my supervisor and Tsutomu Takayama, and achieved the first substantial breakthrough in my PhD. Prof. Hideki's encouraging words and explanations have been invaluable. I am grateful to Prof. Douglas Bridges and Dr. Hannes Diener for hosting us at the University of Canterbury, which was an experience of a lifetime for me. I am also grateful to Dr. Maria Maietti, whose encouragement helped me to believe in myself more.

My friends and family deserve endless gratitude for all the various ways in which they supported me. Octavia Efraim, thank you for being an inspiration and a kindred PhD spirit. We shared many moments of frustration but our conversations kept me grounded and able to move forward. Mark Hooley, thank you for being there for me in the hardest times, for all the kindness and encouragement. Your eccentric humour kept my spirits up. Louis Warren, thank you for being an inspiration for me. I really appreciated our discussions on mathematics, logic, and culture. Liam Kelly, thank you for eagerly listening to all my rants and for the super speedy and thorough proofreading. Elif Firat, thank you for making me feel that I am not alone and for all those Zoom sessions that we had to support each other.

My deepest gratitude goes to Ferdinand and Miroslava Veseli, thank you for supporting me and helping me to overcome all my visa issues. I deeply want to thank my parents, Igor and Iryna Petrovski, for providing the grounding that nurtured my adventurous spirit, preparing me to embrace the world and teaching me to be resilient against all the hardships.

Most importantly, I would like to thank to my husband, Ferdinand Vesely, who supported and encouraged me all the way, who heard my fears and concerns and always helped me find some sense amidst the confusion. I wouldn't have been able to do this without you. Thank you for all your love, belief in me and your endless patience.

Table of Contents

List of Figures xi

Acronyms xiii

1 Introduction 1
Related papers and talks . . . 2
Main contributions . . . 4
Structure of the thesis . . . 5

I Theoretical and historical background 7

2 Constructivism 9
2.1 Brouwer's intuitionism and its development . . . 10
2.2 Recursive constructive mathematics . . . 11
2.3 Bishop's constructive mathematics (BISH) . . . 12
2.4 Intuitionistic type theory . . . 13

3 Realizability 15
3.1 History and general aspects of realizability . . . 15
3.2 Realizability of induction and coinduction . . . 17

4 Lambda Calculus and its variations 19
4.1 Lambda calculus . . . 19
4.2 Simply typed lambda calculus . . . 21

5 Domain theory 23

6 Semantics 27
6.1 Operational semantics . . . 28
6.2 Axiomatic semantics . . . 28
6.3 Denotational semantics . . . 29

7 Harrop formulas 31

8 Program extraction 33

II Theoretical framework for program extraction 37

9 Formal system for program extraction 39
9.1 IFP 40
9.2 IFP' 43
9.3 Program semantics and RIFP 45
9.4 Harrop expressions 49
9.5 Realizability for program extraction 51

10 Soundness 61

11 Operational semantics and adequacy 111
11.1 Finiteness, totality and approximation 112
11.2 Big-step semantics 116
11.3 Adequacy for finite computations 119
11.4 Small-step semantics 124
11.5 Adequacy for infinite computations 129

III Program extraction in practice 133

12 Development of the proof system 135
12.1 General structure of Prawf 135
12.2 Logic component 139
12.3 Realizability and program extraction component 143
12.4 Execution component 145

13 Case studies 147

IV Conclusions 161
The journey 163
Summary 164
Future work 165

References 167

List of Figures

4.1 Typing rules for simply typed lambda calculus [19] 22

9.1 Expressions in IFP 41
9.2 Free variables in IFP 41
9.3 Definition of strict positivity 42
9.4 Inference rules of IFP 44
9.5 Induction and coinduction in IFP' 45
9.6 Definition of programs 47
9.7 Abbreviations for operations on programs [24] 47
9.8 Denotational semantics [24] 47
9.9 Definition of the Harrop property 50
9.10 Realizability interpretation 52

10.1 Monotone predicate transformers 70
10.2 Selected IFP' inference rules with realizers 110

11.1 Visualisation of the domain of realisers, the domain of data and its sub-domains 113

12.1 Intuitive structure of PRAWF 137
12.2 Generation of new goals through rule application 138
12.3 Order of goals 138

Acronyms

BHK Brouwer-Heyting-Kolmogorov interpretation. 15

BISH Bishop's Constructive Mathematics. 9

CoC Calculus of Constructions. 34

cpo Complete partial order. 24

dcpo Directed-complete partial order. 25

EC Execution Component. 136

gfp Greatest fixed point. 24

glb Greatest lower bound. 24

IFP Intuitionistic Fixed Point Logic. 40

IO Input/Output. 136

LC Logic Component. 135

LEM Law of Excluded Middle. 10

lfp Least fixed point. 24

MCICD Monotone and Clausalar Inductive and Coinductive Definitions. 17

MP Markov's principle. 11

RCM Recursive Constructive Mathematics. 9

RIFP Intuitionistic Fixed Point Logic with realizers. 45

RPEC Realizability and Program Extraction Component. 136

SOS Structural Operational Semantics. 28

TCF Theory of Computable Functionals. 18

Chapter 1

Introduction

Contents

Related papers and talks 2
Main contributions 4
Structure of the thesis 5

Logic in computer science has a multitude of applications. For instance, it can be used in modelling better circuits, characterising complexity classes, or describing processes. The focus of this thesis is on applications centred around the notion of a *formal logical proof* as a "string of symbols, which satisfy some precisely stated set of rules and which prove a theorem, which itself must also be expressed as a string of symbols" [56].

Proofs are used in computer science in three fundamentally different ways. Firstly, in program verification proofs are regarded as certificates for the correctness of programs. Secondly, proof search is used as a computation model in logic programming, for example in languages like Prolog or λProlog. Lastly, the Curry-Howard isomorphism allows us to interpret proofs as programs. This thesis focuses on the third one, within which we distinguish (a) the approach used in constructive type theory, where *proof normalisation or cut-elimination* of proofs is used for direct computation [7, 8, 9] and (b) *program extraction*, which can be based on (i) *functional interpretation*, (ii) *realizability for intuitionistic systems*, and (iii) *realizability for classical systems*

The core of functional interpretation, based on Kurt Gödel's *Dialectica interpretation* [59], lies in the reduction of the classical theory to its intuitionistic variant and then further into a quantifier-free theory of functionals of finite type. Gödel's original use of functional interpretation was to provide a consistency proof for Peano arithmetic. Functional interpretation enables extracting con-

structive content from proofs in the intuitionistic and classical theories. It is of interest to many scholars, including Jeremy Avigad, Solomon Feferman, Ulrich Kohlenbach, Paulo Oliva, Helmut Schwichtenberg, Henry Towsner, and others [5, 6, 73, 92, 93, 107, 119].

The second trend – program extraction through realiziability interpretation (for intuitionistic systems) – is at the core of this thesis. Realizability has its roots in the Brouwer-Heyting-Kolmogorov (BHK) interpretation and is usually associated with the names of Stephen Kleene and Georg Kreisel. Anne Troelstra was among the scholars exploring the connection between intuitionistic logical operators and realizability [43]. His work inspired his student, Jaap van Oosten to write a historic essay on realizability that serves as a good introduction into this field [94]. In this thesis we define an enhanced realizability interpretation, based on Kleene's version, which allows obtaining realizers from proofs that include induction and coinduction. The inspiration to use this particular approach comes from consideration of the theory behind the *Minlog* proof assistant [118]. This proof assistant is based on Kreisel's *modified realizability* [74]. In this thesis, we explore how using a different approach to realizability of induction and coinduction impacts the process of program extraction.

Within the "proofs as programs" paradigm restricting proofs to intuitionistic ones due to their constructive nature was common. Michael Parigot believed such a restriction was too strict because "from the programming point of view, constructivity is only needed for Σ^0_1-statements, for which classical and intuitionistic provability coincide" [96]. In this respect, classical proofs may also be considered to be programs as long as there is a system in which a program can be extracted directly from a classical proof, as opposed to translation of classical proofs into intuitionistic ones, and when there is an understanding of "the algorithmic meaning of classical constructions" [96]. His $\lambda\mu$-calculus enabled him to extend the concept of "proofs as programs" to work with classical proofs. Besides Parigot, also Jean-Louis Krivine introduced realizability for classical logic [75, 76, 77]. Parigot's and Krivine's approaches are both connected with Kleene/Kreisel realizability through Gödel's negative translation (embedding Peano arithmetic into Heyting arithmetic).

Publications and talks

The following papers are published versions of some of the materials presented in this thesis. They also contain some additional details related to this research.

Optimized Program Extraction for Induction and Coinduction [21] (by U. Berger and O. Petrovska) This paper presents the first variant of the Intuitionistic Fixed

Point logic (IFP), a formal system that serves as a base for program extraction. The paper also includes the proof of the Soundness Theorem, which is subject to a special admissibility condition. The updated version of the Soundness Theorem presented in this thesis, however, has the mentioned admissibility condition removed.

The paper presents a carefully crafted realizability interpretation based on the distinction between non-Harrop and Harrop formulas, where the latter ones are treated in a special manner. Program extraction based on this realizability interpretation is optimized as the programs that we get are simpler and do not include irrelevant non-computational content. Such optimization, however, comes at a price. The soundness proof becomes more involved and rigorous. Soundness would be easy if there would be no distinction between Harrop/non-Harrop expressions, which will be presented in section 9.4. However, then we would get wasteful programs with unnecessarily complex data types. For instance, a condition for integer division is that one cannot divide by 0. To extract an integer division program, we need to have a proof that employs an axiom, which states that the divisor must be nonzero. All axioms used in proofs are required to be Harrop and, as such, they bear no computational content. If there would be no distinction between Harrop and non-Harrop, then the extracted program would not only take dividend and divisor as its arguments, it would also require a proof that the dividend is not a zero to be executed successfully.

Additionally, this paper shows that well-founded induction is an instance of strictly positive induction. This enables us to derive a new computationally meaningful formulation of the Archimedean property for real numbers. Through an example of program extraction in computable analysis this paper illustrates that Archimedean induction can be used to eliminate countable choice. These materials and related research were presented at the Computability in Europe (CiE) conference in Kiel, Germany as well as at the Continuity, Computability, Constructivity (CCC) workshop in Faro, Portugal in 2018. The paper was published in the LNCS proceedings of the CiE 2018.

PRAWF*: An Interactive Proof System for Program Extraction* [22] (by U. Berger, O. Petrovska, H. Tsuiki) This paper presents an interactive proof system dedicated to program extraction from proofs. The implementation uses an updated version of IFP presented in [24]. This paper gives an overview of the prototype implementation and explains its use through several case studies. An example of conversion between different versions of exact real numbers representation utilizes Archimedean induction as presented in our previous paper. The proof assistant benefits from an improvement of the theory, namely introduction of an intermediate system IFP'. This enables extraction of programs from proofs avoid-

ing the admissibility condition, allowing proofs that use unrestricted strictly positive inductive and coinductive definitions. This improvement was introduced [24], where the corresponding soundness proof was outlined. The present thesis includes a proof of this Soundness Theorem worked out in detail. An overview of this paper and the new soundness proof were presented via Zoom at the 36th British Colloquium for Theoretical Computer Science (BCTCS) in Swansea, UK as well as at the CiE conference in Fisciano, Italy in 2020. The paper was published in the proceedings of the conference.

Main contributions

The following is a brief summary of the main contributions.

Proof of soundness. When a new logical system is introduced one needs to ensure that it is sound. Therefore, our first paper briefly describes the Soundness Theorem, connecting the proofs in IFP with their counterparts in RIFP. The most complicated part of the soundness proof were the cases for induction and coinduction. They turned out to be more complex than initially anticipated. Due to the distinction between non-Harrop formulas, which have computational content, and Harrop formulas, which do not, certain (admissibility) restrictions had to be put in place in order to be able to complete the soundness proof. The restrictions were not severe as in all practical applications proofs turned out to be admissible. However, to be completely confident that no cases were omitted, the aim was to find an alternative approach that would allow lifting this restriction. Fortunately, whilst working on [24], Hideki Tsuiki discovered the workaround through introduction of the intermediary system IFP'. This system is an exact copy of the original IFP with the (co)induction rules redefined. In IFP' the premise of (co)induction includes an additional proof of monotonicity of the relevant operator. Taking this into account, the Soundness Theorem has been redefined and instead of IFP it uses the link between IFP' and RIFP. The initial soundness proof was drafted by the authors in [24], although it was not worked out in detail. After the paper was released to the public, I verified that the Soundness Theorem in this new formulation works. The detailed proof is included in this thesis.

Alternative approach to small-step semantics. Operational semantics used in the previous research of Ulrich Berger related to IFP usually relied on the notion of closures. A closure is a pair, consisting of a program and a corresponding environment. The use of closures in operational semantics enables us to transform and evaluate a program easily, keeping track of the environment details, whilst

avoiding unnecessary intermediate substitutions. Whilst working on the implementation of operational semantics in PRAWF I discovered that the small-step semantics defined in the initial version of [24] had an issue. Specifically, the use of the inference rules could lead to potential loss of parts of the context in the process of evaluation. Therefore, in the released version of the paper the authors opted for inference rules that used substitution instead of closures. While that works from a theoretical point of view, in practice this approach is not as efficient as the approach with closures. Therefore, after reviewing the drawbacks in the initial definitions, I introduced the notion of an *extended closure* to solve the problem with the context inaccuracy. The big-step semantics remained unchanged, whilst the small-step inference rules were extended. The interchangeability between the big-step and small-step inference rules was proven.

PRAWF *implementation.* PRAWF is based on the proof assistant for propositional logic that I developed as a part of my MSc degree. The initial system allowed working with the basic natural deduction rules for first-order logic. The proof assistant was then extended with the new rules for induction, coinduction, (co)closure, as well as equivalence rules. New concepts of a language, a declaration and an axiom were introduced in this updated system. The notions of a predicate and a formula were extended and a new data type of operators was introduced. Aside from the purely logical aspect, the machinery for extracting programs from proofs and working with them was added. This included introduction of the notion of a program and implementation of realizability interpretation and operational semantics. The implementation was carried out alongside the development of the IFP as a theoretical system. Hence, PRAWF was constantly modified to match the theory, whilst the development of the theory benefited from the practical findings encountered during the implementation. The current version of PRAWF allows constructing proofs in IFP and extracting executable programs from them. From the usability perspective it may not be robust enough, considering that the theorem base is small, but the tool is sufficient enough to show that IFP does work in practice.

Structure of the thesis

This thesis consists of four parts.

The first part presents the theoretical and historical background for the given research. We begin with the introduction of constructivism and give an overview of its various trends, including Brouwer's intuitionism, Recursive constructive mathematics, Bishop's constructive mathematics, and Intuitionistic type theory.

Then we proceed to present realizability as the underlying concept in program extraction and draw specific attention to the realizability of induction and coinduction. To present the idea of computability, the following chapter outlines lambda calculus and its extension, simply typed lambda calculus. This is followed by an overview of domain theory and a chapter on various approaches to semantics, including operational, axiomatic, and denotational. The section on Harrop formulas presents them as a tool for separating useful computational content from computationally irrelevant content. The final chapter of this part gives a brief overview of program extraction in some of the most popular proof systems.

The second part presents the formal system for program extraction, called Intuitionistic Fixed Point logic (IFP). IFP is a base and an umbrella term for a number of formal systems (extensions of IFP), which are used for program extraction. We describe the logical system of IFP, its extension IFP', and, finally, RIFP, a version of IFP that includes realizers. Following this, the realizability interpretation for IFP is introduced. The subsequent chapter contains a detailed proof of the Soundness Theorem. The final chapter introduces the relevant operational semantics and includes an adjusted proof of the adequacy theorem, linking denotational and operational semantics.

The third part describes the development of the PRAWF system, highlighting the underlying structure, the main notions and their corresponding data types. It describes a number of small case studies based on proofs involving induction and coinduction. These case studies include walks-through from a proof to an extracted program and its execution. Although the examples are small, they are appropriately simple to present how programs are extracted at a reasonable level of detail. A more complex case study was done by Tsuiki and it is presented in [22].

The fourth part concludes this thesis, giving an overview of my PhD journey and an overall summary of the conducted research. It also describes the areas that can be explored and improved upon in the future.

Part I

Theoretical and historical background

Chapter 2

Constructivism

Contents

2.1 Brouwer's intuitionism and its development **10**

2.2 Recursive constructive mathematics **11**

2.3 Bishop's constructive mathematics (BISH) **12**

2.4 Intuitionistic type theory **13**

This chapter gives a general overview of constructivism and its main schools.

Formalization of mathematics in the beginning of the 20$^{\text{th}}$ century brought about "a highly idealized version of mathematical existence" [28]. David Hilbert's formalism was prevalent in the scientific circles at the time, so it was hard to popularise ideas that questioned classical mathematics. Luitzen Brouwer, however, being concerned about the defects in classical mathematics, endeavoured to set things right; his groundbreaking ideas led to the establishment of *intuitionism*.

Intuitionism can be viewed as not only one of the constructivist schools but also as a basis for constructivism. The main principles of intuitionism proposed by Brouwer in 1907 and 1908 [33, 34] also hold in *Recursive Constructive Mathematics* (RCM) and *Bishop's Constructive Mathematics* (BISH), both of which came about only in the middle of the 20$^{\text{th}}$ century. Brouwer's work was continued by his student Arend Heyting and other followers [35].

Each trend in constructivism has its own peculiarities but the idea that unites them is that a mathematical statement holds only if it is possible to produce an 'evidence' of that. Here we will briefly look at all the major constructive schools and point out their peculiarities.

2.1 Brouwer's intuitionism and its development

The underlying ideas of intuitionism were conceived in Brouwer's thesis "On the Foundations of Mathematics"[1] in 1907. One of the most prominent features was rejection of the *law of excluded middle* (LEM)[2]. The supporters of the formalism in mathematics used LEM when working with finite and infinite mathematical objects. The justification for this lies in their approach to infinite objects, which they viewed in terms of actual (completed) infinity. On the other hand, Brouwer looked at them through the prism of potential infinity and, therefore, saw no rational foundation in extending the use of LEM to infinite situations just because it worked for the finite objects. Interestingly, Brouwer still considered LEM to be correct at the time he wrote his thesis. Then he believed that $A \vee \neg A$ can be interpreted as $\neg A \to \neg A$:

> "A function is differentiable or is not differentiable says *nothing*; it expresses the same as the following: If a function is not differentiable, then it is not differentiable. But the logician, looking at the *words* of the former sentence, and discovering a regularity in the combination of words in this and in similar sentences, here again projects a mathematical system, and he calls such a sentence an *application of the tertium non datur*." [3]

Hilbert's program was based around formalization of mathematics, while Brouwer was rather sceptical about the use of formal language. His views revolved around on the concept that the essence of mathematics lies in mental constructions rather than the use of formal symbols and manipulation with them. In this way mathematics exists not within the scope of language but beyond it. The mathematical language of symbols could be a way to express concepts but it is not in the core of mathematics. In fact, unlike other mathematicians, Brouwer chose descriptive writing style for expressing his ideas over symbolic notation whenever possible. Although, some may argue that this style makes it harder for his readers to grasp the concepts straight away, it also celebrates the core idea

[1] Original title in Dutch: "Over de Grondslagen der Wiskunde"

[2] For every statement A, either A or $\neg A$ holds.

[3] "Een functie is òf differentieerbaar òf niet differentieerbaar zegt *niets*; drukt hetzelfde uit, als het volgende: Als een functie niet differentieerbaar is, is ze niet differentieerbaar. Maar de *woorden* van eerstgenoemde volzin bekijkend, en een regelmatig gedrag in de opvolging der woorden van deze en van dergelijke volzinnen ontdekkend, projecteert de logicus ook hier een wiskundig systeem, en noemt zulk een volzin een *toepassing van het principe van tertium nondatur*." From Brouwer's thesis, the translation taken from L.E.J. Brouwer. Collected Works, volume 1: Philosophy and Foundations of Mathematics (ed. Arend Heyting). North-Holland, Amsterdam, 1975 [4]

that mathematics should be distinguished from its formal representation. This descriptive style is also more appropriate, considering Brouwer's interest in the philosophical aspects of constructivism. However, it is also due to this interest in the philosophy that certain mathematical aspects in his program were disadvantaged. Thus, though Brouwer was trying to oppose idealism, his "system itself had traces of idealism and, worse, of metaphysical speculation"[28].

On the other had, Brouwer's disciple, Heyting, who continued the development of intuitionism, was much less averse to formalization. Indeed, when the Dutch Mathematical Association announced, upon Gerrit Mannoury's suggestion, a call for formalizing Brouwer's ideas, it was Heyting who got the prize for the paper on the topic, which he submitted in 1928 [85]. From there on he continued systematizing Brouwer's ideas and introduced formal rules of intuitionistic logic in [65]. His axiomatization of arithmetic based on the principles of intuitionism became known as *Heyting arithmetic*.

Even more impactful for the development of intuitionism was Stephen Kleene's contribution to its formalization. He believed that using a formal system did not necessarily contradict Brouwer's ideas and, in fact, could enhance and speed up their understanding among mathematicians and logicians. Kleene drew parallels with elementary algebra, where "such formal manipulations add much to the rapidity with which mathematical deductions can be performed" [72]. His most noteworthy contribution was the introduction of the notion of *realizability*[4] in 1941. He further explained his ideas on *Intuitionistic number theory* in [71].

2.2 Recursive constructive mathematics

In the 1940s Andrey Markov, a Russian mathematician, developed a new branch of constructive mathematics, namely recursive constructive mathematics. Fundamentally this was a blend of recursive function theory with intuitionistic logic. Markov also added the principle of unbounded search to his program, known as *Markov's principle* (MP), which states that for all binary sequences consisting of 0s and 1s, if it is not true that all elements of a sequence are 0s, then 1 occurs in this sequence. Formally, this corresponds to the following statement:

$$(\forall n(A(n) \vee \neg A(n)) \wedge \neg\forall n \neg A(n)) \to \exists n\, A(n)$$

In the constructivist community this principle is not fully accepted. Markov's school, which admits MP is known as *Russian recursive mathematics*.

Maarten McKubre-Jordens argues that this uncertainty about MP is partly caused by the difference in the strength of an implication $A \to B$ as compared

[4]a form of interpreting constructive proofs that enables extraction of additional valuable information from them

to a disjunction $\neg A \vee B$ [84]. The first is weaker because if an algorithm takes a correct proof of A and turns it into a correct proof of B, it still does not make it clear whether $\neg A$ or B is the case. In [84] McKubre-Jordens also suggests that one possible justification for admitting MP is that it is possible to implement an algorithm, which can find 1 at some point in a sequence, provided that not all elements in the sequence are 0. However, there is no guarantee this can be achieved before a certain point in time, where it will still be relevant, for example, the end of humanity.

MP can also be formally stated as a rule:

$$\forall n(A(n) \vee \neg A(n)), \neg\neg(\exists n\, A(n)) \vdash \exists n\, A(n)$$

It was proven to be an admissible rule in the constructive counterpart of Peano arithmetic, *Heyting arithmetic*, by Troelstra [120] as well as in various sub-branches of intuitionism by Harvey Friedman [54]. Hugo Herbelin also presented intuitionistic predicate logic IQC_{MP}, which proves MP [64].

2.3 Bishop's constructive mathematics (BISH)

Constructive mathematics à la Bishop emerged in 1967, when Errett Bishop published his *Foundations of Constructive Analysis* [28]. There he presented a constructivist manifesto, looking at the foundations on which mathematics is built, exploring the idealism that existed in the field and, finally, bringing forward the constructivization of mathematics. In his monograph Bishop presented constructive proofs of classical theorems, including the Ascoli's theorem, the Stone-Weierstrass theorem, the Tietze extension theorem, Riemann mapping theorem, etc. As Douglas Bridges mentions in [32], Bishop's approach to constructive analysis "without a commitment to Brouwer's non-classical principles or to the machinery of recursive function theory" had a great impact amongst mathematicians and helped to decrease the existing scepticism towards constructivism.

Yet, in his emphatic publication *Schizophrenia in Contemporary Mathematics* Bishop admits that getting people to accept constructivism in mathematics is not an easy task because there is a constant attack on the common sense due to the attitudes prevalent at the time:

> "One could probably make a long list of schizophrenic attributes of contemporary mathematics, but I think the following short list covers most of the ground: rejection of common sense in favor of formalism; debasement of meaning by wilful refusal to accommodate certain aspects of reality; inappropriateness of means to ends; the esoteric quality of the communication; and fragmentation" [29].

Bishop's approach to constructive mathematics offers flexibility in interpreting the "finite routine"[5]. This allows describing other constructivist trends as well as classical mathematics through BISH. For example, constructive recursive mathematics corresponds to BISH that admits Markov's Principle and the Computable partial functions axiom. BISH with free choice sequence and bar induction represent a model of intuitionistic mathematics, while classical mathematics can be explained as BISH with LEM as an axiom.

Overall, Bishop and his followers focus on more pure mathematical activity as opposed to Brouwer's broader approach to constructivism.

2.4 Intuitionistic type theory

Around the same time that Bishop was writing his monograph, Per Martin-Löf was working on a number of constructive theories, which he presented in his lectures in 1966-68. His *Notes on Constructive Mathematics* were published in 1968 [82] and gave rise to *Intuitionistic type theory* also known as *Constructive type theory* or, simply, *Martin-Löf's type theory*.

In this approach every proposition has a type and new types can be built by combining the existing types using *type constructors*. These constructors are isomorphic with the *logical connectives*. A very common example of a type constructed this way is a function from natural numbers to natural numbers ($\mathbb{N} \to \mathbb{N}$), where the function constructor $\to$ corresponds to implication.

Martin-Löf makes a clear distinction between proofs and derivations. A *derivation* shows us that the statement is true; in other words, it is something that represents how the truth of a *proof object* is derived. The *proof*, on the other hand, is an object that carries the computational content and serves as a witness of the truth.

Each of these constructivist schools continue to develop, with type theory becoming increasingly popular in the logic community.

[5] by this Bishop meant an *algorithm*.

Chapter 3

Realizability

Contents

3.1 History and general aspects of realizability **15**

3.2 Realizability of induction and coinduction **17**

This chapter explains the notion of realizability that is used in program extraction and gives an overview of different approaches to realizability of induction and coinduction.

3.1 History and general aspects of realizability

To understand the theoretical concept of realizability one should look into the Brouwer-Heyting-Kolmogorov (BHK) interpretation[1]. In intuitionistic logic the meaning of a formula is not expressed by its truth or falsity, but by a proof. Consequently, in BHK interpretation the meaning of a formula A is represented by what constitutes a proof of A[2]. If this is a compound formula, then it is necessary to look at proofs of its constituent parts. In this regard the following applies:

- A proof of a conjunctive formula $A \wedge B$ is interpreted as a pair $\langle a, b \rangle$, where A is a proof of A and B is a proof of B;

[1]The name is due to Troelstra and van Dalen, however, there is a dispute in the community whether Brouwer's name should be included, considering that this is a step to formalization of the intuitionistic ideas.

[2]Although BHK is normally associated with intuitionism, Masahiko Sato developed a classical version of BHK interpretation in [103].

- A proof of a disjunctive formula $A \vee B$ is a pair $\langle a,b \rangle$, where a is 0 and b is a proof of A, or a is 1 and b is a proof of B;

- A proof of $A \to B$ is a function, which transforms a proof of A into a proof of B;

- There is no proof of falsity ($\bot$) and a proof of the formula $\neg A$ is defined as $A \to \bot$, that is a function f that converts a proof of A into a proof of $\bot$.

- A proof of $\forall x\ A(x)$ is a function f that converts an element $a \in D$ into a proof of $A(a)$;

- A proof of $\exists x\ A(x)$ is a pair $\langle a,b \rangle$ where a is an element of D, and b is a proof of $A(a)$.

BHK interpretation serves as a solid theoretical basis of realizability. Nevertheless, the notion of proof here is abstract. If we want to consider the use of realizability principles from a program extraction perspective then there is a need for a more "tangible" notion of proof. Realizability provides an approach to obtain such tangible proof object, following the same inductive structure as the BHK-interpretation. Here one may consider either Kleene's realizability interpretation — which uses natural numbers as realizers of formulas in Heyting arithmetic — or Kreisel's modified realizability — which uses typed lambda calculus.

The following clauses describe Kleene's interpretation and explain what it means for a number n to realize a formula A in Heyting arithmetic [72].

- If an atomic formula is true then any n is a realizer. If it is false, then no n is a realizer.

- A conjunctive formula $A \wedge B$ can be realized by a pair $\langle n,m \rangle$ iff n realizes A and m realizes B.

- A disjunctive formula $A \vee B$ can be realized by a pair $\langle n,m \rangle$ iff n is 0 or 1. Same as stated in the BHK interpretation, if n is 0 then m realizes A and if it is 1 then m realizes B.

- An implication formula $A \to B$ can be realized by a number n iff there is a function f encoded by n and whenever m realizes A, $f(m)$ is defined and realizes B.

- A number n realizes a formula $\forall x A(x)$ iff there is a function f encoded by n, which returns a realizer for $A(m)$ for any number m.

- A formula $\exists x A(x)$ can be realized by a pair $\langle n, m \rangle$ iff n is such that $A(n)$ holds and m is a realizer for $A(n)$.

Based on these clauses, one can conclude that if A is a statement that can be proved in Heyting arithmetic then there exists an n which realizes A. Since number realizers encode computable functions, we can apply the same idea to programming language terms.

3.2 Realizability of induction and coinduction

There is a multitude of approaches to realizability interpretation based on different types of logic systems used as a foundation of such interpretation. While realizability interpretation of first-order intuitionistic logic is relatively straightforward, inductive and coinductive definitions are treated diversely.

The use of inductive and coinductive definitions revolves around the notion of *fixed point operators*[3]. A comparative study of various existing definitions of fixed point operators is provided in [122].

The ways of applying realizability interpretation to proofs involving induction and coinduction were explored in various logical system, including not only first-order logic but also second-order logic. For instance, Daniel Leivant [79] and Jean-Louis Krivine [78] independently suggested a special system AF2, which is second-order intuitionistic logic that incorporates equations and the functionality for extracting lambda terms from proofs. Unfortunately, programs obtained in such a way may not necessarily be satisfactory in terms of time complexity. Therefore, Parigot extended this system, introducing recursive type theory TTR in order to avoid the efficiency issues [97].

AF2 was also extended by Favio Miranda-Perea who added monotone, not only positive, (co)-inductive definitions and used clauses to simplify these definitions. A clause in this instance is a tuple, which contains a predicate and function symbols, or tags, associated with this predicate. Tags of a clause can be constructors in case of inductive predicate expressions and destructors in case of coinductive predicate expressions [87]. The resulting logic system called MCICD (Monotone and Clausular Inductive and Coinductive Definitions) includes rules for folding the least fixed point (μI), iteration (μE), primitive recursion (μE^+), co-iteration (νI), primitive co-recursion (νI^+), folding the greatest fixed point/-coinductive inversion (νI^i), and unfolding of the greatest fixed point (νE). This means that inductive and coinductive definitions in MCICD are not symmetrical due to the absence of inductive inversion. However, inductive destructors, which are usually defined by unfolding of the least fixed point, can be obtained

[3]Fixed points are presented in chapter 5.

by means of primitive recursion. This helps to avoid issues with reduction when applying inductive inversion. Realizability interpretation is given in MCICD*, an extension of MCICD with additional terms and first-order existential and restricted formulas. Both MCICD and MCICD* have type preservation and strong normalisation.

An alternative approach is by Makoto Tatsuta [115], who offers two realizability interpretations of monotone coinductive definitions: expansion-preserving realizability and realizability based on second-order logic. Tatsuta looks into monotone coinductive definitions for a number of reasons but mainly because previous studies of realizability of coinductive definitions were restricted to positive formulas [115]. He also believes that positive condition is too restrictive and realizability of monotone coinductive definitions can enhance positive coinductive definitions.

Another system built for working with proofs for the purpose of program extraction is called Theory of computable functionals (TCF)[4]. TCF was developed by Schwichtenberg and used for the development of his theorem prover Minlog [108]. This system contains inductive and coinductive definitions and enables program extraction from classical and constructive proofs. TCF is based on Kreisel's modified realizability [74]. In his thesis, Schwichtenberg's student Kenji Miyamoto presents a combined inductive/coinductive definition in the context of TCF [88]. As Miyamoto points out there, the formal system they are using is similar to Berger's previous research, however, there is a difference in formalization. Our approach in this thesis is close to TCF and is based on previous results in [16, 23] with a number of modifications.

[4]See more on TCF in chapter 8.

Chapter 4

Lambda Calculus and its variations

Contents

4.1 Lambda calculus **19**
4.2 Simply typed lambda calculus **21**

This chapter presents Lambda calculus, the underlying theory of various formal logical systems. Lambda calculus has many extensions and variations. The basic distinction is between typed and untyped lambda calculus. Here we present the basic form of lambda calculus as well as its typed extension. The IFP version presented in this thesis is untyped but we consider types as implicit. It is, however, possible to introduce explicit types in IFP as well, as was done in [24].

4.1 Lambda calculus

Lambda calculus was introduced by Alonso Church in the 1930s to formalise the idea of effective computability [10].

In its most simple form lambda calculus has three sorts of terms.

- Variables (typically x, y, etc.).
- Abstraction ($\lambda x\ M$, where M is a lambda term and x is a variable). Intuitively, $\lambda x\ M$ is a function that takes an argument x, which may or may not occur inside of the expression M.
- Application (MN, where M and N are lambda terms). It corresponds to function application, where the function represented by M is applied to the argument N.

These terms can be manipulated using the following rules, which make lambda calculus the powerful system that it is.

α-equivalence allows bound variables renaming, i.e., two terms are equivalent even if bound variables are renamed consistently. In other words, names of λ-bound variables in terms are insignificant, as long as λ-abstractions express the same function after renaming. This means that variables can only be renamed in a way that avoids capturing free variables. Thus, $\lambda x\ x$ and $\lambda y\ y$ are α-equivalent; they are both identity functions. $\lambda x(\lambda y\ x)$ and $\lambda y(\lambda y\ y)$, on the other hand, are not equivalent as the first one is a constant function, while the second one will always return an identity function.

β-reduction is a way of expressing the notion of function application. It is based on variable substitution. Namely, we can say that a lambda term $((\lambda x\ M)N)$ can be β-reduced to the term $M[N/x]$. This means that all occurrences of variable x in M are substituted by N. It is important, however that N should not contain a free variable that becomes bound in case of such substitution. For example, consider the following: if $M = \lambda z\ (x\ y)$ and $N = z$ and z in M is not substituted, then the application $((\lambda x\ M)N)$ can be wrongly reduced to $\lambda z\ (z\ y)$, making z bound in this expression. However, if α-equivalence is first performed on M, making $M = \lambda w\ (x\ y)$, then we achieve the correct β-reduction to $\lambda w\ (z\ y)$, where z remains free:

$$((\lambda x\ (\lambda w\ (x\ y)))z) \Rightarrow_\beta (\lambda w\ (z\ y))$$

η-conversion is linked to the fact that two functions are equal if they return the same result for all arguments. For example, $f(x) = 10 + x$ and $g(x) = 5 + x + 5$. These two functions will always return the same value of $x + 10$ although they are defined differently. η-conversion can be summed up as follows: the lambda expression $\lambda x\ (f\ x)$ is equivalent to f if and only if x does not occur free in f.

The original Church's lambda calculus formulation was untyped. It is also Turing complete, meaning that it is possible to do recursive computation using only the constructs of pure lambda calculus. In this respect natural numbers are represented by Church numerals, e.g. $0 = \lambda f \lambda x\ (x)$, $1 = \lambda f \lambda x(f\ x)$, $2 = \lambda f \lambda x(f(f\ x))$, etc. β-reduction corresponds to computation. For instance, if we want to add 1 to some number m (a Church numeral), then we need to apply $\lambda n \lambda f \lambda x(f(nfx))$ to m and β-reduce this expression to obtain the result. Recursion is achieved by the fixed point combinator[1] Y.

[1]A combinator is a lambda term with no free variables.

4.2 Simply typed lambda calculus

Looking at β-reduction, one might think that if it is applied enough times, then eventually a normal (not further reducible) form can be obtained. Lambda calculus in its pure form, however, does not satisfy the normalization property. Consider an expression $\Omega = \omega\omega$, where $\omega = \lambda x(xx)$. Applying β-reduction we substitute both xs in the first ω by the second ω. This, however, means that we get the original expression $(\lambda x(xx))(\lambda x(xx))$ back, i.e. β-reduction gets stuck in a loop $\Omega \Rightarrow_\beta \Omega$.

On the other hand, introducing types to lambda calculus solves the problem with normalization. In the programming context, since lambda calculus with types satisfies the normalization property, it can be viewed as a "programming language" in which every program terminates. The normalization property that we have in this case, however, comes with a price of making the system non-Turing complete, that is limiting the number of computable functions that can be defined in it. Schwichtenberg showed in [106] that functions definable in the simply typed lambda calculus are exactly the extended polynomials and in [58] Jean-Yves Girard showed that the functions definable in second-order calculus are exactly the computable functions, whose totality can be proven in second-order arithmetic. This is a huge class of computable functions, which is more than sufficient for practical purposes.

Henk Barendregt defines types as objects of a syntactic nature that may be assigned to lambda terms [11]. There are two main presentations of simply typed lambda calculus. One was proposed by Curry in 1934 [42] and another one by Church in 1940 [37]. The main difference between these approaches is that types in the first one are implicit, while in the second one explicit annotations are given to abstracted variables. This means that in case of lambda calculus *à la* Curry, a program can be written without types and then type inference is performed to check if it is possible to assign a type. In Church's version types should be specified when the program is written.

In its most basic form simply typed lambda calculus is based on the set of types:

$$\mathbf{Ty} \ni \rho, \sigma ::= b \mid \rho \to \sigma$$

Here b stands for base types; $\rho \to \sigma$ is a type of functions mapping elements of type ρ to elements of type σ. This means a function of type $\rho \to \sigma$ can be applied to a term of type ρ and such application results in a term of type σ.

If we consider the terms of the pure lambda calculus, *abstraction* in simply typed lambda calculus has the following forms:

- Church-style: $\lambda x : \rho.M$, which means that the bound variable x is of type ρ, or

$\Gamma, x:\rho \vdash x:\rho$	$\Gamma \vdash c:\rho \quad \text{if } c:\rho \in \mathcal{C}^*$ $^*\mathcal{C}$ is a set of typed constants	
$\dfrac{\Gamma, x:\rho \vdash M:\sigma}{\Gamma \vdash \lambda x:\rho\,.M \;:\; \rho \to \sigma}$	$\dfrac{\Gamma \vdash M:\rho \to \sigma \qquad \Gamma \vdash N:\rho}{\Gamma \vdash M\,N:\sigma}$	
$\dfrac{\Gamma \vdash M:\rho \qquad \Gamma \vdash N:\sigma}{\Gamma \vdash \langle M,N\rangle : \rho \times \sigma}$	$\dfrac{\Gamma \vdash M:\rho \times \sigma}{\Gamma \vdash \pi_0(M):\rho}$	$\dfrac{\Gamma \vdash M:\rho \times \sigma}{\Gamma \vdash \pi_1(M):\sigma}$

Figure 4.1: Typing rules for simply typed lambda calculus [19]

- Curry-style: $\lambda x\, M$, where the type of x is not specified.

Apart from variables, abstraction and application, in simply typed lambda calculus additional terms are introduced. Namely:

- constants (c),
- pairs ($\langle M,N\rangle$, where M and N are lambda terms)
- projections ($\pi_0(M)$ and $\pi_1(M)$, where M is a lambda term) — π_0 will return the first element of the pair and π_1 will return the second element.

For our purposes we also consider an additional type $\rho \times \sigma$ for pairs.

Terms in simply typed lambda calculus can be checked for well-typedness using a set of standard typing rules. Figure 4.1 contains the rules for the simply typed lambda calculus with explicit typing.

Chapter 5

Domain theory

Domain theory emerged in the late 60s-early 70s and is associated with the name of Dana Scott. The driving force behind this theory was his desire to establish a formal semantics for lambda calculus. He approached the task by constructing a model that linked together the syntactical and notational sides of the lambda calculus. This approach uses partial computations that contain intermediary information before the final result is calculated. It is crucial to keep track of the least element, that is the undefined output that will be used as the input in the next step of the computation. Domain theory also relies on ordering of the elements and has connections to *Order theory* and, specifically *Lattice theory* as its branch.

In [110] Scott started exploring the idea of approximation between programs. This is particularly useful for non-terminating programs; it allows expressing the features of a programming language through mathematical constructs in a clear and elegant way.

In Lattice theory a *partially ordered set*, or *poset* is a set (e.g., D) with the *partial order* relation $\sqsubseteq$ between the elements of D [27]. We write it as a pair $(D, \sqsubseteq)$. This is a binary relation expressing that one object contains more information than the one below in the ordering. In other words, the object that is below *approximates* the higher one in the ordering. The partial order is characterized by

- *reflexivity* : $\forall d \in D \; d \sqsubseteq d$,
- *transitivity* : $\forall d, d', d'' \in D \; d \sqsubseteq d' \sqsubseteq d'' \rightarrow d \sqsubseteq d''$,
- *antisymmetry* $\forall d, d' \in D \; d \sqsubseteq d' \sqsubseteq d \rightarrow d = d'$.

A sequence $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq d_3 \; \sqsubseteq \ldots$ of elements of D is called a chain. An element $d \in D$ is called an *upper bound* of the chain if $\forall n \in \mathbb{N} \; d_n \sqsubseteq d$. The *least*

upper bound of the chain, abbreviated as *lub*, is expressed by $\bigsqcup_{n\geq 0} d_n$. The lub is defined in terms of the following:

- $\forall m \geq 0 \;\; d_m \sqsubseteq \bigsqcup_{n\geq 0} d_n$
- $\forall d \in D$ if $(\forall m \geq 0 \; d_m \sqsubseteq d)$ then $\bigsqcup_{n\geq 0} d_n \sqsubseteq d$

The *greatest lower bound*, abbreviated as *glb* is written as $\bigsqcap_{n\geq 0} d_n$. A poset is a *lattice* iff for every two elements $d, d_1 \in D$, there exist the least upper $(d \sqcup d_1)$ and greatest lower $(d \sqcap d_1)$ bounds. If the poset D has the least element, it is usually written as $\bot$. In this case $\bot \sqsubseteq d$ for all elements d in D.

Another notion that plays an important role in domain theory and denotational semantics is a *fixed point*. In simple terms, given a function f and an element a of the function's domain, a is a fixed point of f if $f(a) = a$. Related to this are the notions of the *least fixed point* (*lfp*) and the *greatest fixed point* (*gfp*). In case of a function from a poset to itself, there can only be one lfp represented by the fixed point that is less than any other fixed point in this set with respect to the ordering. *lfp*(f) of a continuous function f is defined as $\sqcup_n f^n(\bot)$, where $f^n(\bot) = f(f...(f(\bot))$ and n runs through all natural numbers. Similarly, gfp is represented by the fixed point which is the greatest fixed point in the ordering. We use lfps for recursion, while gfps are associated with corecursion.

A *chain complete poset*, is a poset with *complete partial order* (*cpo*), where all chains have least upper bounds. A *domain* is defined as a cpo that has a least element. There are many different classes of domains but in this thesis we only work with Scott domains discussed at the end of this section.

The partial order allows us to find the minimal solution to fixed point equations. This solution is the limit of increasing chain of approximations to the solution.

Domain theory distinguishes *primitive domains*, which are just sets of atomic elements, and *compound domains*, which are constructed from the primitive domains [105]. Compound domains include:

- *product* that takes two domains and builds a new one by combining their elements into pairs: $D_1 \times D_2 = \{(d_1, d_2) \mid d_1 \in D_1$ and $d_2 \in D_2\}$, which is essentially the same principle as *Cartesian product* in set theory. Here the partial order relation between the pairs $(d_1, d_2 \sqsubseteq d_1', d_2')$ is defined as $d_1 \sqsubseteq_{D_1} d_1'$ and $d_2 \sqsubseteq_{D_2} d_2'$. The subscripts D_1 and D_2 refer to the corresponding domain ordering.
- *disjoint union* or *sum* of two domains D_1 and D_2 is defined as $D_1 + D_2 = \{(dom_1, d_1) \mid d_1 \in D_1\} \cup \{(dom_2, d_2) \mid d_2 \in D_2\}$, where dom_1 is a label refer-

ring to the domain D_1 and dom_2 is a label referring to the domain D_2. The ordering $\sqsubseteq_{D_1+D_2}$ is defined as follows:

$$(dom_1, d_1) \sqsubseteq_{D_1+D_2} (dom_1, d_2) \text{ iff } (d_1 \sqsubseteq_{D_1} d_2)$$
$$(dom_2, d_1) \sqsubseteq_{D_1+D_2} (dom_2, d_2) \text{ iff } (d_1 \sqsubseteq_{D_2} b2).$$

All other elements are not comparable.

A *separated sum* is a disjoint sum of X and Y with a new bottom element added.

- *function space*, denoted by $\rightarrow$, takes two domains D_1 and D_2 and creates a new domain $D_1 \rightarrow D_2$, which contains functions from domain D_1 to codomain D_2, that is $\{f \mid f : D_1 \rightarrow D_2\}$. Here the domain D_1 consists of a set of inputs and the codomain D_2 is a set of all possible outputs. The partial order between the elements of this new domain is defined as follows: $f \sqsubseteq f' \overset{\text{Def}}{=} \forall d \in D_1 \; (f(d) \sqsubseteq_{D_2} f'(d))$. The function space is usually the space of continuous functions ordered pointwise. A function $f : D_1 \rightarrow D_2$ is continuous if it is monotone and respects directed sups. The latter means $f(\sup_n d_n) = \sup_n f(d_n)$ for any increasing chain d_n.
- *lifted domains*. The domain builder $()_\perp$ is used to create a lifted domain. Here $\perp$ refers to non-termination or non-definedness. For instance, from the domain D we can create a new domain $D_\perp$, which contains all the elements of D (called *proper elements*) plus the extra *improper element* $\perp$.

Semantic domains are normally defined via equations. However, sometimes programming languages need to be interpreted through *reflexive domains*, which are defined recursively and have a form $D = F(D)$. Associated with this is the notion of *Cartesian closed categories of domains*, which describes domains closed under function spaces. If in a poset every directed subset has a sup, then it is called *directed-complete partial order (dcpo)*. Dcpo is Cartesian closed. Finite sets belong here. For infinite sets it is not the case. For instance, a set $\mathbb{N}$ defined in the usual way is not a dcpo. In order to make it into a dcpo, a top element needs to be added. A poset in which every element is the sup of the compact elements below it is called an algebraic poset. Recursive domain equations can be viewed as fixed point equations that are solved in categories instead of cpos [112].

With respect to finite sets, we need to consider *compact (finite) elements*. If $(S, \sqsubseteq)$ is an ordered set and $c \in S$, c is compact iff $c \ll c$, where $\ll$ expresses the *way below relation*. For $x, y \in S$ we say $x \ll y$ iff for every directed subset D of S: if D admits a sup and $y \sqsubseteq \sup D$, then there is some $d \in D$, such that $x \sqsubseteq d$.

Another prominent name in domain theory is Yuri Ershov. While Scott focused on semantics, Ershov worked on his *Theory of numbering* [48], aiming to

give a foundation for computability theory for structures that were not necessarily just natural numbers.

Although their motivation differs, one can draw a lot of parallels in their work. This was not apparent until after the fall of the Iron Curtain as communication between Western scientists and their Soviet counterparts was not very active prior to that point.

The central point of both theories are *domains*. With the multitude of domains and their interpretations out there it is important to pinpoint *algebraic domains* as the subject of interest in this research. An algebraic domain, also known as *Scott domain* or *Scott-Ershov domain* is a cpo in which all the finite subsets that have some upper bound also have a lub.

Our research uses Scott's motivation, for the most part, but it also includes the computability aspect. We do not only give the denotational semantics of programs but also present adequacy, showing the link between denotational and operational semantics.

Domain and its applications are of interest to many researchers, including Achim Jung and Samson Abramsky, who gave a comprehensive overview of Domain Theory in [1], Viggo Stoltenberg-Hansen, who presented mathematical theory of domains in [113, 31], Abbas Edalat, whose collaborative research includes many applications of Domain theory to computable analysis [45, 44, 46], Klaus Keimel [70, 57, 1, 47], Martín Escardó [49, 47, 50, 68], Jens Blank [30, 18, 31], Andrej Bauer [13, 14], Lars Birkedal and Rasmus Møgelberg [26, 89], and others.

Chapter 6

Semantics

Contents

6.1 Operational semantics **28**
6.2 Axiomatic semantics **28**
6.3 Denotational semantics **29**

Showing the correctness of a program is not a trivial task. *Syntax* and adherence to a *grammar* are important in the creation of a working program. However, it is also essential to ensure that the meaning of this program is as expected. *Semantics* is exactly the area that works with meanings. Traditionally, in computer science we distinguish three ways of looking at the meaning of a program.

- *Operational semantics*, which conveys the meaning of computational steps, that is starting from a certain *state*[1] and looking at how the end result of the computation is achieved.
- *Axiomatic semantics*, which conveys the meaning through axioms and rules of a logic that describe the intended connotation of program commands; this semantics is used to show partial correctness of a program in terms of pre- and post-conditions.
- *Denotational semantics*, which conveys the meaning of a program in terms of a mathematical function that assigns to each program an object in some mathematical space.

[1]One of the core notions in semantics is the state of the memory. Assuming that a program is computed through variables, the state can be interpreted as a function that maps these variables into values.

6.1 Operational semantics

Operational semantics is associated with Gordon Plotkin, who introduced *structural operational semantics (SOS)*, also known as *small-step semantics* in 1981 [99], and Gilles Kahn, who, noticing the growing popularity of Plotkin's ideas, introduced the *big-step operational semantics* in 1987 [69]. Kahn referred to it as *natural semantics*. The use of adjective "operational" is attributed to Scott [111].

There is also an alternative way of representing operational semantics called *reduction semantics* first presented by Matthias Felleisen and Robert Hieb in 1992 [52], however, for the purposes of our research, we only focus on the first two, although we sometimes refer to both, big and small, semantics in terms of *reduction steps*.

SOS can be expressed in terms of *transition systems*. These systems are used to describe how a procedure (calculation) is carried out step by step using *transitions*. These transitions must follow specially defined inference rules, which ensure the validity of the transitions. Small-step semantics represents the calculation process with high level detail, making it easy to follow and intuitive. This led to popularity of SOS in the computer science community. In some situations, however, such detailed description is unnecessary, and the rules can be represented in a simpler, more *natural* way. This is where Kahn's approach is more suitable, allowing the use of fewer rules and resulting in simpler proofs. Big step semantics is a simple way of reasoning about evaluation, but SOS allows observing intermediate state transitions and provides easier reasoning about infinite computations (we can advance a computation as many steps as we need).

6.2 Axiomatic semantics

Axiomatic semantics appeared in the late 60s and is associated with the names of Robert Floyd [53] and Tony Hoare [66]. In his 1969 paper Hoare acknowledges Floyd's contribution:

> "The formal treatment of program execution presented in this paper is clearly derived from Floyd. The main contributions of the author appear to be: (1) a suggestion that axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined; (2) a comprehensive evaluation of the possible benefits to be gained by adopting this approach both for program proving and for formal language definition" [66].

Axiomatic semantics is built around assertions about programs and rules that help one to check the truth of these assertions.

For example, we have an assertion P (pre-condition) and an assertion R (post-condition). We construct a so-called *Hoare triple* by combining these assertions with a program Q in the following way: $P\{Q\}R$. If the pre-condition holds and the program executes and terminates, then the post-condition holds. To show how this works in the case of arithmetic, several axioms about the general properties of the basic arithmetical operation need to be presented in the form of axioms. For instance, amongst the axioms that Hoare introduces in his paper are commutativity of addition and multiplication, associativity of addition and multiplication, addition of 0, and others. Based on that, we can check the validity of a triple, for instance $x = 1\{x := x+0\}x = 1$ is valid.

Since axiomatic semantics is developed to check the correctness of programs, Hoare stresses the need to consider assignments. He formalizes the *Axiom of Assignment* as $\vdash P_0\{x := f\}P$, where x is variable and f is an expression that may contain x. Assuming that we are working in a programming language that has no side effects, by performing the substitution $[x/f]$ we get P_0 from P. Additionally, he introduces several inference rules to combine axioms and theorems for more robustness. These include rules of consequence, composition, and iteration.

6.3 Denotational semantics

Denotational semantics is associated with Christopher Strachey [114] as well as Scott's research in the area of domains [61, 109].

Strachey was concerned about the knowledge gap and lack of agreement that his contemporary programming language designers had when it came to understanding of the meaning of programs.

> "The trouble seems to be that programming language designers often have a rather parochial outlook and appear not to be aware of the range of semantic possibilities for programming languages. As a consequence they never explain explicitly some of the most important features of a programming language and the decisions among these ... have a very important effect on the general flavour of the language. ... although the subject is clearly a branch of mathematics, we still have virtually no theorems of general application and remarkably few specialised results" [114].

The idea behind denotational semantics is giving meanings to programs through associations with certain mathematical objects.

One can draw a parallel with philosophy, where a number is just a representation of the idea of a number. Similarly, in a programming context the integer 2, which is internally represented by a combination of bits, corresponds to $2 \in \mathbb{Z}$ in

mathematics. Giving a *denotation* to each object in a programming language enables us to ensure that the intended concepts are indeed represented correctly. For instance, the expression $[\![2+3]\!]$ evaluates to $[\![5]\!]$. The meaning of this expression in mathematics is $5 \in \mathbb{Z}$. To understand why it is indeed true, we look at the meaning of its components $[\![2]\!]$ and $[\![3]\!]$ and the meaning of the operation $[\![+]\!]$, which in this case corresponds to the addition in mathematics. Regardless of whether one evaluates this expression first or simply looks at the components, the correspondence between the programming concepts and mathematical objects remains correct.

While this looks fairly straightforward, defining a function in a program as a mathematical function was tricky, specifically because there needs to be a mathematical object that represents recursive functions. Here is where domain theory and the concept of partial functions came into play to express the computations that can loop and potentially fail. Recursion in this case is represented as an approximation of a result.

Chapter 7

Harrop formulas

Considering *Curry-Howard isomorphism*[1] one may think that "translation" of a mathematical proof into a computer program is a straightforward process, which does not require any additional processing. However, such a "translation" may result in programs of unnecessarily high complexity. Therefore, some filtering of proof content is required. While normalization brings one level of such filtering, it still does not guarantee that all of the "translated" content is computationally relevant. This motivates us to introduce the distinction between Harrop and non-Harrop formulas with the purpose of filtering out "pseudorealizers" and including only relevant content.

In 1932 Kurt Gödel made an observation, without stating the proof, that

- If a disjunction $A \vee B$ is a theorem provable in the intuitionistic propositional calculus then either A is a provable theorem or B is a provable theorem in this calculus.

The proof of this observation, referred to as the *disjunction property*, was later given by Gerhard Gentzen, John McKinsey and Alfred Tarski, Kleene and others [90]. Kurt Schütte obtained the same result for predicate calculus using cut elimination. Helena Rasiowa and Roman Sikorski also proved it for predicate logic. Schütte's model of the calculus resembles that of Gentzen, although he does not use sequences. Rasiowa and Sikorski prove this by looking at the algebraic structure of predicate calculus. Their approach is inspired by the algebraic proofs for propositional calculus by McKinsey and Tarski as well as Ladislav Rieger [102].

Another property of intuitionistic logic is the *existence property* proven by Kleene

[1] Curry-Howard isomorphism is direct relationship between proofs and programs.

- An existential statement $\exists x A(x)$ is a provable theorem if and only if for some closed term t, $A(t)$ is a provable theorem.

With these properties in mind, to separate constructively valid content from non-constructive content, a subset of formulas H is defined below. This subset does not contain disjunction or existential formulas in strictly positive position.

- Atomic formulas, including $\perp$, are H.
- $A \wedge B$ is H if both A and B are;
- $A \rightarrow B$ is H if B is H and A is an arbitrary well-formed formula;
- $\forall x\, A$ is H if A is.

Such formulas are known in various sources as *Harrop formulas* after Ronald Harrop, who introduced them in 1956. However, Schwichtenberg and Troelstra refer to them in [121] as *Rasiowa-Harrop formulas* as Rasiowa had introduced them independently in her 1954 and 1955 papers. In fact, Harrop himself mentions her research in his 1956 paper [62].

A more complex definition of Harrop formulas is explored in [86], however, for our purposes, using the above definition as a base is more appropriate. We will, however, modify it slightly by allowing existential quantifiers without restriction, and extend it to cater for the language that we are using. The changes will be described in section 9.4.

Chapter 8

Program extraction

Computer technologies developed immensely in the last 30 years and became ubiquitous. Having an undoubtedly positive effect on our lives, such rapid progress was not without faults. The speed of program development was often prioritised. Although software checks were performed, especially in safety-critical applications, they turned out not to be sufficient in a number of cases and led to fatalities and financial losses. As we welcome technology into our lives yet more, making sure that the software, seamlessly integrated in everyday objects, is indeed correct stands out as a priority.

Classically, program correctness was done by writing a program and adding a separate proof of its correctness. Alternative to that is proving a theorem that is a formalized representation of program specifications and extracting a program from it together with the proof of its correctness. This approach, although different, takes its inspiration from *program synthesis*, a method of deriving a program from the provided specification [55, 81].

The first program extractor, *PX (Program eXtractor)*, appeared around 1985 and was developed by Susumu Hayashi at Kyoto University [63]. Essentially it is a constructive logic for computation and a tool for program extraction written in Lisp. It can also be seen as a foundation of type theories. Although it may at first seem that the system development was inspired by type theory, which was on the rise at the time, PX is in fact based on an untyped theory. The actual inspiration came from the research performed by the Japanese group. Specifically, in 1979 Shigeki Goto [60] and Masahiko Sato [104] looked at using Gödel's interpretation for extracting programs from proofs in Heyting arithmetic. Hayashi found the results achieved in this way unsatisfactory due to slowness of the experimental reducer developed for Sato's approach and the lack of correctness proof at compilation for Goto's system. PX uses realizability interpretation, called *refined px-realizability*, instead of Gödel's interpretation and is built on the basis of the

constructive system T_0 [51].

Nuprl [80], *Coq* [117], *Isabelle/HOL* [67], and *Minlog* [118] are among the modern proof assistants that have program extraction functionalities. Each of the mentioned tools implements this functionality in its specific way. Neither of them was built with program extraction as its primary goal.

In 1984 Robert Constable and his collaborators released *Nuprl*, a proof development system based on Martin-Löf type theory. *Nuprl* takes its roots in Bate's *refinement logic* [12], *Cornell Program Synthesizer* [116], *Lambda-prl* [39] and aims to present constructive type theory as practiced in Cornell.

A detailed tutorial draft describing the capabilities of the system appeared 1985. This tutorial was officially published in 1986 [38]. Already back then the system had a special evaluation mechanism that allowed using *Nuprl* as a functional programming language. This method enabled extraction of constructive meanings of theorems in the form of (executable) terms. The users could save these terms into a library and run them on an interactive evaluator. The terms (programs) are extracted in ML. *Nuprl* includes libraries of theorems and other mathematical data and libraries of ML programs. These are helpful for generating proofs for new theorems.

Over the years *Nuprl* developed into *Nuprl LPE*, an extensive logical programming environment with multiple inference engines, editors, evaluators, and translators. All these independent modules communicate with the central library module that stores information about completed proofs and their environments [3]. Overall, *Nuprl LPE* is a powerful framework aimed at facilitating the proof process.

Coq is another system that allows program extraction. This tool is based on dependent type theory and the Curry-Howard isomorphism. Its roots go back to 1984, when Thierry Coquand and Gerard Huet presented their *Theory of Constructions*, which was a preliminary version of *The Calculus of Constructions* (abbreviated as CoC) published in 1988 [40]. An important step in the development of the system was introduction of a formal realizability interpretation by Christine Paulin-Mohring in 1989 [98] and subsequent implementation of the program extraction as an extra feature of *Coq* by Benjamin Werner. Later program extraction was re-implemented by Jean-Christophe Filliâtre and Pierre Letouzey. Currently, *Coq* allows extraction of programs in OCaml, Haskell, and Scheme.

Although program extraction is fairly well developed in *Coq*, it is not the only way of ensuring correctness of programs. An alternative approach aimed at generating the correctness proof from a program itself is presented in [95].

Isabelle is a framework that can work with different logics and *Isabelle/HOL* is a widely-used system which implements a variant of Church's simple theory of types. Therefore, unlike *Nuprl* and *Coq*, which are based dependent type theories, *Isabelle* allows representing proofs as lambda terms. With that in mind, Stefan Berghofer came up with a framework enabling extraction of programs from constructive proofs in *Isabelle/HO*L and successfully applied it in several non-trivial cases [25].

Minlog was developed by Schwichtenberg *et al.* who aimed to create a system that would use "the proofs-as-programs paradigm to let program development go hand-in-hand with program verification" [15]. It is related to implementations of constructive type theory. Programs extracted from *Minlog* are simply typed lambda terms with higher type primitive recursion. In other systems programs might not be as clearly separated from proofs and may have complicated dependent types.

Theory of Computable Functionals (TCF), the underlying system of *Minlog*, is based on minimal logic, where formulas are defined through implication and universal quantification operations. Structurally speaking, TCF works with *formula forms*, *predicate forms*, and *clause forms*. Rules for well-formed forms are given in [108]. Existence, conjunction and disjunction are predicates defined inductively similarly to the definitions by Martin-Löf from [83]. TCF works with both inductive and coinductive definitions. For example, in the case of natural numbers one can define totality and cototality and the corresponding least and greatest fixed point axioms accordingly.

The *Minlog* system served as an inspiration for our proof assistant. Indeed, there are a lot of similarities in the underlying system of TCF and the system that we are using. Both systems are based on first-order logic, and our proof assistant can be viewed as a variant of *Minlog* with a focus on abstract mathematics. The underlying difference is in the way how induction and coinduction are defined. We treat them as fixed points of strictly positive operators, while in *Minlog* they are expressed via clauses, similar to the way it is done in *Coq* and *Agda*. Our approach allows us to treat induction and coinduction as dual to each other.

From a theoretical perspective the two approaches are equivalent but in practical terms there is a big difference. This is because in our case we aim for more abstraction and extract programs using different notions of realizability.

When programs are extracted from proofs, there is always a concern about their efficiency in terms of complexity. Therefore, there is still the need to find appropriate program simplification strategies. Alexei Nogin attempted this for *Nuprl* with partial success and outlined several principles of programming in [91].

Minlog distinguishes between non-computational and computational formulas, and further between computationally irrelevant (Harrop) and computationally relevant (non-Harrop) formulas. Such a distinction means that the extracted programs are cleared of any content that is computationally irrelevant. This optimises the time complexity of these programs.

TCF makes use of non-computational quantifiers $\forall^{nc}$ and $\exists^{nc}$, i.e. quantifiers which allow one to avoid dependency of an extracted program on the quantified variable. Although we are not explicitly marking quantifiers as non-computational in our system, the realizability definitions for quantified formulas (see fig. 9.10) depict the same principle.

An interesting study comparing program extraction from normalization proofs in *Coq*, *Isabelle* and *Minlog*, which draws attention to the differences between the systems and aspects that each system can borrow from the others, is given in [17].

Another proof assistant that allows program extraction is *Agda* [2]. An example of program extraction for exact real number computation is presented in [36].

To conclude, there are various systems that have program extraction integrated to various extent. Neither of these tools is currently able to work with our intended formal system.

Part II

Theoretical framework for program extraction

Chapter 9

Formal system for program extraction

Contents

9.1 IFP 40

9.2 IFP' 43

9.3 Program semantics and RIFP 45

9.4 Harrop expressions 49

9.5 Realizability for program extraction 51

This chapter looks at the theoretical framework developed specifically for the purpose of program extraction. We begin with the introduction of the formal system of intuitionistic fixed point logic (IFP) as the core on which the program extraction infrastructure is built. This includes the derivative system RIFP, which is a version of IFP with realizers. Here we present our version of realizability interpretation as a core part of the program extraction process.

Intuitionistic fixed point logic with its extensions emerged as a result of search for a new approach to program extraction in which induction and coinduction are dual. Induction is used by proof theorists, logicians and mathematicians on a regular basis. Coinduction, although used as well, usually takes longer to comprehend, especially for those who are new to it. Therefore, the duality of these two concepts could be helpful and enhance the understanding of coinduction.

The development of IFP was initiated by Berger. It is hard to pinpoint when and where exactly IFP was conceived, however a formal system that closely resembles IFP is outlined in [16].

The existing formalization underwent multiple reevaluations on the basis of program extraction efficiency. It was also adjusted and improved after consideration of the restrictions that we encountered initially when proving that the system was sound.

The development of the proof assistant alongside the theory also urged a number of modifications, for instance in terms of operational semantics. The following subsections present IFP as a *schema*, rather than a fixed system for a specific mathematical field. One can view it as a family of systems for formalization of various mathematical fields.

9.1 IFP

The base of IFP is intuitionistic first-order logic. It is extended through the addition of the least and the greatest fixed points of *strictly positive (s.p.)* operators. An operator is strictly positive if it is of a form $\lambda X\, P$, where every free occurrence of the variable X in the predicate P is at a strictly positive position, i.e. not in the left hand side of an implication. The definition of free variables is given in fig. 9.2. This definition distinguishes between object and predicate variables in IFP and also includes program variables, which will be introduced in section 9.3.

IFP is parametric in a set of *sorts*, a set of *terms*, where each term has a particular sort, and a set of *predicate constants*, where each predicate constant has a certain *arity*, i.e. a list of sorts. A sort denotes a set of mathematical objects of the same 'kind'. A term is either a variable, a constant, or a function symbol applied to terms. A function symbol is the name of a function which, in the first place, is an abstract object. With a function there may be associated a computation process but this is not necessarily so. For simplicity, we assume we have only one sort. We also define three types of expressions simultaneously: *predicates* (P, Q), *formulas* (A, B), and *operators* (Φ, Ψ). Expressions existing in the language of IFP are concisely represented in fig. 9.1. Since we opted to use only one sort here, arity is represented by the cardinality for the same purpose of making notation more concise.

Amongst predicates one distinguishes *predicate variables* (e.g., X, Y), or *predicate constants* (e.g., P_0, Q_0). Each of these have a fixed arity. A predicate constant is a parameter and users can choose what predicate constants they want. For instance, we define a predicate constant $\perp$ of zero arity, which represents falsity. We have used $\perp$ earlier to denote a domain element. The predicate constant $\perp$ has no relation to it. In this thesis we use this symbol to denote both notions. The intended meaning can be derived from the context in which it is used.

Predicates can also be *abstractions* (e.g., $\lambda\vec{x}\, A$). In this case $\vec{x}$ is a shorthand for a list of variable-sort tuples, the length of which represents the arity of such

$$
\begin{array}{lll}
\textit{Formulas} \ni A, B & ::= & P(\vec{t}) \ (P \text{ not an abstraction}^1, \vec{t} \text{ arity}(P) \text{ many terms}) \\
& & \mid \ A \wedge B \ \mid \ A \vee B \ \mid \ A \to B \ \mid \ \forall x\, A \ \mid \ \exists x\, A \\
\textit{Predicates} \ni P, Q & ::= & X \ \mid \ P_0 \ (P_0 \text{ atomic}) \ \mid \ \lambda \vec{x}\, A \ \mid \ \mu\,\Phi \ \mid \ \nu\,\Phi \\
& & (\text{arity}(\lambda \vec{x}\, A) = |\vec{x}|, \ \text{arity}(\mu\Phi) = \text{arity}(\nu\Phi) = \text{arity}(\Phi)) \\
\textit{Operators} \ni \Phi, \Psi & ::= & \lambda X\ P \quad (\text{arity}(\lambda X\ P) = \text{arity}(X) = \text{arity}(P))
\end{array}
$$

Figure 9.1: Expressions in IFP

$$
\begin{array}{rll}
FV(x) &= \{x\} & (x \text{ any* variable}) \\
FV(c) &= \emptyset & (c \text{ any* constant}) \\
FV(f(t)) &= FV(t) & \\
FV(P(t)) &= FV(P) \cup FV(t) & (P \text{ not an abstraction}) \\
FV(A \,\Box\, B) &= FV(A) \cup FV(B) & (\Box \text{ in } \{\wedge, \vee, \to\}) \\
FV(\Box(x)\ A) &= FV(A) \setminus \{x\} & (\Box \text{ in } \{\forall, \exists\}) \\
FV(\lambda x\, A) &= FV(A) \setminus \{x\} & \\
FV(\diamond(\Phi)) &= FV(\Phi) & (\diamond \text{ in } \{\mu, \nu\}) \\
FV(\lambda X\ P) &= FV(P) \setminus \{X\} & \\
& & \\
FObV(E) &= \{x \in FV(E) \mid x \text{ is an object variable}\} & \\
FPredV(E) &= \{x \in FV(E) \mid x \text{ is a predicate variable}\} & \\
FProgV(E) &= \{x \in FV(E) \mid x \text{ is a program variable}\} & \\
& & {}^*\text{obj., pred., or prog.}
\end{array}
$$

Figure 9.2: Free variables in IFP

a predicate. Lastly, predicates can be *fixed points* $\mu\Phi$ and $\nu\Phi$. The arity of these predicates corresponds to the arity of the operator Φ.

An atomic predicate formula is of a form $P(\vec{t})$, where P is a predicate that is not an abstraction[1] and $\vec{t}$ is a list of terms associated with this predicate. The sorts that correspond to these terms represent the arity of P. An equation $s = t$ is an example of such atomic formula with an infix notation. It can be written as $=(s,t)$, where $=$ is a predicate and s and t are terms of the same sort.

[1]If P would be an abstraction, we would need to normalize the formula. We exclude abstraction to avoid redundancy in this case.

$P(\vec{t})$ is s.p. in X iff P is s.p. in X.

$A \diamond B$ where $\diamond \in \{\wedge, \vee\}$ is s.p. in X iff A and B are s.p. in X.

$A \to B$ is s.p. in X iff X is not free in A and B is s.p. in X.

$\diamond x\, A$ where $\diamond \in \{\forall, \exists\}$ is s.p. in X iff A is s.p. in X.

Y is s.p. in X (irrespective of whether $Y = X$ or $Y \neq X$).

P_0, where P_0 is a predicate constant, is s.p. in X.

$\lambda \vec{x}\, A$ is s.p. in X iff A is s.p. in X.

$\diamond \Phi$, where $\diamond \in \{\mu, \nu\}$, is s.p. in X iff Φ is s.p. in X.

$\lambda Y\, P$ is s.p. in X iff $Y = X$ or P is s.p. in X.

Figure 9.3: Definition of strict positivity

Composite formulas include conjunction ($A \wedge B$), disjunction ($A \vee B$), implication ($A \to B$), and quantified formulas ($\forall x\, A, \exists x\, A$). As mentioned earlier, the predicate constant $\bot$ has zero arity, so the corresponding formula for it is $\bot()$.

Operators are defined as abstractions over predicate variables, for example $\lambda X\ P$, where the arities of X and P must coincide. Moreover, the arity of $\lambda X\ P$ is also the same. Throughout this thesis and for the purposes of IFP we assume that *all operators are strictly positive*. Strict positivity of expressions is defined in fig. 9.3. In a general sense, a free occurrence of a predicate variable is strictly positive if it is not in the left part of an implication.

The application of an operator to a predicate is defined as $(\lambda X\, P)(Q) = P[Q/X]$. When a predicate P is defined as the least or the greatest fixed point, we also write $P \stackrel{\mu}{=} \Phi(P)$ and $P \stackrel{\nu}{=} \Phi(P)$ accordingly. If the operator Φ has the form $\lambda X \lambda \vec{x}\ A$, then we also write $P(\vec{x}) \stackrel{\mu}{=} A[P/X]$ and $P(\vec{x}) \stackrel{\nu}{=} A[P/X]$. The equivalence relation ($\equiv$) for formulas is defined as $A \equiv B \stackrel{\text{Def}}{=} A \leftrightarrow B$ and for predicates as $P \equiv Q \stackrel{\text{Def}}{=} \forall \vec{x}\,(P(\vec{x}) \equiv Q(\vec{x}))$. We use $\oplus$ to denote an exclusive or; $\exists! x P(x)$ denotes that there is only one unique x, i.e. $\exists x\, P(x) \wedge \neg \exists y\, (P(y) \wedge x \neq y)$. Inclusion is expressed by $\subseteq$. It is defined as $P \subseteq Q \stackrel{\text{Def}}{=} \forall \vec{x}(P(\vec{x}) \to Q(\vec{x}))$.

An expression is *regular* if it contains only inductive predicates $\mu \Phi$ and coinductive predicates $\nu \Phi$, where the operator Φ is s.p.. We assume that *all expressions are regular*.

The version of IFP presented in this thesis is untyped. It is possible to introduce types for extra precision as it is done in [24], however, this is not essential. This untyped representation is sufficient as we view types as implicit.

The proof rules of IFP are outlined in fig. 9.4. They include the usual natural deduction rules for intuitionistic first-order logic as well as rules for induction and coinduction, closure and coclosure, and equality. We also add variations of induction and coinduction rules. These include strong induction (SI), half-strong induction (HSI), strong coinduction (SC), and half-strong coinduction (HSC). Technically, the variations can be derived from the standard versions of induction and coinduction and, therefore, may seem redundant. However, they are useful from the practical perspective, making the proving process less cumbersome.

IFP also permits axioms, which have to be closed non-computational formulas. By *non-computational (nc)* we mean that they contain neither disjunctions nor free predicate variables. We will see that the requirement for axioms to be non-computational makes sense because it does not break the Soundness Theorem.

Initially, IFP included only one rule for induction and another one for coinduction. However, strengthened versions of these rules were introduced in IFP in [24] for convenience. Since the operator Φ is monotone, we have the following inclusions:

$$\Phi(P \cap \mu(\Phi)) \subseteq \Phi(P) \cap \mu(\Phi) \subseteq \Phi(P)$$

$$\Phi(P \cup \nu(\Phi)) \supseteq \Phi(P) \cup \nu(\Phi) \supseteq \Phi(P)$$

This means that the premises in case of *strong* and *half strong (co)induction* are weaker than those in case of the ordinary (co)induction, showing that the variants are indeed strengthenings of the original versions. The derivability of the variants from the original version is shown in [16].

We include strong and half-strong (co)induction in IFP to have it aligned with the practical use of the IFP-based proof assistant. This provides users with flexibility during the process of proof construction. However, from the purely theoretical point of view these rules are not required as they are derivable.

9.2 IFP'

In the next section we look at RIFP, an extension of IFP, which also includes realizers. The link between IFP and RIFP was proven via the soundness theorem in [21] albeit with the admissibility condition mentioned in the introduction on page 4. To lift the restriction caused by that condition, an intermediate system of IFP' was introduced in [24]. It allowed the use of unrestricted strictly positive inductive and coinductive definitions.

Natural deduction:

Use rule	Proof by axiom
$\Gamma, A \vdash A$	$\Gamma \vdash A$, where A is in a set of axioms. Only non-computational axioms are allowed.

	Introduction	Elimination
$\wedge$	$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge^+$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge^-_{\mathrm{l}}$ $\quad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge^-_{\mathrm{r}}$
$\to$	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to^+$	$\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to^-$
$\vee$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee^+_{\mathrm{l}}$ $\quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee^+_{\mathrm{r}}$	$\dfrac{\Gamma \vdash A \vee B \qquad \Gamma \vdash A \to C \qquad \Gamma \vdash B \to C}{\Gamma \vdash C}\ \vee^-$
		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \mathrm{efq}$
$\forall$	$\dfrac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x A(x)}\ \forall^+$	$\dfrac{\Gamma \vdash \forall x A(x)}{\Gamma \vdash A(t)}\ \forall^-$
$\exists$	$\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x A(x)}\ \exists^+$	$\dfrac{\Gamma \vdash \exists x A(x) \qquad \Gamma \vdash \forall x (A(x) \to B)}{\Gamma \vdash B}\ \exists^-$

$\forall^+$ is subject to a variable condition that $x \notin FV(\Gamma)$. $\exists^-$ is also subject to a variable condition that $x \notin FV(B)$.

(Co)induction, (co)closure and variations:

$\dfrac{\Gamma \vdash \Phi(P) \subseteq P}{\Gamma \vdash \mu\Phi \subseteq P}\ \mathrm{IND}$	$\dfrac{\Gamma \vdash P \subseteq \Phi(P)}{\Gamma \vdash P \subseteq \nu\Phi}\ \mathrm{COIND}$
$\dfrac{\Gamma \vdash \Phi(P \cap \mu\Phi) \subseteq P}{\Gamma \vdash \mu\Phi \subseteq P}\ \mathrm{SI}$	$\dfrac{\Gamma \vdash P \subseteq \Phi(P \cup \nu\Phi)}{\Gamma \vdash P \subseteq \nu\Phi}\ \mathrm{SC}$
$\dfrac{\Gamma \vdash \Phi(P) \cap \mu\Phi \subseteq P}{\Gamma \vdash \mu\Phi \subseteq P}\ \mathrm{HSI}$	$\dfrac{\Gamma \vdash P \subseteq \Phi(P) \cup \nu\Phi}{\Gamma \vdash P \subseteq \nu\Phi}\ \mathrm{HSC}$
$\dfrac{}{\Gamma \vdash \Phi(\diamond\Phi) \subseteq \diamond\Phi}\ \mathrm{cl},\ \diamond \in \{\mu, \nu\}$	$\dfrac{}{\Gamma \vdash \diamond\Phi \subseteq \Phi(\diamond\Phi)}\ \mathrm{cocl},\ \diamond \in \{\mu, \nu\}$

Equational reasoning:

$$\dfrac{}{\Gamma \vdash t = t}\ \mathrm{refl} \qquad \dfrac{\Gamma \vdash s = t}{\Gamma \vdash t = s}\ \mathrm{sym} \qquad \dfrac{\Gamma \vdash A[s/x] \qquad \Gamma \vdash s = t}{\Gamma \vdash A[t/s]}\ \mathrm{cong}$$

Figure 9.4: Inference rules of IFP

$\dfrac{\Gamma \vdash \Phi(P) \subseteq P \qquad Mon(\Phi)}{\Gamma \vdash \mu\Phi \subseteq P}$ IND'	$\dfrac{\Gamma \vdash P \subseteq \Phi(P) \qquad Mon(\Phi)}{\Gamma \vdash P \subseteq \nu\Phi}$ COIND'
$\dfrac{\Gamma \vdash \Phi(P \cap \mu\Phi) \subseteq P \qquad Mon(\Phi)}{\Gamma \vdash \mu\Phi \subseteq P}$ SI'	$\dfrac{\Gamma \vdash P \subseteq \Phi(P \cup \nu\Phi) \qquad Mon(\Phi)}{\Gamma \vdash P \subseteq \nu\Phi}$ SC'
$\dfrac{\Gamma \vdash \Phi(P) \cap \mu\Phi \subseteq P \qquad Mon(\Phi)}{\Gamma \vdash \mu\Phi \subseteq P}$ HSI'	$\dfrac{\Gamma \vdash P \subseteq \Phi(P) \cup \nu\Phi \qquad Mon(\Phi)}{\Gamma \vdash P \subseteq \nu\Phi}$ HSC'

Figure 9.5: Induction and coinduction in IFP'

IFP' rules coincide with the IFP rules, apart from those for induction and coinduction. In IFP', we add a premise, requiring the monotonicity of the operator Φ. In our implementation, the monotonicity check is performed automatically when converting from IFP to IFP', so there is no additional proof obligation for the user. The presence of this extra premise is useful when doing proofs by induction on IFP' rules. Monotonicity is defined below. X and Y are fresh variables.

$$Mon(\Phi) \stackrel{\text{Def}}{=} X \subseteq Y \rightarrow \Phi(X) \subseteq \Phi(Y) \tag{9.1}$$

These new rules for all variations of induction and coinduction are presented in fig. 9.5. They have a condition that free assumptions in the monotonicity proof must not contain X or Y free.

While the initial proof of the soundness theorem showed the direct link between IFP and RIFP, the version of the soundness presented later in this thesis uses IFP' instead of IFP. In practice, IFP proofs are translated into IFP' proofs and supplemented with the proof of monotonicity of the operator. Therefore, if we can show (a) that monotonicity of s.p. operators can be proven in IFP' and (b) that if IFP proves A, then IFP' proves A, it suffices to state the soundness theorem in the form that is given in this thesis. These proofs will be given before the Soundness Theorem in chapter 10.

9.3 Program semantics and RIFP

RIFP, or *Intuitionistic fixed point logic with realizers*, is a version of IFP enriched by a new sort δ and terms of sort δ (denoted M, N, K, L), which are called programs. Therefore, special predicate variables $\tilde{X}$, $\tilde{Y}$ are introduced. These predicates are extended versions of the corresponding IFP predicate variables X, Y, which admit an extra argument of sort δ for realizers. The notions of formula, predicate, operator and the rules are also updated in a similar manner.

The programs that we extract are untyped. The use of typed programs is possible, as explained in [24]. Yet, that leads to an unnecessarily complicated realizability interpretation, which we avoid here without compromising the correctness of the extracted programs.

The denotational semantics of the programming language is expressed via a Scott domain of realizers, defined as the solution to the following recursive domain equation.

$$D = (\mathbf{Nil} + \mathbf{Lt}(D) + \mathbf{Rt}(D) + \mathbf{Pair}(D \times D) + \mathbf{F}(D \to D))_\bot$$

where $+$ is the disjoint union and $\times$ is the Cartesian product of domains. Here, **Nil**, **Lt**, **Rt**, and **Pair** are *constructors*. **Lt**, **Rt**, and **Pair** are program equivalents of introduction rules. For example, $\land^+$ corresponds to pairing; in case of $\lor^+$ we have injections: $\mathbf{Lt}(d)$ and $\mathbf{Rt}(d)$ (where $d \in D$) respectively correspond to the left and right injection. The **F** label used with $D \to D$ denotes the continuous function space. The $\bot$ subscript adds a bottom element, which is the least element of the domain D and specifically useful when working with infinite data.

Initially in [21] we distinguished between programs denoting elements of D and function terms denoting continuous functions on D:

$$\begin{array}{lcl} \mathit{Programs} \ni M,N & ::= & a,b \text{ (variables)} \mid \mathbf{Nil} \mid \mathbf{Lt}(M) \mid \mathbf{Rt}(M) \mid \mathbf{Pair}(M,N) \mid \\ & & \mathbf{case}(M,\alpha,\beta) \mid \mathbf{proj}_i(M)(i \in \{0,1\}) \mid \alpha\ M \mid \\ & & \mathbf{F}(\alpha) \mid \mathbf{rec}(\alpha) \\ \mathit{Function\ terms} \ni \alpha,\beta & ::= & f,g \text{ (variables)} \mid \lambda a\ M \mid \mathbf{app}(M) \end{array}$$

With that in mind, RIFP also included several axioms for converting between functions and programs. Later on, inspired by the implementation process, we revised this definition, removing function terms and adding new program constructs. This made the definition simpler and more readable (see fig. 9.6). It is used in both, [24] and [22].

In fig. 9.6 a, b denote program variables of the sort δ. The elements inside the brackets of $\mathbf{case}\ M\ \mathbf{of}\ \{Cl_1;\ldots;Cl_n\}$ are *clauses*. These clauses are the form $C(a_1,\ldots,a_k) \to N$, where C stands for a constructor **Lt**, **Rt**, or **Pair**, $a_1,\ldots,a_n$ are variables, and N is a program.

The program construct $\mathbf{rec}M$ computes the least fixed point of M, making M recursive. Since we are not concerned with types in this thesis, we could have handled recursion using a Y-combinator, commonly used in presentations of untyped lambda calculus. However, we introduce **rec** both as a convenience and for efficiency in our implementation. We use $\bot$ for convenience to denote “undefined”. For example, $\mathbf{rec}\,(\lambda x\ x)$ has similar semantics to $\bot$ because it is a loop that never produces anything, so it is “undefined”.

$$
\begin{array}{rcl}
\mathit{Programs} \ni M,N & ::= & a,b \;\; \text{(program variables, i.e. variables of sort } \delta\text{)} \\
& | & \mathbf{Nil} \mid \mathbf{Lt}(M) \mid \mathbf{Rt}(M) \mid \mathbf{Pair}(M,N) \\
& | & \mathbf{case}\ M\ \mathbf{of}\ \{Cl_1;\ldots;Cl_n\} \\
& | & \lambda a M \\
& | & M N \\
& | & \mathbf{rec} M \\
& | & \bot
\end{array}
$$

Figure 9.6: Definition of programs

$$
\begin{array}{rcl}
M \circ N & \stackrel{\text{Def}}{=} & \lambda a\ M(N a) \\
[M+N] & \stackrel{\text{Def}}{=} & \lambda c\ \mathbf{case}\ c\ \mathbf{of}\ \{\mathbf{Lt}(a) \to M a; \mathbf{Rt}(b) \to N b\} \\
\langle M,N \rangle & \stackrel{\text{Def}}{=} & \lambda c\ \mathbf{Pair}(M c, N c) \\
\pi_{\mathbf{Lt}}(M) & \stackrel{\text{Def}}{=} & \mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{Pair}(a,b) \to a\} \\
\pi_{\mathbf{Rt}}(M) & \stackrel{\text{Def}}{=} & \mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{Pair}(a,b) \to b\}
\end{array}
$$

Figure 9.7: Abbreviations for operations on programs [24]

$$
\begin{array}{rcl}
[\![a]\!]\eta & = & \eta(a) \\
[\![C(M_1,\ldots,M_k)]\!]\eta & = & C([\![M_1]\!]\eta,\ldots,[\![M_k]\!]\eta) \\
[\![\mathbf{case}\ M\ \mathbf{of}\ \{Cl_1;\ldots;Cl_n\}]\!]\eta & = & [\![K]\!]\eta[\vec{a} \mapsto \vec{d}] \quad \text{if } [\![M]\!]\eta = C(\vec{d}) \\
& & \quad \text{and some } Cl_i \text{ is of the form } C(\vec{a}) \to K \\
[\![\lambda a\ M]\!]\eta & = & \mathbf{F}(f) \quad \text{where } f(d) = [\![M]\!]\eta[a \mapsto d] \\
[\![M N]\!]\eta & = & f([\![N]\!]\eta) \quad \text{if } [\![M]\!]\eta = \mathbf{F}(f) \\
[\![\mathbf{rec} M]\!]\eta & = & \text{the least fixed point of } f \\
& & \quad \text{if } [\![M]\!]\eta = \mathbf{F}(f) \\
[\![M]\!]\eta & = & \bot \quad \text{in all other cases}
\end{array}
$$

Figure 9.8: Denotational semantics [24]

The notions of program composition, sum, pairing, and left/right projections are defined in fig. 9.7 by desugaring. With respect to the denotational semantics, every program M denotes a corresponding element $[\![M]\!]\eta \in D$. Here η stands for the environment, which maps the free variables of M to elements of D. For closed terms the environment can be omitted as it is empty. See fig. 9.8.

Additionally, the denotational semantics of programs is reflected through a number of universally quantified axioms presented below. We consider these axioms simply as facts in our standard semantics. They are needed to prove soundness formally, even though they are not used in the proof of Theorem 1 explicitly. These axioms are also added to RIFP as per [24]. This time there is no restriction for axioms to be non-computational formulas as realizability interpretation is not applied to RIFP. In the following axioms $\mathbf{fun}(a)$ stands for $\exists b(a = \lambda c\,(b\,c))$.

(i) $\mathbf{case}\ C_i(\vec{b})\ \mathbf{of}\ \{C_1(\vec{a}_1) \to M_1; \ldots; C_n(\vec{a}_n) \to M_n\} = M_i[\vec{b}/\vec{a}_i]$

(ii) $\bigwedge_i \forall \vec{b}\, a \neq C_i(\vec{b}) \to \mathbf{case}\ a\ \mathbf{of}\ \{C_1(\vec{a}_1) \to M_1; \ldots; C_n(\vec{a}_n) \to M_n\} = \bot$

(iii) $(\lambda b M)\, a = M[a/b]$

(iv) $\neg \mathbf{fun}(a) \to a\, b = \bot$

(v) $\mathbf{fun}(a) \land \mathbf{fun}(b) \land \forall c\, (a\, c = b\, c) \to a = b$

(vi) $\bigoplus_{C \text{ constructor}} \exists! \vec{b}\, (a = C(\vec{b}))\ \oplus\ \mathbf{fun}(a)\ \oplus\ (a = \bot)$

(vii) $\mathbf{rec}\, a = a\, (\mathbf{rec}\, a)$

(viii) $P(\bot) \land \forall b\, (P(b) \to P(a\, b)) \to P(\mathbf{rec}\, a)$

The justification of the first three axioms is straightforward. The first axiom corresponds to the usual reduction of a case analysis construct. The second axiom states that a case analysis of a program that does not evaluate to an application of a constructor is undefined. The third axiom corresponds to standard β-reduction.

The fourth axiom means that if a is not an abstraction then applying it to b yields "undefined". The fifth axiom corresponds to Leibniz equality, saying that if both a and b are abstractions and applying c to each of them yields the same result, then a and b are equal. The sixth axiom means that any program is equivalent to a constructor application, an abstraction, or is undefined. The seventh axiom states that **rec** computes a fixed point of its argument.

In the last axiom P must be of the form $\lambda a A$, where A is constructed by disjunction, conjunction and universal quantification applied to equations between programs. We call such P *admissible*. The last axiom is a restricted form of

fixed point (Scott) induction. In general, for Scott induction we need predicates, which are closed under suprema of chains, a property that the admissible predicates possess. This property can be shown by induction on admissible predicates.

Now, with RIFP defined, we can gradually move to realizability interpretation, which is the base for program extraction. However, before we do so, we look at filtering out irrelevant information from proofs first. The distinction between Harrop and non-Harrop expressions, presented in the following section, is the key to such filtering.

9.4 Harrop expressions

To extract programs, which contain only computationally relevant information, we want to distinguish expressions that are computationally non-trivial from the irrelevant content included in the proofs.

A fairly restrictive notion of an expression with computationally trivial content is the class of *non-computational (nc)* expressions, which contain no disjunctions and no free predicate variables. However, for our purposes, we only need computational content in a strictly positive position of an expression. Hence, we only need to filter out expressions based on the more general Harrop property: a *Harrop expression* is an expression containing neither disjunction nor a free predicate variable in a strictly positive position. We define the Harrop property for formulas, predicates and operators simultaneously. The definition is given in fig. 9.9.

Since the language of IFP is richer, the previous notion of Harrop formulas given in chapter 7 is not fully suitable for our purposes. We need to modify and extend it by considering different types of expressions, which also contain predicate variables and fixed points of operators. Moreover, the original definitions by Rasiowa and Harrop are in the context of arithmetic, while in our system (IFP, see section 9.1) variables range over abstract objects, which makes our approach to existential quantifiers different.

In the original definition, an existential quantifier claims the existence of a natural number, which includes a construction of this number. In case of IFP claiming the existence of an object does not include the claim of a construction of this object. Hence, we permit existential quantifiers without any restrictions, so an existential formula can be Harrop.

As an example, in our system the arithmetic understanding of existential quantification is expressed by the bounded existential quantifier $\exists x\,(\mathbb{N}(x) \wedge A(x))$ (instead of just $\exists x A(x)$). On the surface, this formula appears to be Harrop by our new definition, however, we also need to consider the predicates that this expression contains. Here the predicate $\mathbb{N}$ involves a disjunction (see page 148,

$P(\vec{t})$ is Harrop iff P is Harrop

$A \wedge B$ is Harrop iff A and B are Harrop.

$A \vee B$ is not Harrop.

$A \to B$ is Harrop iff B is Harrop.

$\diamond x A$ where $\diamond \in \{\forall, \exists\}$ is Harrop iff A is Harrop.

X is not Harrop, where X is a predicate variable.

P_0 is Harrop, where P_0 is predicate constant.

$\lambda \vec{x} A$ is Harrop iff A is Harrop.

$\diamond \Phi$ where $\diamond \in \{\mu, \nu\}$ is Harrop iff Φ is Harrop.

$\lambda X\, P$ is Harrop iff $P[\hat{X}/X]$ is Harrop where $\hat{X}$ is a fresh predicate constant.

Figure 9.9: Definition of the Harrop property

intuitively a natural number is 0 *or* a successor). This means that this expression is computationally non-trivial not because it is an existential formula but because it is an expression, which contains a non-Harrop predicate.

When analysing an operator Φ, which is an abstraction $\lambda X\, P$, we need to understand whether the abstracted expression is Harrop. However, just looking at the Harrop property of P won't suffice here. X is a predicate variable, which might appear free in P, hence making P non-Harrop. For instance, if $P = \lambda x (X \wedge X)$, then P is non-Harrop. However, X is a recursion variable and its instances will be substituted eventually. With that in mind, we want to understand whether there is anything else in Φ other than those instances of X that makes Φ non-Harrop. In order to do this, we formally substitute X in P by a fresh predicate constant $\hat{X}$ and then check whether $P[\hat{X}/X]$ is Harrop. Clearly, $\lambda x (\hat{X} \wedge \hat{X})$ is Harrop.

The Harrop property is stable under substitution, which is expressed by the following lemma.

Lemma 1 For an expression E the following holds:

(a) If P is Harrop and E is Harrop, then $E[P/\hat{X}]$ is also Harrop;

(b) If E is non-Harrop, then $E[P/\hat{X}]$ is non-Harrop without restriction on P;

(c) If P is non-Harrop and E is non-Harrop, then $E[P/X]$ is also non-Harrop;

(d) If E is Harrop, then $E[P/X]$ is Harrop without restriction on P.

Proof. Proof by straightforward induction on E. □

9.5 Realizability for program extraction

In this section we introduce the notion of a *realizer*, which intuitively is the bearer of the computation content of the expression it realizes.

We formalize realizability in RIFP by simultaneously defining predicates $\mathbf{R}(E)$ and $\mathbf{H}(E)$ for every expression E. We refer to the first one as the realizability interpretation of E and the second one as a simplified realizability interpretation. Intuitively this means that in the first case the expression has computational content, while in the second case it does not. More precisely, for every

- non-Harrop formula A we define a predicate $\mathbf{R}(A)$ of arity (δ);
- non-Harrop predicate P of arity $(\vec{\iota})$ we define a predicate $\mathbf{R}(P)$ with an extra argument for realizers, that is $(\vec{\iota}, \delta)$;
- non-Harrop operator Φ of arity $(\vec{\iota})$ we define an operator $\mathbf{R}(\Phi)$ with an extra argument for realizers;
- Harrop formula A we define a formula $\mathbf{H}(A)$;
- Harrop predicate P we define a predicate $\mathbf{H}(P)$ of the same arity;
- Harrop operator Φ we define an operator $\mathbf{H}(\Phi)$ of the same arity.

Every IFP predicate variable X of arity $(\vec{\iota})$ gets a corresponding RIFP equivalent $\tilde{X}$ of arity $(\vec{\iota}, \delta)$. The realizability interpretation is presented in detail in fig. 9.10. Note that we sometimes use $a\,\mathbf{r}\,A$ instead of $\mathbf{R}(A)(a)$ for convenience and to stress visually that a is a realizer of A. These two notations are equivalent. For brevity, we use $\mathbf{H}_X(P)$ for $(\mathbf{H}(P[\hat{X}/X]))[X/\hat{X}]$, which is a temporary substitution of the predicate variable by a fresh predicate constant discussed in the previous section.

$$
\begin{array}{rll}
\mathbf{R}(A) &= \lambda a\ (\mathbf{H}(A) \wedge a = \mathbf{Nil}) & (A \text{ Harrop}) \\
\mathbf{R}(P(t)) &= \lambda a\ \mathbf{R}(P)(t,a) & (P(t) \text{ non-Harrop; } P \text{ is not an abstraction}) \\
\mathbf{H}(P(t)) &= \mathbf{H}(P)(t) & (P(t) \text{ Harrop}) \\
\mathbf{R}(A \wedge B) &= \lambda c\ (\mathbf{R}(A)(\pi_{\mathbf{Lt}}c) \wedge \mathbf{R}(B)(\pi_{\mathbf{Rt}}c)) & (A, B \text{ non-Harrop}) \\
\mathbf{R}(A \wedge B) &= \lambda a\ (\mathbf{R}(A)(a) \wedge \mathbf{H}(B)) & (A \text{ non-Harrop, } B \text{ Harrop}) \\
\mathbf{R}(A \wedge B) &= \lambda b\ (\mathbf{H}(A) \wedge \mathbf{R}(B)(b)) & (A \text{ Harrop, } B \text{ non-Harrop}) \\
\mathbf{H}(A \wedge B) &= \mathbf{H}(A) \wedge \mathbf{H}(B) & (A, B \text{ Harrop}) \\
\mathbf{R}(A \vee B) &= \lambda c\ (\exists a(c = \mathbf{Lt}(a) \wedge \mathbf{R}(A)(a)) & \\
& \qquad \vee \exists b(c = \mathbf{Rt}(b) \wedge \mathbf{R}(B)(b))) & \\
\mathbf{R}(A \to B) &= \lambda f\ (\forall a(\mathbf{R}(A)(a) \to \mathbf{R}(B)(f\,a))) & (A, B \text{ non-Harrop}) \\
\mathbf{R}(A \to B) &= \lambda b\ (\mathbf{H}(A) \to \mathbf{R}(B)(b)) & (A \text{ Harrop, } B \text{ non-Harrop}) \\
\mathbf{H}(A \to B) &= \exists a\ \mathbf{R}(A)(a) \to \mathbf{H}(B) & (B \text{ non-Harrop}) \\
\mathbf{R}(\diamond x\ A) &= \lambda a\ \mathbf{R}(\diamond x\ (A))(a) = \lambda a\ \diamond x\ (\mathbf{R}(A)(a)) & (A \text{ non-Harrop; } \diamond \in \{\forall, \exists\}) \\
\mathbf{H}(\diamond x\ A) &= \diamond x\ \mathbf{H}(A) & (A \text{ Harrop; } \diamond \in \{\forall, \exists\}) \\
& & \\
\mathbf{R}(P) &= \lambda(\vec{x}, a)(\mathbf{H}(P) \wedge a = \mathbf{Nil}) & (P \text{ Harrop}) \\
\mathbf{R}(X) &= \tilde{X} & (X \text{ predicate variable}) \\
\mathbf{H}(P_0) &= P_0 & (P_0 \text{ predicate constant}) \\
\mathbf{R}(\lambda\vec{x}\ A) &= \lambda(\vec{x}, a)\ \mathbf{R}(A)(a) & (\text{Abstraction}) \\
\mathbf{R}(\diamond(\Phi)) &= \diamond(\mathbf{R}(\Phi)) & (\Phi \text{ non-Harrop; } \diamond \in \{\mu, \nu\}) \\
\mathbf{H}(\diamond(\Phi)) &= \diamond(\mathbf{H}(\Phi)) & (\Phi \text{ Harrop; } \diamond \in \{\mu, \nu\}) \\
& & \\
\mathbf{R}(\lambda X\ P) &= \lambda\tilde{X}\ \mathbf{R}(P) & (P \text{ any predicate}) \\
\mathbf{H}(\lambda X\ P) &= \lambda X\ \mathbf{H}_X(P) & (P \text{ is s.p. in } X)
\end{array}
$$

Figure 9.10: Realizability interpretation

To prove the soundness of the system, we first need to show that certain properties of expressions hold under substitution.

Lemma 2 Let E be an IFP expression (predicate, operator or formula) and P an IFP predicate, then the following hold:

(a) If P is non-Harrop then $\mathbf{R}(E[P/X]) = \mathbf{R}(E)[\mathbf{R}(P)/\tilde{X}]$.

(b) If P is Harrop then $\mathbf{R}(E[P/X]) = \mathbf{R}(E[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$.

(a) and (b) hold for arbitrary E. For the case that E is a Harrop expression, (a) is equivalent to

(c) If P is non-Harrop then $\mathbf{H}(E[P/X]) = \mathbf{H}(E)[\mathbf{R}(P)/\tilde{X}]$

and (b) is equivalent to

(d) If P is Harrop then $\mathbf{H}(E[P/X]) = \mathbf{H}(E[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$.

Proof. We start with (c) and (d), which follow directly from (a) and (b) respectively simply by unfolding the definition of $\mathbf{R}$. Note, that for (c), since E is Harrop, no free predicate variable appears in a strictly positive position. Consequently, following the definition of $\mathbf{H}$ and $\mathbf{R}$, all predicate variables will be replaced with their "tilde" counterpart, including X with $\tilde{X}$. This means that the arities of $\mathbf{R}(P)$ and $\tilde{X}$ will match. To show (c), we rewrite (a) as below.

$$\mathbf{R}(E[P/X]) = \mathbf{R}(E)[\mathbf{R}(P)/\tilde{X}]$$
$$(\lambda a\ \mathbf{H}(E[P/X]) \land a = \mathbf{Nil}) = (\lambda a\ \mathbf{H}(E)[\mathbf{R}(P)/\tilde{X}] \land a = \mathbf{Nil}) \quad \text{since } E \text{ is Harrop}$$

Similarly, for (d) we rewrite (b):

$$\mathbf{R}(E[P/X]) = \mathbf{R}(E[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$$
$$(\lambda a\ \mathbf{H}(E[P/X]) \land a = \mathbf{Nil}) = (\lambda a\ \mathbf{H}(E[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] \land a = \mathbf{Nil}) \quad \text{since } E \text{ is Harrop}$$

(a) and (b) are proven by induction on the size of expressions.

We begin with the proof of (a), starting with predicates, where for most cases the proof is straightforward. The most interesting are the cases of predicate variables and fixed points.

For a predicate variable, substitution will only occur if the predicate is the one that will be substituted. With that in mind, $\mathbf{R}(X[P/X]) = \mathbf{R}(P)$, which is the same as the result of this substitution $\mathbf{R}(X)[\mathbf{R}(P)/\tilde{X}] = \tilde{X}[\mathbf{R}(P)/\tilde{X}] = \mathbf{R}(P)$.

For a fixed point, i.e., case $E = \diamond(\Phi)$, where $\diamond \in \{\mu, \nu\}$. We show $\mathbf{R}(\diamond(\Phi)[P/X]) = \mathbf{R}(\diamond(\Phi))[\mathbf{R}(P)/\tilde{X}]$.

$$\begin{array}{rll} & \mathbf{R}(\diamond(\Phi)[P/X]) & \\ = & \mathbf{R}(\diamond(\Phi[P/X])) & \text{subst.} \\ = & \diamond(\mathbf{R}(\Phi[P/X])) & \text{def. realiz.} \\ = & \diamond(\mathbf{R}(\Phi)[\mathbf{R}(P)/\tilde{X}]) & \text{by i.h.} \\ = & \diamond(\mathbf{R}(\Phi))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\ = & \mathbf{R}(\diamond(\Phi))[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.} \end{array}$$

Case $E = Q(s)$, where Q is not an abstraction. We need to show $\mathbf{R}(Q(s)[P/X]) = \mathbf{R}(Q(s))[\mathbf{R}(P)/\tilde{X}]$.

$$\begin{array}{rll} & \mathbf{R}(Q(s)[P/X]) & \\ = & \mathbf{R}(Q[P/X])(s) & \text{subst.} \\ = & \lambda a\ (\mathbf{R}(Q[P/X])(s,a)) & \text{def. realiz.} \\ = & \lambda a\ (\mathbf{R}(Q[\mathbf{R}(P)/\tilde{X}])(s,a)) & \text{by i.h.} \\ = & (\lambda a\ \mathbf{R}(Q)(s,a))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\ = & \mathbf{R}(Q(s))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \end{array}$$

Now for the operator case $\Phi = \lambda Y\ Q$, we assume w.l.o.g. that $Y \neq X$ and $Y \notin FV(P)$. We need to show $\mathbf{R}(\lambda Y Q[P/X]) = \mathbf{R}(\lambda Y Q)[\mathbf{R}(P)/\tilde{X}]$.

$$\begin{array}{rll} & \mathbf{R}((\lambda Y Q)[P/X]) & \\ = & \mathbf{R}(\lambda Y(Q[P/X])) & \text{subst.} \\ = & \lambda\tilde{Y}\mathbf{R}(Q[P/X]) & \text{def. realiz.} \\ = & \lambda\tilde{Y}(\mathbf{R}(Q)[\mathbf{R}(P)/\tilde{X}]) & \text{by i.h.} \\ = & (\lambda\tilde{Y}(\mathbf{R}(Q))[\mathbf{R}(P)/\tilde{X}]) & \text{subst.} \\ = & \mathbf{R}(\lambda Y Q)[\mathbf{R}(P)/\tilde{X}] & \text{subst and def. realiz.} \end{array}$$

Case $A \wedge B$ (non-Harrop). We need to show $\mathbf{R}((A \wedge B)[P/X]) = \mathbf{R}(A \wedge B)[\mathbf{R}(P)/\tilde{X}]$.
Case A and B are non-Harrop:

$$\begin{array}{rcl} & \mathbf{R}((A \wedge B)[P/X]) & \\ = & \mathbf{R}(A[P/X] \wedge B[P/X]) & \text{subst.} \\ = & \lambda a\ (\mathbf{R}(A[P/X])(\pi_{\mathbf{Lt}}(a)) \wedge \mathbf{R}(B[P/X])(\pi_{\mathbf{Rt}}(a))) & \text{def. realiz., lemma 1(c)} \\ = & \lambda a\ (\mathbf{R}(A)[\mathbf{R}(P)/\tilde{X}](\pi_{\mathbf{Lt}}(a)) \wedge \mathbf{R}(B)[\mathbf{R}(P)/\tilde{X}](\pi_{\mathbf{Rt}}(a))) & \text{by i.h.} \\ = & \lambda a\ (\mathbf{R}(A)(\pi_{\mathbf{Lt}}(a)) \wedge \mathbf{R}(B)(\pi_{\mathbf{Rt}}(a)))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\ = & \mathbf{R}(A \wedge B)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.} \end{array}$$

Case A is non-Harrop and B is Harrop:

$$\begin{array}{rcl}
& \mathbf{R}((A \wedge B)[P/X]) & \\
= & \mathbf{R}(A[P/X] \wedge B[P/X]) & \text{subst.} \\
= & \lambda a\ (\mathbf{R}(A[P/X])(a)) \wedge \mathbf{H}(B[P/X]) & \text{def. realiz., lemma 1(c,d)} \\
= & \lambda a\ (\mathbf{R}(A)[\mathbf{R}(P)/\tilde{X}](a)) \wedge \mathbf{H}(B)[\mathbf{R}(P)/\tilde{X}] & \text{by i.h.} \\
= & (\lambda a\ (\mathbf{R}(A)(a)) \wedge \mathbf{H}(B))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\
= & \mathbf{R}(A \wedge B)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.}
\end{array}$$

Case A is Harrop and B is non-Harrop is proven using the similar approach.

<u>Case $A \wedge B$</u> (Harrop). Here both A and B are Harrop and so are $A[P/X]$ and $B[P/X]$. Again, we show $\mathbf{R}((A \wedge B)[P/X]) = \mathbf{R}(A \wedge B)[\mathbf{R}(P)/\tilde{X}]$:

$$\begin{array}{rcl}
& \mathbf{R}((A \wedge B)[P/X]) & \\
= & \mathbf{R}(A[P/X] \wedge B[P/X]) & \text{subst.} \\
= & \lambda a\ (\mathbf{H}(A[P/X] \wedge B[P/X]) \wedge a = \mathbf{Nil}) & A[P/X] \wedge B[P/X] \text{ is Harrop} \\
= & \lambda a\ (\mathbf{H}(A[P/X]) \wedge \mathbf{H}(B[P/X]) \wedge a = \mathbf{Nil}) & \text{def. realiz., lemma 1(d)} \\
= & \lambda a\ (\mathbf{H}(A)[\mathbf{R}(P)/\tilde{X}] \wedge \mathbf{H}(B)[\mathbf{R}(P)/\tilde{X}] \wedge a = \mathbf{Nil}) & \text{by i.h.} \\
= & \lambda a\ (\mathbf{H}(A) \wedge \mathbf{H}(B) \wedge a = \mathbf{Nil})[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\
= & \lambda a\ (\mathbf{H}(A \wedge B) \wedge a = \mathbf{Nil})[\mathbf{R}(P)/\tilde{X}] & A \text{ and } B \text{ are both Harrop} \\
= & \mathbf{R}(A \wedge B)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.}
\end{array}$$

<u>Case $A \vee B$</u>. Disjunction is always non-Harrop regardless of whether A and B are Harrop. We prove this for the case that A and B are both non-Harrop. The proofs for other combinations are similar – for Harrop formulas realizers are equal to **Nil**.

Here we need to show $\mathbf{R}((A \vee B)[P/X]) = \mathbf{R}(A \vee B)[\mathbf{R}(P)/\tilde{X}]$.

$$\begin{array}{rcl}
& \mathbf{R}((A \vee B)[P/X]) & \\
= & \mathbf{R}(A[P/X] \vee B[P/X]) & \text{subst.} \\
= & \lambda c\ (\mathbf{R}(A[P/X] \vee B[P/X])(c)) & \text{def. realiz.,} \\
& & \text{lemma 1(c)} \\
= & \lambda c\ (\exists a(c = \mathbf{Lt}(a) \wedge \mathbf{R}(A[P/X])(a)) \vee \exists b(c = \mathbf{Rt}(b) \wedge \mathbf{R}(B[P/X])(b))) & \text{def. realiz.} \\
= & \lambda c\ (\exists a(c = \mathbf{Lt}(a) \wedge \mathbf{R}(A)[\mathbf{R}(P)/\tilde{X}](a)) \vee \exists b(c = \mathbf{Rt}(b) \wedge \mathbf{R}(B)[\mathbf{R}(P)/\tilde{X}](b))) & \text{by i.h.} \\
= & \lambda c\ (\exists a(c = \mathbf{Lt}(a) \wedge \mathbf{R}(A)(a)) \vee \exists b(c = \mathbf{Rt}(b) \wedge \mathbf{R}(B)(b)))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\
= & \mathbf{R}(A \vee B)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.}
\end{array}$$

<u>Case $A \to B$</u> (non-Harrop). We show $\mathbf{R}((A \to B)[P/X]) = \mathbf{R}(A \to B)[\mathbf{R}(P)/\tilde{X}]$.

Case A and B are both non-Harrop:

$$\begin{array}{lcl} & \mathbf{R}((A \to B)[P/X]) & \\ = & \mathbf{R}(A[P/X] \to B[P/X]) & \text{subst.} \\ = & \lambda f(\mathbf{R}(A[P/X] \to B[P/X])(f)) & \text{def. realiz., lemma 1(c)} \\ = & \lambda f(\forall a(\mathbf{R}(A[P/X])(a) \to \mathbf{R}(B[P/X])(f\ a))) & \text{def. realiz.} \\ = & \lambda f(\forall a(\mathbf{R}(A)[\mathbf{R}(P)/\tilde{X}](a) \to \mathbf{R}(B)[\mathbf{R}(P)/\tilde{X}](f\ a))) & \text{by i.h.} \\ = & \lambda f(\forall a(\mathbf{R}(A)(a) \to \mathbf{R}(B)(f\ a)))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\ = & \mathbf{R}(A \to B)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.} \end{array}$$

Case A is Harrop and B is non-Harrop:

$$\begin{array}{lcl} & \mathbf{R}((A \to B)[P/X]) & \\ = & \mathbf{R}(A[P/X] \to B[P/X]) & \text{subst.} \\ = & \lambda a\ (\mathbf{H}(A[P/X]) \to \mathbf{R}(B[P/X])(a)) & \text{def. realiz., lemma 1(c,d)} \\ = & \lambda a\ (\mathbf{H}(A)[\mathbf{R}(P)/\tilde{X}] \to \mathbf{R}(B)[\mathbf{R}(P)/\tilde{X}](a)) & \text{by i.h.} \\ = & \lambda a\ (\mathbf{H}(A)[\mathbf{R}(P)/\tilde{X}] \to \mathbf{R}(B)(a)[\mathbf{R}(P)/\tilde{X}]) & \text{subst.} \\ = & \lambda a\ (\mathbf{H}(A) \to \mathbf{R}(B)(a))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\ = & \mathbf{R}(A \to B)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.} \end{array}$$

Case $A \to B$ (Harrop). In this case B is Harrop. We need to show $\mathbf{R}((A \to B)[P/X]) = \mathbf{R}(A \to B)[\mathbf{R}(P)/\tilde{X}]$.

Case A is non-Harrop and B is Harrop:

$$\begin{array}{lcl} & \mathbf{R}((A \to B)[P/X]) & \\ = & \mathbf{R}(A[P/X] \to B[P/X]) & \text{subst.} \\ = & \exists a\ \mathbf{R}(A[P/X])(a) \to \mathbf{H}(B[P/X]) & \text{def. realiz., lemma 1(d)} \\ = & \exists a\ \mathbf{R}(A)[\mathbf{R}(P)/\tilde{X}](a) \to \mathbf{H}(B)[\mathbf{R}(P)/\tilde{X}] & \text{by i.h.} \\ = & (\exists a\ \mathbf{R}(A)(a) \to \mathbf{H}(B))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\ = & \mathbf{R}(A \to B)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.} \end{array}$$

Case $\diamond x\ A$, where $\diamond \in \{\forall, \exists\}$ Here w.l.o.g. we assume $x \notin FV(P)$.

For formulas with quantifiers we consider Harrop and non-Harrop formulas in the same way, since in case of quantifiers the definition of realizability, $\mathbf{R}(\diamond x\ A) = \lambda a\ (\diamond x\ \mathbf{R}(A)(a))$ works for both types of formulas. If A is Harrop, $a = \mathbf{Nil}$.

We need to show $\mathbf{R}((\diamond x\, A)[P/X]) = \mathbf{R}(\diamond x\, A)[\mathbf{R}(P)/\tilde{X}]$.

$$
\begin{array}{lll}
 & \mathbf{R}((\diamond x\, A)[P/X]) & \\
= & \mathbf{R}(\diamond x\, A[P/X]) & \text{subst.} \\
= & \lambda a\ (\diamond x\ (\mathbf{R}(A[P/X])(a))) & \text{def. realiz., lemma 1(c)} \\
= & \lambda a\ (\diamond x\ (\mathbf{R}(A)[\mathbf{R}(P)/\tilde{X}](a))) & \text{by i.h.} \\
= & \lambda a\ (\diamond x\ (\mathbf{R}(A)(a)))[\mathbf{R}(P)/\tilde{X}] & \text{subst.} \\
= & \mathbf{R}(\diamond x\, A)[\mathbf{R}(P)/\tilde{X}] & \text{def. realiz.}
\end{array}
$$

To prove the statement (b), i.e., if P is Harrop then $\mathbf{R}(E[P/X]) = \mathbf{R}(E[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$, we need to prove (b′): if P is Harrop then $\mathbf{R}(E[P/\hat{X}]) = \mathbf{R}(E)[\mathbf{H}(P)/\hat{X}]$ holds for all expressions E. Here $\hat{X}$ is a predicate constant.

We obtain (b) by applying (b′) to $E[\hat{X}/X]$, that is $\mathbf{R}(E[\hat{X}/X][P/\hat{X}])$ is equal to $\mathbf{R}(E[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$. Since by substitution $E[\hat{X}/X][P/\hat{X}] = E[P/X]$, (b) holds.

The proof of (b′) is by induction on the structure of E.

For most cases when E is a predicate, the proof is straightforward.

Case $E = \diamond(\Phi)$. We show $\mathbf{R}(\diamond(\Phi)[P/\hat{X}]) = \mathbf{R}(\diamond(\Phi))[\mathbf{R}(P)/\hat{X}]$.

$$
\begin{array}{lll}
 & \mathbf{R}(\diamond(\Phi)[P/\hat{X}]) & \\
= & \mathbf{R}(\diamond(\Phi[P/\hat{X}])) & \text{subst.} \\
= & \diamond(\mathbf{R}(\Phi[P/\hat{X}])) & \text{def. realiz.} \\
= & \diamond(\mathbf{R}(\Phi)[\mathbf{R}(P)/\hat{X}]) & \text{by i.h.} \\
= & \diamond(\mathbf{R}(\Phi))[\mathbf{R}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(\diamond(\Phi))[\mathbf{R}(P)/\hat{X}] & \text{def. realiz.}
\end{array}
$$

Case $E = Q(s)$, where Q is not an abstraction. We need to show $\mathbf{R}(E[P/\hat{X}]) = \mathbf{R}(E)[\mathbf{H}(P)/\hat{X}]$:

$$
\begin{array}{lll}
 & \mathbf{R}(Q(s)[P/\hat{X}]) & \\
= & \mathbf{R}(Q[P/\hat{X}])(s) & \text{subst.} \\
= & \lambda a\ \mathbf{R}(Q[P/\hat{X}])(s,a) & \text{def. realiz.} \\
= & \lambda a\ \mathbf{R}(Q[\mathbf{H}(P)/\hat{X}])(s,a) & \text{by i.h.} \\
= & \lambda a\ \mathbf{R}(Q)(s,a)[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(Q(s))[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.}
\end{array}
$$

Now for the operator case $\underline{\Phi = \lambda Y\ Q}$ we assume w.l.o.g. that $Y \neq \hat{X}$ and $Y \notin FV(P)$. We need to show $\mathbf{R}(\lambda YQ[P/\hat{X}]) = \mathbf{R}(\lambda YQ)[\mathbf{H}(P)/\hat{X}]$.

$$
\begin{array}{lll}
& \mathbf{R}((\lambda YQ)[P/\hat{X}]) & \\
= & \mathbf{R}(\lambda Y(Q[P/\hat{X}])) & \text{subst.} \\
= & \lambda\tilde{Y}(\mathbf{R}(Q[P/\hat{X}])) & \text{def. realiz.} \\
= & \lambda\tilde{Y}(\mathbf{R}(Q)[\mathbf{H}(P)/\hat{X}]) & \text{by i.h.} \\
= & (\lambda\tilde{Y}(\mathbf{R}(Q))[\mathbf{H}(P)/\hat{X}]) & \text{subst.} \\
= & \mathbf{R}(\lambda YQ)[\mathbf{H}(P)/\hat{X}] & \text{subst.}
\end{array}
$$

$\underline{\text{Case } A \wedge B}$ (non-Harrop). We need to show $\mathbf{R}((A\wedge B)[P/\hat{X}]) = \mathbf{R}(A\wedge B)[\mathbf{H}(P)/\hat{X}]$.
Case A and B are non-Harrop:

$$
\begin{array}{lcl}
& \mathbf{R}((A\wedge B)[P/\hat{X}]) & \\
= & \mathbf{R}(A[P/\hat{X}]\wedge B[P/\hat{X}]) & \text{subst.} \\
= & \lambda a\ (\mathbf{R}(A[P/\hat{X}])(\pi_{\mathbf{Lt}}(a)) \wedge \mathbf{R}(B[P/\hat{X}])(\pi_{\mathbf{Rt}}(a))) & \text{def. realiz., lemma 1(b)} \\
= & \lambda a\ (\mathbf{R}(A)[\mathbf{H}(P)/\hat{X}](\pi_{\mathbf{Lt}}(a)) \wedge \mathbf{R}(B)[\mathbf{H}(P)/\hat{X}](\pi_{\mathbf{Rt}}(a))) & \text{by i.h.} \\
= & \lambda a\ (\mathbf{R}(A)(\pi_{\mathbf{Lt}}(a)) \wedge \mathbf{R}(B)(\pi_{\mathbf{Rt}}(a)))[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(A\wedge B)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.}
\end{array}
$$

Case A is non-Harrop and B is Harrop:

$$
\begin{array}{lcl}
& \mathbf{R}((A\wedge B)[P/\hat{X}]) & \\
= & \mathbf{R}(A[P/\hat{X}]\wedge B[P/\hat{X}]) & \text{subst.} \\
= & \lambda a\ (\mathbf{R}(A[P/\hat{X}])(a)) \wedge \mathbf{H}(B[P/\hat{X}]) & \text{def. realiz., lemma 1(a,b)} \\
= & \lambda a\ (\mathbf{R}(A)[\mathbf{H}(P)/\hat{X}](a) \wedge \mathbf{H}(B)[\mathbf{H}(P)/\hat{X}]) & \text{by i.h.} \\
= & \lambda a\ (\mathbf{R}(A)(a) \wedge \mathbf{H}(B))[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(A\wedge B)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.}
\end{array}
$$

Case A is Harrop and B is non-Harrop is proven using the similar approach.

$\underline{\text{Case } A \wedge B}$ (Harrop). Here both A and B are Harrop and so are $A[P/\hat{X}]$ and

$B[P/\hat{X}]$. Again, we need to show $\mathbf{R}((A \wedge B)[P/\hat{X}]) = \mathbf{R}(A \wedge B)[\mathbf{H}(P)/\hat{X}]$:

$$\begin{array}{cll}
 & \mathbf{R}((A \wedge B)[P/\hat{X}]) & \\
= & \mathbf{R}(A[P/\hat{X}] \wedge B[P/\hat{X}]) & \text{subst.} \\
= & \lambda a\ (\mathbf{H}(A[P/\hat{X}]) \wedge \mathbf{H}(B[P/\hat{X}]) \wedge a = \mathbf{Nil}) & \text{def. realiz., lemma 1(a)} \\
= & \lambda a\ (\mathbf{H}(A)[\mathbf{H}(P)/\hat{X}] \wedge \mathbf{H}(B)[\mathbf{H}(P)/\hat{X}] \wedge a = \mathbf{Nil}) & \text{by i.h.} \\
= & \lambda a\ (\mathbf{H}(A) \wedge \mathbf{H}(B) \wedge a = \mathbf{Nil})[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(A \wedge B)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.}
\end{array}$$

<u>Case $A \vee B$</u>. Disjunction is always non-Harrop regardless of whether A and B are Harrop. As in (a), we show a proof for the case that A and B are both non-Harrop. We need to show $\mathbf{R}((A \vee B)[P/\hat{X}]) = \mathbf{R}(A \vee B)[\mathbf{H}(P)/\hat{X}]$.

$$\begin{array}{cll}
 & \mathbf{R}((A \vee B)[P/\hat{X}]) & \\
= & \mathbf{R}(A[P/\hat{X}] \vee B[P/\hat{X}]) & \text{subst.} \\
= & \lambda c\ (\exists a(c = \mathbf{Lt}(a) \wedge \mathbf{R}(A[P/\hat{X}])(a)) \vee \exists b(c = \mathbf{Rt}(b) \wedge \mathbf{R}(B[P/\hat{X}])(b))) & \text{def. realiz.,} \\
 & & \text{lemma 1(b)} \\
= & \lambda c\ (\exists a(c = \mathbf{Lt}(a) \wedge \mathbf{R}(A)[\mathbf{H}(P)/\hat{X}](a)) \vee \exists b(c = \mathbf{Rt}(b) \wedge \mathbf{R}(B)[\mathbf{H}(P)/\hat{X}](b))) & \text{by i.h.} \\
= & \lambda c\ (\exists a(c = \mathbf{Lt}(a) \wedge \mathbf{R}(A)(a)) \vee \exists b(c = \mathbf{Rt}(b) \wedge \mathbf{R}(B)(b)))[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(A \vee B)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz}
\end{array}$$

<u>Case $A \to B$</u> (non-Harrop). We show $\mathbf{R}((A \to B)[P/\hat{X}]) = \mathbf{R}(A \to B)[\mathbf{H}(P)/\hat{X}]$.
Case A and B are both non-Harrop:

$$\begin{array}{cll}
 & \mathbf{R}((A \to B)[P/\hat{X}]) & \\
= & \mathbf{R}(A[P/\hat{X}] \to B[P/\hat{X}]) & \text{subst.} \\
= & \lambda f(\forall a(\mathbf{R}(A[P/\hat{X}])(a) \to \mathbf{R}(B[P/\hat{X}])(f\ a))) & \text{def. realiz., lemma 1(b)} \\
= & \lambda f(\forall a(\mathbf{R}(A)[\mathbf{H}(P)/\hat{X}](a) \to \mathbf{R}(B)[\mathbf{H}(P)\hat{X}](f\ a))) & \text{by i.h.} \\
= & \lambda f(\forall a(\mathbf{R}(A)(a) \to \mathbf{R}(B)(f\ a)))[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(A \to B)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.}
\end{array}$$

Case A is Harrop and B is non-Harrop:

$$\begin{array}{cll}
 & \mathbf{R}((A \to B)[P/\hat{X}]) & \\
= & \mathbf{R}(A[P/\hat{X}] \to B[P/\hat{X}]) & \text{subst.} \\
= & \lambda a\ (\mathbf{H}(A[P/\hat{X}]) \to B[P/\hat{X}](a)) & \text{def. realiz., lemma 1(a,b)} \\
= & \lambda a\ (\mathbf{H}(A)[\mathbf{H}(P)/\hat{X}] \to B[\mathbf{H}(P)/\hat{X}](a)) & \text{by i.h.} \\
= & \lambda a\ (\mathbf{H}(A) \to B(a))[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\
= & \mathbf{R}(A \to B)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.}
\end{array}$$

Case $A \to B$ (Harrop). In this case B is Harrop. We need to show $\mathbf{R}((A \to B)[P/\hat{X}]) = \mathbf{R}(A \to B)[\mathbf{H}(P)/\hat{X}]$.

Case A is non-Harrop and B is Harrop:

$$\begin{array}{rcl} & \mathbf{R}((A \to B)[P/\hat{X}]) & \\ = & \mathbf{R}(A[P/\hat{X}] \to B[P/\hat{X}]) & \text{subst.} \\ = & \exists a\ \mathbf{R}(A[P/\hat{X}])(a) \to \mathbf{H}(B[P/\hat{X}]) & \text{def. realiz., lemma 1(a,b)} \\ = & \exists a\ \mathbf{R}(A)[\mathbf{H}(P)/\hat{X}](a) \to \mathbf{H}(B)[\mathbf{H}(P)/\hat{X}] & \text{by i.h.} \\ = & (\exists a\ \mathbf{R}(A)(a) \to \mathbf{H}(B))[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\ = & \mathbf{R}(A \to B)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.} \end{array}$$

Case $\diamond x\, A$, where $\diamond \in \{\forall, \exists\}$ Here w.l.o.g. we assume $x \notin FV(P)$.

For formulas with quantifiers we consider Harrop and non-Harrop formulas in the same way, since in case of quantifiers the definition of realizability, $\mathbf{R}(\diamond x\, A) = \lambda a\ (\diamond x\ \mathbf{R}(A)(a))$, works for both types of formulas. In case A is Harrop, $a = \mathbf{Nil}$.

We need to show $\mathbf{R}((\diamond x\, A)[P/\hat{X}]) = \mathbf{R}(\diamond x\, A)[\mathbf{H}(P)/\hat{X}]$.

$$\begin{array}{rcl} & \mathbf{R}((\diamond x\, A)[P/\hat{X}]) & \\ = & \mathbf{R}(\diamond x\, A[P/\hat{X}]) & \text{subst.} \\ = & \lambda a\ (\diamond x\ (\mathbf{R}(A[P/\hat{X}])(a))) & \text{def. realiz., lemma 1(c)} \\ = & \lambda a\ (\diamond x\ (\mathbf{R}(A)[\mathbf{H}(P)/\hat{X}](a))) & \text{by i.h.} \\ = & \lambda a\ (\diamond x\ (\mathbf{R}(A)(a)))[\mathbf{H}(P)/\hat{X}] & \text{subst.} \\ = & \mathbf{R}(\diamond x\, A)[\mathbf{H}(P)/\hat{X}] & \text{def. realiz.} \end{array}$$

□

Chapter 10

Soundness

This section includes a detailed proof of the soundness theorem for IFP (Theorem 1). First, we define a number of lemmas useful for proving the soundness theorem.

Lemma 3 If IFP, IFP', or RIFP proves $\Gamma \vdash A$, then the same system proves $\Gamma[P/X] \vdash A[P/X]$, $\Gamma[P/\hat{X}] \vdash A[P/\hat{X}]$, where A, P, X are arbitrary formulas, predicates, predicate variables, respectively, and $\hat{X}$ is an arbitrary predicate constant that does not appear in any axiom.

Proof. By structural induction on derivations in IFP, IFP' and RIFP.

Axiom. Substitution has no effect in case a derivation is using an axiom. This is due to the restriction that axioms cannot contain free predicate variables.

Use rule. In case of a proof by assumption we have $\Gamma \vdash A$, where $A \in \Gamma$. It is clear that to prove $A[P/X]$ the substitution should also apply to $A \in \Gamma$.

$\wedge^+$. Assume $\Gamma \vdash (A \wedge B)$ has been derived from $\Gamma \vdash A$ and $\Gamma \vdash B$ by $\wedge^+$. By i.h., $\Gamma[P/X] \vdash A[P/X]$ and $\Gamma[P/X] \vdash B[P/X]$. Hence $\Gamma[P/X] \vdash A[P/X] \wedge B[P/X]$, by $\wedge^+$, which is the same as $\Gamma[P/X] \vdash (A \wedge B)[P/X]$.

$\wedge_l^-$. Assume $\Gamma \vdash A$ has been derived from $\Gamma \vdash A \wedge B$ by $\wedge_l^-$. By i.h. $\Gamma[P/X] \vdash (A \wedge B)[P/X]$, which is the same as $\Gamma[P/X] \vdash A[P/X] \wedge B[P/X]$. Hence, $\Gamma[P/X] \vdash A[P/X]$ by $\wedge_l^-$.

$\wedge_r^-$. Proven in a similar way.

$\vee_l^+$. Assume $\Gamma \vdash (A \vee B)$ have been derived from $\Gamma \vdash A$ by $\vee_l^+$. By i.h. $\Gamma[P/X] \vdash$

$A[P/X]$. Hence, $\Gamma[P/X] \vdash A[P/X] \vee B[P/X]$ by $\vee_l^+$, which is the same as $\Gamma[P/X] \vdash (A \vee B)[P/X]$.

$\underline{\vee_r^+}$. Proven in a similar way.

$\underline{\vee^-}$. Assume $\Gamma \vdash C$ has been derived from $\Gamma \vdash A \vee B$, $\Gamma \vdash A \to C$, and $\Gamma \vdash B \to C$ by $\vee^-$. By induction hypothesis we have the following:

- $\Gamma[P/X] \vdash (A \vee B)[P/X]$
- $\Gamma[P/X] \vdash (A \to C)[P/X]$, which is the same as $\Gamma[P/X] \vdash A[P/X] \to C[P/X]$
- $\Gamma[P/X] \vdash (B \to C)[P/X]$, which is the same as $\Gamma[P/X] \vdash B[P/X] \to C[P/X]$

$\Gamma[P/X] \vdash A[P/X] \vee B[P/X]$ is the same as $\Gamma[P/X] \vdash (A \vee B)[P/X]$, so all the premises of the $\vee^-$ rule hold. Hence, $\Gamma[P/X] \vdash C[P/X]$.

$\underline{\to^+}$. Assume $\Gamma \vdash (A \to B)$ has been derived from $\Gamma, A \vdash B$ by $\to^+$. By i.h. $\Gamma[P/X], A[P/X] \vdash B[P/X]$. Hence, $\Gamma[P/X] \vdash A[P/X] \to B[P/X]$ by $\to^+$, which is the same as $\Gamma[P/X] \vdash (A \to B)[P/X]$.

$\underline{\to^-}$ Assume $\Gamma \vdash B$ has been derived from $\Gamma \vdash A \to B$ and $\Gamma \vdash A$. By i.h. $\Gamma[P/X] \vdash (A \to B)[P/X]$ and $\Gamma[P/X] \vdash A[P/X]$. Since $\Gamma[P/X] \vdash (A \to B)[P/X]$ is the same as $\Gamma[P/X] \vdash A[P/X] \to B[P/X]$ we can apply the $\to^+$ rule. Hence, $\Gamma[P/X] \vdash B[P/X]$.

$\underline{\forall^+}$. Assume $\Gamma \vdash \forall x\, A$ has been derived from $\Gamma \vdash A$ by $\forall^+$. Here we assume that $x \notin FV(P)$ as it can be renamed if needed. By i.h. $\Gamma[P/X] \vdash A[P/X]$. Hence $\Gamma[P/X] \vdash \forall x\, A[P/X]$.

$\underline{\forall^-}$. Assume $\Gamma \vdash A[t/x]$ has been derived from $\Gamma \vdash \forall x\, A$ by $\forall^-$. By i.h. $\Gamma[P/X] \vdash \forall x\, A[P/X]$. Hence, by $\forall^-$, we get $\Gamma[P/X] \vdash A[P/X][t/x]$. Here we assume that $x \notin FV(P)$ as it can be renamed if needed, so we can swap the substitutions and obtain $\Gamma[P/X] \vdash A[t/x][P/X]$.

$\underline{\exists^+}$. Assume $\Gamma \vdash \exists x\, A$ has been derived from $\Gamma \vdash A[t/x]$. By i.h. $\Gamma[P/X] \vdash A[P/X][t/x]$. As in the previous case, we can swap the substitution to obtain $\Gamma[P/X] \vdash A[t/x][P/X]$. Hence, by $\exists^+$, $\Gamma[P/X] \vdash \exists x\, A[P/X]$.

$\underline{\exists^-}$. Assume $\Gamma \vdash B$ has been derived from the premises $\Gamma \vdash \exists x\, A(x)$ and $\Gamma \vdash (\forall x\, (A(x) \to B))$. By i.h. we have $\Gamma[P/X] \vdash \exists x\, A(x)[P/X]$ and $\Gamma[P/X] \vdash \forall x\, (A(x) \to B)[P/X]$. Applying substitution, we get $\Gamma[P/X] \vdash \forall x\, (A(x)[P/X] \to B[P/X])$. Hence,

$\Gamma[P/X] \vdash B[P/X]$ by $\exists^-$. Note that the variable condition is satisfied because x is a bound variable and we can rename it if needed.

IND. Assume $\Gamma \vdash (\mu(\Phi) \subseteq Q)$ derived from $\Gamma \vdash \Phi(Q) \subseteq Q$. By i.h. $\Gamma[P/X] \vdash (\Phi(Q) \subseteq Q)[P/X]$, which can be rewritten as $\Gamma[P/X] \vdash (\Phi(Q))[P/X] \subseteq Q[P/X]$ (by distributivity). By IND, we get $\Gamma[P/X] \vdash \mu(\Phi)[P/X] \subseteq Q[P/X]$. By distributivity, this corresponds to our goal $\Gamma[P/X] \vdash (\mu(\Phi) \subseteq Q)[P/X]$.

Proofs for strong and half-strong induction are done in the same way and the proof of coinduction is completely dual.

The IFP' versions of different kinds of induction and coinduction are done similarly, additionally taking into account the monotonicity of the operator.

Proofs of all the above statements in a version with $[P/\hat{X}]$ are done in exactly the same way.

□

Lemma 4 (a) If RIFP proves $a\,\mathbf{r}A$ from assumptions that do not contain the predicate variable X and if P is a non-Harrop predicate of the same arity as X, then RIFP proves $a\,\mathbf{r}\,(A[P/X])$ from the same assumptions.

(b) If RIFP proves $a\,\mathbf{r}\,(A[\hat{X}/X])$ from assumptions that do not contain the predicate constant $\hat{X}$ and if P is a Harrop predicate of the same arity as X, then RIFP proves $a\,\mathbf{r}\,(A[P/X])$ from the same assumptions.

Proof. From $a\,\mathbf{r}A$ using Lemma 3 we obtain $(a\,\mathbf{r}A)[\mathbf{R}(P)/\tilde{X}]$ and using Lemma 2 (a), in case P is non-Harrop, we can rewrite it as $a\,\mathbf{r}\,(A[P/X])$. Similarly, we use Lemma 3 to get $(a\,\mathbf{r}\,(A[\hat{X}/X]))[\mathbf{H}(P)/\hat{X}]$ from $a\,\mathbf{r}\,(A[\hat{X}/X])$ and if P is Harrop by Lemma 2 (b) we get $a\,\mathbf{r}\,(A[P/X])$. □

Lemma 5 For any programs p,q and program variables a, and for any terms s,t and object variables x,y:

(a) $\mathbf{R}(E)[t/x] = \mathbf{R}(E[t/x])$ for all expressions E

(b) $(\mathbf{R}(A)(p))[t/x, q/a] = \mathbf{R}(A[t/x])(p[q/a])$ for all formulas A

(c) $(\mathbf{R}(P)(s,p))[t/x, q/a] = \mathbf{R}(P[t/x])(s[t/x], p[q/a])$ for all predicates P

Proof. Proof: Simultaneous structural induction on A and P.

We begin with the proof of (a):

Case $E = P(s)$, P not an abstraction (but note that $\mathbf{R}(P)$ may or may not be an abstraction).

To prove $\mathbf{R}(E)[t/x] = \mathbf{R}(E[t/x])$, we begin rewriting the left-hand side:

$$\begin{array}{lll} & \mathbf{R}(P(s))[t/x] & \\ = & (\lambda b\ \mathbf{R}(P)(s,b))[t/x] & b \text{ is fresh} \\ = & (\lambda b\ (\mathbf{R}(P)(s,b))[t/x]) & \text{since } b \text{ is fresh} \\ = & \lambda b\ (\mathbf{R}(P[t/x])(s[t/x],b)) & \text{by i.h. (c)} \end{array}$$

On the right-hand side:

$$\begin{array}{lll} & \mathbf{R}((P(s))[t/x]) & \\ = & \mathbf{R}(P[t/x](s[t/x])) & \text{Lemma } (P(s))[t/x] = P[t/x](s[t/x]) \\ = & \lambda b\ (\mathbf{R}(P[t/x])(s[t/x],b)) & b \text{ is fresh} \end{array}$$

Case $E = A\Box B$, where $\Box$ stands for $\wedge, \vee, \rightarrow$.
We need to show $\mathbf{R}(A\Box B)[t/x] = \mathbf{R}(A\Box B[t/x])$. Since $\mathbf{R}(A\Box B)[t/x]$ is equal to $\lambda b\ (\mathbf{R}(A\Box B)(b))[t/x]$, $\mathbf{R}(A\Box B[t/x])$ is equal to $\lambda b\ (\mathbf{R}(A\Box B[t/x])(b))$ and b is an arbitrary program variable it suffices to prove these in the (b) version. The same applies to the case $E = \diamond y\ A$, where $\diamond \in \{\forall, \exists\}$.

Now we prove (b):

Case $A = P(s)$, P not an abstraction.
To prove $(\mathbf{R}(A)(p))[t/x, q/a] = \mathbf{R}(A[t/x])(p[q/a])$ we start with the left-hand side:

$$\begin{array}{lll} & (\mathbf{R}(P(s))(p))[t/x,q/a] & \\ = & ((\lambda b\ \mathbf{R}(P)(s,b))(p))[t/x,q/a] & b \text{ is fresh} \\ = & (\mathbf{R}(P)(s,p))[t/x,q/a] & \text{since } b \text{ is fresh} \\ = & \mathbf{R}(P[t/x])(s[t/x],p[q/a]) & \text{by i.h. (c)} \end{array}$$

On the right-hand side:

$$\begin{array}{lll} & \mathbf{R}((P(s))[t/x]))(p[q/a]) & \\ = & \mathbf{R}(P[t/x](s[t/x])))(p[q/a]) & \text{Lemma } (P(s))[t/x] = P[t/x](s[t/x]) \\ = & (\lambda b\ \mathbf{R}(P[t/x])(s[t/x],b))(p[q/a]) & b \text{ is fresh} \\ = & \mathbf{R}(P[t/x])(s[t/x],p[q/a]) & \text{since } b \text{ is fresh} \end{array}$$

Case $A \wedge B$, where A and B are non-Harrop.
To prove $(\mathbf{R}(A \wedge B)(p))[t/x, q/a] = \mathbf{R}((A \wedge B)[t/x])(p[q/a])$ we start with the left-hand side:

$$\begin{array}{lll} & (\mathbf{R}(A \wedge B)(p))[t/x,q/a] & \\ = & (\mathbf{R}(A)(\pi_{\mathbf{Lt}}(p)) \wedge \mathbf{R}(B)(\pi_{\mathbf{Rt}}(p)))[t/x,q/a] & \text{def. realiz.} \\ = & (\mathbf{R}(A)(\pi_{\mathbf{Lt}}(p)))[t/x,q/a] \wedge (\mathbf{R}(B)(\pi_{\mathbf{Rt}}(p)))[t/x,q/a] & \text{distrib.} \\ = & \mathbf{R}(A[t/x])(\pi_{\mathbf{Lt}}(p[q/a])) \wedge \mathbf{R}(B[t/x])(\pi_{\mathbf{Rt}}(p[q/a])) & \text{by i.h. (b)} \end{array}$$

On the right-hand side:

$$\begin{aligned}
& \mathbf{R}(A \wedge B)[t/x]))(p[q/a]) \\
= \; & \mathbf{R}(A[t/x] \wedge B[t/x])(p[q/a]) \\
= \; & \mathbf{R}(A[t/x])(\pi_{\mathbf{Lt}}(p[q/a])) \wedge \mathbf{R}(B[t/x])(\pi_{\mathbf{Rt}}(p[q/a]))
\end{aligned}$$

Case $A \wedge B$, where A is non-Harrop and B is Harrop.
To prove $(\mathbf{R}(A \wedge B)(p))[t/x, q/a] = \mathbf{R}((A \wedge B)[t/x])(p[q/a])$ we start with the left-hand side:

$$\begin{aligned}
& (\mathbf{R}(A \wedge B)(p))[t/x, q/a] & \\
= \; & (\mathbf{R}(A)(p) \wedge \mathbf{H}(B))[t/x, q/a] & \text{def. realiz.} \\
= \; & \mathbf{R}(A[t/x])(p[q/a]) \wedge \mathbf{H}(B[t/x]) & \text{by i.h. (b)}
\end{aligned}$$

On the right-hand side:

$$\begin{aligned}
& \mathbf{R}(A \wedge B)[t/x]))(p[q/a]) \\
= \; & \mathbf{R}(A[t/x] \wedge B[t/x])(p[q/a]) \\
= \; & \mathbf{R}(A[t/x])(p[q/a]) \wedge \mathbf{H}(B[t/x])
\end{aligned}$$

Case $A \vee B$, where A and B are non-Harrop.
To prove $(\mathbf{R}(A \vee B)(p))[t/x, q/a] = \mathbf{R}((A \vee B)[t/x])(p[q/a])$ we start with the left-hand side:

$$\begin{aligned}
& (\mathbf{R}(A \vee B)(p))[t/x, q/a] & \\
= \; & (\exists b(p = \mathbf{Lt}(b) \wedge \mathbf{R}(A)(b)) \vee \exists c(p = \mathbf{Rt}(c) \wedge \mathbf{R}(B)(c)))[t/x, q/a] & \text{def. realiz.} \\
= \; & (\exists b(p = \mathbf{Lt}(b) \wedge \mathbf{R}(A)(b)))[t/x, q/a] \vee (\exists c(p = \mathbf{Rt}(c) \wedge \mathbf{R}(B)(c)))[t/x, q/a] & \text{distrib.} \\
= \; & \exists b((p[q/a]) = \mathbf{Lt}(b) \wedge \mathbf{R}(A[t/x])(b)) \vee \exists c((p[q/a]) = \mathbf{Rt}(c) \wedge \mathbf{R}(B[t/x])(c)) & \text{by i.h. (b)} \\
& & b,\, c \text{ arbitrary}
\end{aligned}$$

On the right-hand side:

$$\begin{aligned}
& \mathbf{R}(A \vee B)[t/x]))(p[q/a]) & \\
= \; & \mathbf{R}(A[t/x] \vee B[t/x])(p[q/a]) & \text{distrib.} \\
= \; & \exists b((p[q/a]) = \mathbf{Lt}(b) \wedge \mathbf{R}(A[t/x])(b) \vee \exists c((p[q/a]) = \mathbf{Rt}(c) \wedge \mathbf{R}(B[t/x]))(c) & \text{def. realiz.}
\end{aligned}$$

Case $A \to B$, where A and B are non-Harrop.
To prove $(\mathbf{R}(A \to B)(p))[t/x, q/a] = \mathbf{R}((A \to B)[t/x])(p[q/a])$ we start with the left-hand side:

$$\begin{aligned}
& (\mathbf{R}(A \to B)(p))[t/x, q/a] & \\
= \; & (\forall b(\mathbf{R}(A)(b) \to \mathbf{R}(B)(p\ b)))[t/x, q/a] & \text{def. realiz.} \\
= \; & \forall b'(\mathbf{R}(A)(b) \to \mathbf{R}(B)(p\ b))[t/x, q/a, b'/b] & \\
= \; & \forall b'(\mathbf{R}(A[t/x])(b') \to \mathbf{R}(B[t/x])((p[q/a])\ b')) & \text{subst.}
\end{aligned}$$

On the right-hand side:

$$\begin{aligned}
&\mathbf{R}((A \to B)[t/x])(p[q/a]) & \\
=\ & \mathbf{R}(A[t/x] \vee B[t/x])(p[q/a]) & \text{distrib.}\\
=\ & \forall b' (\mathbf{R}(A[t/x])(b') \to \mathbf{R}(B[t/x])((p[q/a])\ b')) & \text{def. realiz.}
\end{aligned}$$

Case $\diamond y\ A$, where $\diamond \in \forall, \exists$ A is non-Harrop.
To prove $(\mathbf{R}(\diamond y\ A)(p))[t/x, q/a] = \mathbf{R}((\diamond y\ A)[t/x])(p[q/a])$ we start with the left-hand side:

$$\begin{aligned}
&(\mathbf{R}(\diamond y\ A)(p))[t/x, q/a] & \\
=\ & \diamond y'\ \mathbf{R}(A[y'/y])(p)[t/x, q/a] & \text{def. realiz.}\\
=\ & \diamond y'\ \mathbf{R}(A[y'/y, t/x])(p[q/a]) & \text{by i.h. (b)}
\end{aligned}$$

On the right-hand side:

$$\begin{aligned}
&\mathbf{R}((\diamond y\ A)[t/x])(p[q/a]) & \\
=\ & \mathbf{R}(\diamond y'\ A[y'/y, t/x])(p[q/a]) & \text{subst.}\\
=\ & \diamond y'\ \mathbf{R}(A[y'/y, t/x])(p[q/a]) & \text{def. realiz.}
\end{aligned}$$

Now we prove (c):
Case P is a constant.
To prove $(\mathbf{R}(P)(s,p))[t/x, q/a] = \mathbf{R}(P[t/x])(s[t/x], p[q/a])$ we begin with the left-hand side:

$$\begin{aligned}
&(\mathbf{R}(P)(s,p))[t/x, q/a] \\
=\ & (P(s,p))[t/x, q/a] \\
=\ & P[t/x](s[t/x], p[q/a])
\end{aligned}$$

Right-hand side: $\mathbf{R}(P[t/x])(s[t/x], p[q/a]) = P[t/x](s[t/x], p[q/a])$ by the definition of realizers.

Case P is a variable X.
To prove $(\mathbf{R}(P)(s,p))[t/x, q/a] = \mathbf{R}(P[t/x])(s[t/x], p[q/a])$ we begin with the left-hand side:

$$\begin{aligned}
&(\mathbf{R}(X)(s,p))[t/x, q/a] \\
=\ & (\tilde{X}(s,p))[t/x, q/a] \\
=\ & \tilde{X}[t/x](s[t/x], p[q/a])
\end{aligned}$$

Right-hand side: $\mathbf{R}(X[t/x])(s[t/x], p[q/a]) = \tilde{X}[t/x](s[t/x], p[q/a])$ by the definition of realizers.

Case $P = \lambda y\, A$, where w.l.o.g y not free in t.
To prove $(\mathbf{R}(P)(s,p))[t/x,q/a] = \mathbf{R}(P[t/x])(s[t/x],p[q/a])$ we begin with the left-hand side:

$$
\begin{array}{lll}
 & (\mathbf{R}(\lambda y\, A)(s,p))[t/x,q/a] & \\
= & ((\lambda(y,b)\ \mathbf{R}(A)(b))(s,p))[t/x,q/a] & b \text{ fresh} \\
= & (\mathbf{R}(A)[s/y](p))[t/x,q/a] & \text{since } b \text{ is fresh} \\
= & (\mathbf{R}(A[s/y])(p))[t/x,q/a] & \text{by i.h (a)} \\
= & \mathbf{R}((A[s/y])[t/x])(p[q/a]) & \text{by i.h. (b)} \\
= & \mathbf{R}(A[t/x,s[t/x]/y])(p[q/a]) & \text{Lemma } (A[s/y])[t/x] = A[t/x,s[t/x]/y]
\end{array}
$$

Right-hand side:

$$
\begin{array}{ll}
(\mathbf{R}(\lambda y\, A[t/x])(s[t/x],p[q/a])) & (y \text{ not free in } t) \\
=((\lambda(y,b)\ \mathbf{R}(A[t/x])(b))(s[t/x],p[q/a])) & b \text{ fresh} \\
=\mathbf{R}(A[t/x])(b)[s[t/x]/y,p[q/a]/b] & \\
=\mathbf{R}(A[t/x][s[t/x]/y])([p[q/a]/b]) & \text{by i.h. b} \\
=(A[t/x,s[t/x]/y])(p[q/a]) & \text{since } b \text{ is fresh}
\end{array}
$$

Case $P = \diamond(\lambda X\ Q)$, where $\diamond$ stands for μ or ν; P non-Harrop, i.e. $Q[\hat{X}/X]$ non-Harrop. To prove $(\mathbf{R}(P)(s,p))[t/x,q/a] = \mathbf{R}(P[t/x])(s[t/x],p[q/a])$ we proceed on the left-hand side:

$$
\begin{array}{lll}
 & (\mathbf{R}(\diamond(\lambda X Q))(s,p))[t/x,q/a] & \\
= & (\diamond(\lambda\tilde{X}\ \mathbf{R}(Q)))(s,p))[t/x,q/a] & \\
= & (\diamond(\lambda\tilde{X}(\mathbf{R}(Q)[t/x])))(s[t/x],p[q/a]) & \text{since } \diamond(\lambda\tilde{X}\ \mathbf{R}(Q)) \text{ is not an abstraction} \\
= & (\diamond(\lambda\tilde{X}\ \mathbf{R}(Q[t/x])))(s[t/x],p[q/a]) & \text{by i.h. (a)}
\end{array}
$$

On the right-hand side:

$$
\begin{array}{lll}
 & \mathbf{R}((\diamond(\lambda X\ Q))[t/x])(s[t/x],p[q/a]) & \\
= & (\diamond(\lambda\tilde{X}\ \mathbf{R}(Q)))[t/x])(s[t/x],p[q/a]) & \\
= & \diamond(\lambda\tilde{X}\ \mathbf{R}(Q)[t/x])(s[t/x],p[q/a]) & \\
= & \diamond(\lambda\tilde{X}\ \mathbf{R}(Q[t/x]))(s[t/x],p[q/a]) & \text{by i.h. (a)}
\end{array}
$$

Case $P = \diamond(\lambda X\ Q)$, where $\diamond$ stands for μ or ν; P Harrop, i.e. $Q[\hat{X}/X]$ Harrop. To prove $(\mathbf{R}(P)(s,p))[t/x,q/a] = \mathbf{R}(P[t/x])(s[t/x],p[q/a])$ we rewrite the left-hand

side first:

$$
\begin{aligned}
& (\mathbf{H}(\diamond(\lambda X\ Q))(s) \wedge p = Nil)[t/x, q/a] \\
= \ & (\diamond(\lambda X\ \mathbf{H}_X(Q))(s) \wedge p = Nil)[t/x, q/a] \\
= \ & \diamond(\lambda X(\mathbf{H}_X(Q)[t/x]))(s[t/x]) \wedge p[q/a] = \mathbf{Nil} \\
= \ & \diamond(\lambda X(\mathbf{H}_X(Q[t/x])))(s[t/x]) \wedge p[q/a] = \mathbf{Nil} && \text{by (a)}
\end{aligned}
$$

The right-hand side:

$$
\begin{aligned}
& \mathbf{R}((\diamond(\lambda X\ Q))[t/x])(s[t/x], p[q/a]) \\
= \ & \mathbf{R}(\diamond(\lambda X\ (Q[t/x])))(s[t/x], p[q/a]) && \text{since } \lambda X\ (Q[t/x]) \text{ is still Harrop} \\
= \ & \diamond(\lambda X\ \mathbf{H}_X(Q[t/x])))(s[t/x]) \wedge p[q/a] = \mathbf{Nil}
\end{aligned}
$$

□

The correlation between the IFP and RIFP with respect to free object, predicate and program variables, is shown in the following lemma.

Lemma 6 For each IFP-expression E:

(a) $FObV(\mathbf{R}(E)) = FObV(E)$

(b) $FPredV(\mathbf{R}(E)) = \{\tilde{X} | X \in FPredV(E)\}$

(c) $FProgV(\mathbf{R}(E)) = \emptyset$

Note that if E is a formula, then for all programs p:

(a*) $FObV(\mathbf{R}(E)(p)) = FObV(\mathbf{R}(E)) = FObV(E)$

(b*) $FPredV(\mathbf{R}(E)(p)) = FPredV(\mathbf{R}(E)) = \{\tilde{X} | X \in FPredV(E)\}$

Proof. The proof is straightforward by induction on E. Here we include several chosen cases.

For (a):

Case $E = X$, where X is a predicate variable. Then $\mathbf{R}(X) = \tilde{X}$ is again a predicate variable. But $FObV(\tilde{X}) = \emptyset = FObV(X)$.

Case $E = \lambda x\, A$. We show $FObV(\mathbf{R}(\lambda x\, A)) = FObV(\lambda x\, A)$. On the left hand side:

$$
\begin{aligned}
& FObV(\mathbf{R}(\lambda x\, A)) \\
= \ & FObV(\lambda(x,a)\ \mathbf{R}(A)(a)) && \text{def. realiz.} \\
= \ & FObV(\mathbf{R}(A)(a)) \setminus \{x\} && a \text{ is prog. var.} \\
= \ & FObV(A) \setminus \{x\} && \text{by i.h.}
\end{aligned}
$$

On the right-hand side we have $FObV(\lambda x\ A)$, which by definition of free variables is also equal to $FObV(A)\setminus\{x\}$.

For case (a*) we consider $\mathbf{R}(E)$ applied to a program (s,p) where s is a vector of object terms of suitable length.

Case $E=A\wedge B$. We need to show that $FObV(\mathbf{R}(A\wedge B)(p))=FObV(A\wedge B)$.

$$
\begin{array}{ccl}
 & FObV(\mathbf{R}(A\wedge B)(p)) & \\
 & FObV(\mathbf{R}(A)(\pi_{\mathbf{Lt}}(p))\wedge\mathbf{R}(B)(\pi_{\mathbf{Rt}}(p))) & \text{def. realiz.}\\
= & FObV(\mathbf{R}(A)(\pi_{\mathbf{Lt}}(p)))\cup FObV(\mathbf{R}(B)(\pi_{\mathbf{Rt}}(p))) & \text{def. free var.}\\
= & FObV(A)\cup FObV(B) & \text{by i.h.}
\end{array}
$$

Based on the definition of free variables, $FObV(A\wedge B)$ corresponds to the latter.

For (b):

Case $E=X$, where X is a predicate variable. $FPredV(\mathbf{R}(E))=FPredV(\mathbf{R}(X))=\{\tilde{X}\}$. But also $FPredV(X)=\{\tilde{X}\}$.

Case $E=\diamond\ (\lambda Y\ Q)$, where $\diamond$ stands for μ or ν; E non-Harrop, i.e. $Q[\hat{X}/X]$ non-Harrop. We show $FPredV(\mathbf{R}(\diamond\ (\lambda Y\ Q)))=\{\tilde{X}\mid X\in FPredV(\diamond\ (\lambda Y\ Q))\}$. On the left hand side:

$$
\begin{array}{crl}
 & FPredV(\mathbf{R}(\diamond\ (\lambda Y\ Q))) & \\
= & FPredV(\diamond\ \mathbf{R}(\lambda Y\ Q)) & \text{def. realiz.}\\
= & FPredV(\mathbf{R}(\lambda Y\ Q)) & \text{def. free var.}\\
= & FPredV(\lambda\tilde{Y}\ \mathbf{R}(Q)) & \text{def. realiz.}\\
= & FPredV(\mathbf{R}(Q))\setminus\{\tilde{Y}\} & \\
= & \{\tilde{X}\mid X\in FPredV(Q)\}\setminus\{\tilde{Y}\} & \text{by i.h.}
\end{array}
$$

$\{\tilde{X}\mid X\in FPredV(\lambda Y\ Q)\}=\{\tilde{X}\mid X\in(FPredV(Q)\setminus\{Y\})\}$ by the definition of free variables. Since $\{\tilde{Y}|Y\in\{Y\}\}$, then $\{\tilde{X}\mid X\in FPredV(Q)\}\ \setminus\{\tilde{Y}\}=\{\tilde{X}\mid X\in(FPredV(Q)\setminus\{Y\})\}$.

For case (b*) we consider $\mathbf{R}(E)$ applied to a program (s,p) where s is a vector of object terms of suitable length.

Case $E = A \wedge B$. We show $FPredV(\mathbf{R}(A \wedge B)(p)) = \{\tilde{X} \mid X \in FPredV(A \wedge B)\}$.

$$
\begin{array}{lll}
 & FPredV(\mathbf{R}(A \wedge B)(p)) & \\
 & FPredV(\mathbf{R}(A)(\pi_{\mathbf{Lt}}(p)) \wedge \mathbf{R}(B)(\pi_{\mathbf{Rt}}(p))) & \text{def. realiz.} \\
= & FPredV(\mathbf{R}(A)(\pi_{\mathbf{Lt}}(p))) \cup FPredV(\mathbf{R}(B)(\pi_{\mathbf{Rt}}(p))) & \text{def. free var.} \\
= & \{\tilde{X} \mid X \in FPredV(A)\} \cup \{\tilde{X} \mid X \in FPredV(B)\} & \text{by i.h.}
\end{array}
$$

Based on the definition of free variables, $\{\tilde{X} \mid X \in FPredV(A \wedge B)\} = \{\tilde{X} \mid X \in FPredV(A) \cup FPredV(B)\}$, which can also be represented as $\{\tilde{X} \mid X \in FPredV(A)\} \cup \{\tilde{X} \mid X \in FPredV(B)\}$.

For (c) the proof is straightforward since $\mathbf{R}(E) = \lambda a\, \mathbf{R}(E)(a)$ for all E, so the only program variable is bound. Hence, we conclude that $FProgV(E) = \emptyset$. $\square$

1. $(f^{-1} \circ Q)(\vec{x}, a) \overset{\text{Def}}{=} Q(\vec{x}, f\, a)$
2. $(f \circ Q)(\vec{x}, b) \overset{\text{Def}}{=} \exists a\, (f\, a = b \wedge Q(\vec{x}, a))$
3. $(b^{-1} * Q)(\vec{x}) \overset{\text{Def}}{=} Q(\vec{x}, b)$
4. $(a * P)(\vec{x}, b) \overset{\text{Def}}{=} a = b \wedge P(\vec{x})$
5. $\Delta(P)(\vec{x}, b) \overset{\text{Def}}{=} P(\vec{x})$
6. $\exists(Q)(\vec{x}) \overset{\text{Def}}{=} \exists a\, Q(\vec{x}, a)$
7. $f \circ g \overset{\text{Def}}{=} \lambda a(f(g\ a))$

Figure 10.1: Monotone predicate transformers

We define a number of monotone predicate transformers and prove equivalences and statements, which are required to proceed with the proofs for induction and coinduction as shown in fig. 10.1. Q is a predicate of arity $(\vec{\iota}, \delta)$. P is a predicate of arity $(\vec{\iota})$. $f, a, b : \delta$ are domain elements; $(f\ a)$ denotes application in D.

The following lemmas present general equivalences, adjunctions as well as realizability equivalences.

Lemma 7 *Equivalences.*

(a) $f^{-1} \circ (g^{-1} \circ Q) \equiv (g \circ f)^{-1} \circ Q$

(b) $f \circ (g \circ Q) \equiv (f \circ g) \circ Q$

(c) $a^{-1} * (f^{-1} \circ Q) \equiv (f\,a)^{-1} * Q$

(d) $f \circ (a * P) \equiv (f\,a) * P$

(e) $f^{-1} \circ \Delta P \equiv \Delta P$

(f) $\exists(f \circ Q) \equiv \exists(Q)$

(g) $f^{-1} \circ P \cap g^{-1} \circ Q \equiv \langle f, g\rangle^{-1} \circ (\pi_{\mathbf{Lt}}{}^{-1} \circ P \cap \pi_{\mathbf{Rt}}{}^{-1} \circ Q)$

(h) $f \circ P \cup g \circ Q \equiv [f + g] \circ (\mathbf{Lt} \circ P \cup \mathbf{Rt} \circ Q)$

(i) $(b^{-1} * Q) \cap P \equiv b^{-1} * (Q \cap \Delta P)$

Adjunctions.

(j) $Q \subseteq f^{-1} \circ Q' \leftrightarrow f \circ Q \subseteq Q'$

(k) $P \subseteq b^{-1} * Q \leftrightarrow b * P \subseteq Q$

(l) $Q \subseteq \Delta P \leftrightarrow \exists(Q) \subseteq P$

Proof. In this proof we use definitions from fig. 10.1. We begin with the equivalences. In order to prove them, we need to apply the same arguments to both sides.

To prove the equivalence (a) $f^{-1} \circ (g^{-1} \circ Q) \equiv (g \circ f)^{-1} \circ Q$ we apply $(\vec{x}, a)$ to both sides. We start with the left-hand side:

$$
\begin{array}{lrl}
 & (f^{-1} \circ (g^{-1} \circ Q))(\vec{x}, a) & \\
\equiv & (g^{-1} \circ Q)(\vec{x}, f\ a) & \text{def. 1} \\
\equiv & Q(\vec{x}, g(f\ a)) & \text{def. 1 and 7}
\end{array}
$$

Now we proceed on the right-hand side:

$$
\begin{array}{lrl}
 & ((g \circ f)^{-1} \circ Q)(\vec{x}, a) & \\
\equiv & Q(\vec{x}, (g \circ f)\ a) & \text{def. 1} \\
\equiv & Q(\vec{x}, g(f\ a)) & \text{def. 7}
\end{array}
$$

Hence, this equivalence statement is valid.

To prove the equivalence (b) $f \circ (g \circ Q) \equiv (f \circ g) \circ Q$ we apply $(\vec{x}, b)$ to both sides. First, we work on the left-hand side:

$$
\begin{array}{lrl}
 & (f \circ (g \circ Q))(\vec{x}, b) & \\
\equiv & \exists a'(f\ a' = b \wedge (g \circ Q)(\vec{x}, a')) & \text{def. 2} \\
\equiv & \exists a'(f\ a' = b \wedge \exists a(g\ a = a' \wedge Q(\vec{x}, a))) & \text{def. 2} \\
\equiv & \exists a(f\ (g\ a) = b \wedge Q(\vec{x}, a)) & a' = g\ a
\end{array}
$$

Now, we proceed on the right-hand side:

$$
\begin{array}{lrl}
 & ((f \circ g) \circ Q)(\vec{x}, b) & \\
\equiv & \exists a(((g \circ f)\ a) = b \wedge Q(\vec{x}, a)) & \text{def. 2} \\
\equiv & \exists a(f\ (g\ a) = b \wedge Q(\vec{x}, a)) & \text{def. 7}
\end{array}
$$

Hence, both sides can be transformed into the same form, making this equivalence statement valid.

To prove the equivalence (c) $a^{-1} * (f^{-1} \circ Q) \equiv (f\,a)^{-1} * Q$ we apply $(\vec{x})$ to both sides. We proceed on the left-hand side first:

$$
\begin{array}{lrl}
 & (a^{-1} * (f^{-1} \circ Q))(\vec{x}) & \\
\equiv & (f^{-1} \circ Q)(\vec{x}, a) & \text{def. 3} \\
\equiv & Q(\vec{x}, (f\ a)) & \text{def. 1}
\end{array}
$$

Now, on the right-hand side $((f\,a)^{-1} * Q)(\vec{x})$ is equivalent to $Q(\vec{x}, (f\,a))$ def. 3. Hence, the equivalence statement (c) is valid.

To prove the equivalence (d) $f \circ (a * P) \equiv (f\,a) * P$ we apply $(\vec{x}, b)$ to both sides. The left-hand side:

$$
\begin{array}{llr}
 & (f \circ (a * P))(\vec{x}, b) & \\
\equiv & \exists c(f\ c = b \wedge (a * P)(\vec{x}, c)) & \text{def. 2} \\
\equiv & \exists c(f\ c = b \wedge (a = c \wedge P(\vec{x})) & \text{def. 4} \\
\equiv & f\ a = b \wedge P(\vec{x}) & c = a
\end{array}
$$

Now, we proceed on the right-hand side. def. 4, $((f\,a) * P)(\vec{x}, b)$ is equivalent to $f\,a = b \wedge P(\vec{x}))$. Hence, this equivalence statement is valid.

To prove the equivalence (e) $f^{-1} \circ \Delta P \equiv \Delta P$ we apply $(\vec{x}, b)$ to both sides. We proceed on the right-hand side:

$$
\begin{array}{llr}
 & f^{-1} \circ \Delta(P)(\vec{x}, b) & \\
\equiv & \Delta(P)(\vec{x}, f\ b) & \text{def. 1} \\
\equiv & (P)(\vec{x}) & \text{def. 5}
\end{array}
$$

On the left-hand side, def. 5, $\Delta(P)(\vec{x}, b)$ is equivalent to $P(\vec{x})$. Hence, the equivalence statement (e) is valid.

To prove the equivalence (f) $\exists(f \circ Q) \equiv \exists(Q)$ we apply $(\vec{x})$ to both sides. On the left-hand side:

$$
\begin{array}{llr}
 & \exists(f \circ Q)(\vec{x}) & \\
\equiv & \exists b(f \circ Q)(\vec{x}, b) & \text{def. 6} \\
\equiv & \exists b(\exists a(f\ a = b \wedge Q(\vec{x}, a)) & \text{def. 2} \\
\equiv & \exists a(Q(\vec{x}, a)) & b = f\ a
\end{array}
$$

On the right-hand side $\exists(Q)(\vec{x}) \equiv \exists a(Q)(\vec{x}, a)$ def. 6. Hence, the equivalence statement (f) is valid.

To prove the equivalence (g) $f^{-1} \circ P \cap g^{-1} \circ Q \equiv \langle f, g\rangle^{-1} \circ ({\pi_{\mathbf{Lt}}}^{-1} \circ P \cap {\pi_{\mathbf{Rt}}}^{-1} \circ Q)$ we apply $(\vec{x}, a)$ to both sides. def. 1 the left-hand side is equivalent to

$P(\vec{x}, f\ a)\ \wedge\ Q(\vec{x}, g\ a)$. On the right hand-side we have the following:

$$\begin{array}{lrl}
 & (\langle f,g\rangle^{-1} \circ (\pi_{\mathbf{Lt}}{}^{-1} \circ P\ \cap\ \pi_{\mathbf{Rt}}{}^{-1} \circ Q))(\vec{x},a) & \\
\equiv & (\pi_{\mathbf{Lt}}{}^{-1} \circ P\ \cap\ \pi_{\mathbf{Rt}}{}^{-1} \circ Q)(\vec{x}, \langle f,g\rangle\ a) & \text{def. 1} \\
\equiv & (\pi_{\mathbf{Lt}}{}^{-1} \circ P)(\vec{x}, \langle f,g\rangle\ a) \wedge\ (\pi_{\mathbf{Rt}}{}^{-1} \circ Q)(\vec{x}, \langle f,g\rangle\ a) & \text{def. intersection} \\
\equiv & P(\vec{x}, \pi_{\mathbf{Lt}}(\langle f,g\rangle\ a))\ \wedge\ Q(\vec{x}, \pi_{\mathbf{Rt}}(\langle f,g\rangle\ a)) & \text{def. 1} \\
\equiv & P(\vec{x}, f\ a)\ \wedge\ Q(\vec{x}, g\ a) & \text{def. of proj. and def. } \langle f,g\rangle
\end{array}$$

Hence, the equivalence statement (g) is valid.

To prove the equivalence (h) $f \circ P\ \cup\ g \circ Q \equiv [f+g] \circ (\mathbf{Lt} \circ P\ \cup\ \mathbf{Rt} \circ Q)$ we apply $(\vec{x}, b)$ to both sides. We begin with the left-hand side:

$$\begin{array}{lrl}
 & (f \circ P\ \cup\ g \circ Q)(\vec{x}, b) & \\
\equiv & \exists a(f\ a = b \wedge P(\vec{x}, a))\ \vee\ \exists a(g\ a = b \wedge Q(\vec{x}, a)) & \text{def. 2}
\end{array}$$

Now, we proceed on the right-hand side:

$$\begin{array}{lrl}
 & ([f+g] \circ (\mathbf{Lt} \circ P\ \cup\ \mathbf{Rt} \circ Q))(\vec{x}, b) & \\
\equiv & \exists a([f+g]\ a = b \wedge (\mathbf{Lt} \circ P\ \cup\ \mathbf{Rt} \circ Q)(\vec{x}, a)) & \text{def. 2} \\
\equiv & \exists a([f+g]\ a = b \wedge (\exists a'(\mathbf{Lt}\ a' = a \wedge P(\vec{x}, a'))\ \vee\ \exists b'(\mathbf{Rt}\ b' = a \wedge Q(\vec{x}, b')))) & \text{def. 2}
\end{array}$$

By distributivity, the last statement can be rewritten as follows:

$$\exists a([f+g]\ a = b \wedge \exists a'(\mathbf{Lt}\ a' = a \wedge P(\vec{x}, a')))\ \vee\ \exists a([f+g]\ a = b \wedge \exists b'(\mathbf{Rt}\ b' = a \wedge Q(\vec{x}, b'))))$$

The left side of this disjunction, can be rewritten as $\exists a([f+g](\mathbf{Lt}\ a') = b \wedge P(\vec{x}, a'))$, which is equivalent to $\exists a((f\ a') = b \wedge P(\vec{x}, a'))$. Similarly, we get $\exists b'((g\ b') = b \wedge Q(\vec{x}, b'))$ on the right side of the disjunction. Now if we substitute a' and b' by a then both sides of we see that this equivalence is valid.

To prove the equivalence (i) $(b^{-1} * Q)\ \cap\ P \equiv b^{-1} * (Q\ \cap\ \Delta P)$ we apply $(\vec{x})$ to the left-hand side:

$$\begin{array}{lrl}
 & (b^{-1} * Q)(\vec{x}) \wedge P(\vec{x}) & \\
\equiv & Q(\vec{x}, b) \wedge P(\vec{x}) & \text{def. 3 and 5}
\end{array}$$

Now, on the right-hand side we also apply $(\vec{x})$:

$$\begin{array}{lrl}
 & (b^{-1} * (Q \cap \Delta P))(\vec{x}) & \\
\equiv & (Q \cap \Delta P)(\vec{x}, b) & \text{def. 3} \\
\equiv & Q(\vec{x}, b) \wedge P(\vec{x}) & \text{def. 3 and 5}
\end{array}$$

Therefore, both sides are equivalent.

We now procees with the proof of the adjunctions.

For the adjunction (j) $Q \subseteq f^{-1} \circ Q' \leftrightarrow f \circ Q \subseteq Q'$ we look at the left-hand side inclusion first and apply $(\vec{x}, a)$ to it:

$$\begin{aligned} & \forall \vec{x}, a(Q(\vec{x}, a) \rightarrow (f^{-1} \circ Q')(\vec{x}, a)) & \\ \equiv \quad & \forall \vec{x}, a(Q(\vec{x}, a) \rightarrow Q'(\vec{x}, f\ a)) & \text{def. 1} \end{aligned}$$

Now we look at the right-hand side inclusion and apply $(\vec{x}, b)$ to it:

$$\begin{aligned} & \forall \vec{x}, b((f \circ Q)(\vec{x}, b) \rightarrow Q(\vec{x}, b)) & \\ \equiv \quad & \forall \vec{x}, b(\exists a(f\ a = b \wedge Q(\vec{x}, a)) \rightarrow Q(\vec{x}, b)) & \text{def. 2} \\ = \quad & \forall \vec{x}(\exists a(Q(\vec{x}, a)) \rightarrow Q(\vec{x}, f\ a)) & \text{let } b = f\ a \\ \equiv \quad & \forall \vec{x}, a(Q(\vec{x}, a) \rightarrow Q'(\vec{x}, f\ a)) & \text{since } (\exists a\ Q(a) \rightarrow B) \leftrightarrow \forall a\ (Q(a) \rightarrow B) \end{aligned}$$

Hence, both inclusions can be transformed into the same form, making this adjunction valid.

For the adjunction (k) $P \subseteq b^{-1} * Q \leftrightarrow b * P \subseteq Q$ we look at the left-hand side inclusion first and apply $(\vec{x})$ to it:

$$\begin{aligned} & \forall \vec{x}(P(\vec{x}) \rightarrow b^{-1} * Q(\vec{x})) & \\ \equiv \quad & \forall \vec{x}(P(\vec{x}) \rightarrow Q(\vec{x}, b)) & \text{def. 3} \end{aligned}$$

Now we look at the right-hand side inclusion and apply $(\vec{x}, b)$ to it:

$$\begin{aligned} & \forall \vec{x}, b((b * P)(\vec{x}, b) \rightarrow Q(\vec{x}, b)) & \\ \equiv \quad & \forall \vec{x}(P(\vec{x}) \rightarrow Q(\vec{x}, b)) & \text{def. 4; } b = b \end{aligned}$$

Hence, both inclusions can be transformed into the same form, making this adjunction valid.

For the adjunction (l) $Q \subseteq \Delta P \leftrightarrow \exists(Q) \subseteq P$ we look at the left-hand side inclusion first and apply $(\vec{x}, a)$ to it:

$$\begin{aligned} & \forall \vec{x}, a(Q(\vec{x}, a) \rightarrow \Delta(P)(\vec{x}, a)) & \\ \equiv \quad & \forall \vec{x}, a(Q(\vec{x}, a) \rightarrow P(\vec{x})) & \text{def. 5} \end{aligned}$$

Now we look at the right-hand side inclusion and apply $(\vec{x})$ to its both sides:

$$\begin{array}{lll} & \forall\vec{x}(\exists(Q)(\vec{x}) \to P(\vec{x})) & \\ \equiv & \forall\vec{x}(\exists a(Q)(\vec{x},a) \to P(\vec{x})) & \text{def. 4} \\ \equiv & \forall\vec{x},a(Q(\vec{x},a) \to P(\vec{x})) & \text{since } (\exists a\ Q(a) \to B) \leftrightarrow \forall a\ (Q(a) \to B) \end{array}$$

Hence, both inclusions can be transformed into the same form, making this adjunction valid.

□

In the following lemma and the soundness theorem we use the notation $a\,\mathbf{r}\,A$ to denote $\mathbf{R}(A)(a)$ for convenience.

Lemma 8 *Realizability*. Let Q and Q' be non-Harrop predicates, P and P' Harrop predicates, f a function and a a variable:

(a) $f\,\mathbf{r}\,(Q \subseteq Q') \leftrightarrow \mathbf{R}(Q) \subseteq f^{-1} \circ \mathbf{R}(Q') \leftrightarrow f \circ \mathbf{R}(Q) \subseteq \mathbf{R}(Q')$

(b) $a\,\mathbf{r}\,(P \subseteq Q) \leftrightarrow \mathbf{H}(P) \subseteq a^{-1} * \mathbf{R}(Q) \leftrightarrow a * \mathbf{H}(P) \subseteq \mathbf{R}(Q)$

(c) $\mathbf{H}(Q \subseteq P) \leftrightarrow \mathbf{R}(Q) \subseteq \Delta\mathbf{H}(P) \leftrightarrow \exists(\mathbf{R}(Q)) \subseteq \mathbf{H}(P)$

(d) $\mathbf{H}(P \subseteq P') \leftrightarrow \mathbf{H}(P) \subseteq \mathbf{H}(P')$

(e) $\mathbf{R}(Q \ \cap \ Q') \equiv \pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q) \ \cap \ \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(Q')$

(f) $\mathbf{R}(Q \cup Q') \equiv \mathbf{Lt} \circ \mathbf{R}(Q) \cup \mathbf{Rt} \circ \mathbf{R}(Q')$

(g) $\mathbf{R}(Q \ \cap \ P) \equiv \mathbf{R}(Q) \cap \Delta(\mathbf{H}(P))$

(h) $\exists(\mathbf{R}(Q)) \cup \mathbf{H}(P) \equiv \exists(\mathbf{R}(Q \cup P))$

Proof. In this proof we use definitions from fig. 10.1.

For the statement (a) $f\,\mathbf{r}\,(Q \subseteq Q') \leftrightarrow \mathbf{R}(Q) \subseteq f^{-1} \circ \mathbf{R}(Q') \leftrightarrow f \circ \mathbf{R}(Q) \subseteq \mathbf{R}(Q')$ we start with the first part and write out its equivalents:

$$\begin{array}{lll} & f\,\mathbf{r}\,(Q \subseteq Q') & \\ \equiv & f\,\mathbf{r}\,(\forall\vec{x}(Q(\vec{x}) \to Q'(\vec{x}))) & \text{unfolding } \subseteq \\ \equiv & \forall\vec{x}\ f\,\mathbf{r}\,(Q(\vec{x}) \to Q'(\vec{x})) & \text{def. } f\,\mathbf{r}\,\forall\vec{x}\ Q(\vec{x}) \\ \equiv & \forall\vec{x}\ (\forall a\ (a\,\mathbf{r}\,Q(\vec{x}) \to (f\ a)\,\mathbf{r}\,Q'(\vec{x}))) & \text{def. } f\,\mathbf{r}\,(Q \to Q') \\ \equiv & \forall\vec{x},a\ (\mathbf{R}(Q)(\vec{x},a) \to \mathbf{R}(Q')(\vec{x},f\ a)) & \text{def. } \mathbf{R} \end{array}$$

Now we unfold the definition of $\subseteq$ in $\mathbf{R}(Q) \subseteq f^{-1} \circ \mathbf{R}(Q')$ and apply $(\vec{x}, a)$ to it:

$$\begin{array}{lll} & \forall \vec{x}, a\ (\mathbf{R}(Q)(\vec{x}, a) \to f^{-1} \circ \mathbf{R}(Q')(\vec{x}, a)) & \\ \equiv & \forall \vec{x}, a\ (\mathbf{R}(Q)(\vec{x}, a) \to \mathbf{R}(Q')(\vec{x}, f\ a)) & \text{def. } 1 \end{array}$$

Lastly, we transform $f \circ \mathbf{R}(Q) \subseteq \mathbf{R}(Q')$ in a similar way, applying arbitrary $(\vec{x}, b)$ to it to obtain the following equivalent statements:

$$\begin{array}{lll} & \forall \vec{x}, b\ (f \circ \mathbf{R}(Q)(\vec{x}, b) \to \mathbf{R}(Q')(\vec{x}, b)) & \\ \equiv & \forall \vec{x}\ (\exists a\ (\mathbf{R}(Q)(\vec{x}, a)) \to \mathbf{R}(Q')(\vec{x}, f\ a)) & \text{def. } 2,\ b = f\ a \\ \equiv & \forall \vec{x}, a\ (\mathbf{R}(Q)(\vec{x}, a) \to \mathbf{R}(Q')(\vec{x}, f\ a)) & \text{since } (\exists a\ Q(a) \to B) \leftrightarrow \forall a\ (Q(a) \to B) \end{array}$$

Hence, all three parts of the statement (a) can be transformed into the same form.

For the statement (b) $a\,\mathbf{r}\,(P \subseteq Q) \leftrightarrow \mathbf{H}(P) \subseteq a^{-1} * \mathbf{R}(Q) \leftrightarrow a * \mathbf{H}(P) \subseteq \mathbf{R}(Q)$ we look at the first part of the statement and produce its equivalents:

$$\begin{array}{lll} & a\,\mathbf{r}\,(P \subseteq Q) & \\ \equiv & a\,\mathbf{r}\,(\forall \vec{x}(P(\vec{x}) \to Q(\vec{x}))) & \text{unfolding } \subseteq \\ \equiv & \forall \vec{x}\ a\,\mathbf{r}\,(P(\vec{x}) \to Q(\vec{x}))) & \text{def. } a\,\mathbf{r} \forall \vec{x}\ Q(\vec{x}) \\ \equiv & \forall \vec{x}\ (\mathbf{H}(P)(\vec{x}) \to a\,\mathbf{r}\,Q(\vec{x})) & \text{since } P \text{ is Harrop} \end{array}$$

Now if we unfold the definition of $\subseteq$ in $\mathbf{H}(P) \subseteq a^{-1} * \mathbf{R}(Q)$ and apply $(\vec{x})$ to it:

$$\begin{array}{lll} & \forall \vec{x}\ (\mathbf{H}(P)(\vec{x}) \to a^{-1} * \mathbf{R}(Q)(\vec{x})) & \\ \equiv & \forall \vec{x}\ (\mathbf{H}(P)(\vec{x}) \to \mathbf{R}(Q)(\vec{x}, a)) & \text{def. } 3 \\ \equiv & \forall \vec{x}\ (\mathbf{H}(P)(\vec{x}) \to a\,\mathbf{r}\,Q(\vec{x})) & \text{def. of } \mathbf{R}(Q)(\vec{x}, a) \end{array}$$

Lastly, we transform $a * \mathbf{H}(P) \subseteq \mathbf{R}(Q)$ in a similar way, applying $(\vec{x}, b)$ to it to obtain the following equivalent statements:

$$\begin{array}{lll} & \forall \vec{x}, b\ (a * \mathbf{H}(P)(\vec{x}, b) \to \mathbf{R}(Q)(\vec{x}, b)) & \\ \equiv & \forall \vec{x}, b\ (a = b \wedge \mathbf{H}(P)(\vec{x}) \to \mathbf{R}(Q)(\vec{x}, b)) & \text{def. } 4 \\ \equiv & \forall \vec{x}\ (\mathbf{H}(P)(\vec{x}) \to \mathbf{R}(Q)(\vec{x}, a)) & \text{let } b = a \end{array}$$

Hence, all three parts of the statement (b) can be transformed into the same form.

Now for the statement (c) $\mathbf{H}(Q \subseteq P) \leftrightarrow \mathbf{R}(Q) \subseteq \Delta\mathbf{H}(P) \leftrightarrow \exists(\mathbf{R}(Q)) \subseteq \mathbf{H}(P)$ we know that $\mathbf{H}(Q \subseteq P)$ stands for $\forall\vec{x}\ (\exists a\ (a\,\mathbf{r}\,Q(\vec{x})) \to \mathbf{H}(P)(\vec{x}))$ (by the definition of realizability since Q is non-Harrop and P is Harrop).

Now if we unfold the definition of $\subseteq$ and apply arbitrary $(\vec{x}, a)$ to $\mathbf{R}(Q) \subseteq \Delta\mathbf{H}(P)$, so we obtain the following equivalent formulas:

$$\begin{array}{lll} & \forall\vec{x}, a\ (\mathbf{R}(Q)(\vec{x}, a) \to \Delta(\mathbf{H}(P))(\vec{x}, a)) & \\ \equiv & \forall\vec{x}, a\ (\mathbf{R}(Q)(\vec{x}, a) \to \mathbf{H}(P)(\vec{x})) & \text{def. 5} \\ \equiv & \forall\vec{x}\ (\exists a\ (a\,\mathbf{r}\,Q(\vec{x})) \to \mathbf{H}(P)(\vec{x})) & \text{def. of } \mathbf{R}(Q)(a) \text{ and} \\ & & \text{since } (\exists a\ Q(a) \to B) \leftrightarrow \forall a\ (Q(a) \to B) \end{array}$$

Lastly, we transform $\exists(\mathbf{R}(Q)) \subseteq \mathbf{H}(P)$ by unfolding the definition of $\subseteq$ and applying $(\vec{x})$ to both sides of the implication:

$$\begin{array}{lll} & \forall\vec{x}\ (\exists(\mathbf{R}(Q))(\vec{x}) \to \mathbf{H}(P)(\vec{x})) & \\ \equiv & \forall\vec{x}\ (\exists a\ (\mathbf{R}(Q))(\vec{x}, a) \to \mathbf{H}(P)(\vec{x})) & \text{def. 6} \\ \equiv & \forall\vec{x}\ (\exists a\ (a\,\mathbf{r}\,Q(\vec{x})) \to \mathbf{H}(P)(\vec{x})) & \text{def. of } \mathbf{R}(Q)(a) \end{array}$$

Hence, all three parts of the statement (c) can be transformed into the same form.

For the statement (d) $\mathbf{H}(P \subseteq P') \leftrightarrow \mathbf{H}(P) \subseteq \mathbf{H}(P')$ we know that $\mathbf{H}(Q \subseteq P)$ stands for $\forall\vec{x}\ (\mathbf{H}(P)(\vec{x}) \to \mathbf{H}(P')(\vec{x}))$ (by the definition of realizability and since P and P' are Harrop). On right-hand side, if we rewrite the inclusion $\mathbf{H}(P) \subseteq \mathbf{H}(P')$, by the definition of inclusion, we also obtain $\forall\vec{x}\ (\mathbf{H}(P)(\vec{x}) \to \mathbf{H}(P')(\vec{x}))$.

For the statement (e) $\mathbf{R}(Q\ \cap\ Q') \equiv \pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q)\ \cap\ \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(Q')$, firstly, we work with the left-hand side:

$$\begin{array}{lll} & (\mathbf{R}(Q \cap Q'))(\vec{x}, c) & \\ \equiv & c\,\mathbf{r}\,((Q \cap Q')(\vec{x})) & \text{def. of } \mathbf{R} \\ \equiv & c\,\mathbf{r}\,(Q(\vec{x}) \wedge Q'(\vec{x})) & \text{by distributivity} \\ \equiv & (\pi_{\mathbf{Lt}} c)\,\mathbf{r}\,Q(\vec{x}) \wedge (\pi_{\mathbf{Rt}} c)\,\mathbf{r}\,Q'(\vec{x}) & \text{def. } c\,\mathbf{r}\,Q \wedge Q' \end{array}$$

Now we proceed with the right-hand side $\pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q)\ \cap\ \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(Q')$, unfolding the definition of an intersection and applying $(\vec{x}, c)$ to the both sides of the

conjunction:

$$
\begin{array}{lll}
 & (\pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q))(\vec{x},c) \wedge (\pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(Q'))(\vec{x},c) & \\
\equiv & \mathbf{R}(Q)(\vec{x},(\pi_{\mathbf{Lt}}c)) \wedge \mathbf{R}(Q')(\vec{x},(\pi_{\mathbf{Rt}}c)) & \text{def. 1} \\
\equiv & (\pi_{\mathbf{Lt}}c)\,\mathbf{r}\,Q(\vec{x}) \wedge (\pi_{\mathbf{Rt}}c)\,\mathbf{r}\,Q'(\vec{x}) & \text{def. of } \mathbf{R}
\end{array}
$$

Therefore, we can conclude that the statement (e) holds.

For the statement (f) $\mathbf{R}(Q \cup Q') \equiv \mathbf{Lt} \circ \mathbf{R}(Q) \cup \mathbf{Rt} \circ \mathbf{R}(Q')$ we first proceed with the left-hand side and apply $(\vec{x},c)$:

$$
\begin{array}{lll}
 & \mathbf{R}(Q \cup Q')(\vec{x},c) & \\
\equiv & c\,\mathbf{r}\,(Q \cup Q')(\vec{x}) & \text{def. of } \mathbf{R} \\
\equiv & c\,\mathbf{r}\,(Q(\vec{x}) \vee Q'(\vec{x})) & \text{unfolding def. of} \cap \\
\equiv & \exists a\ (c = \mathbf{Lt}(a) \wedge \mathbf{R}(Q)(\vec{x},a)) \vee \exists b\ (c = \mathbf{Rt}(b) \wedge \mathbf{R}(Q')(\vec{x},b)) & \text{def. } c\,\mathbf{r}\,Q \vee Q'
\end{array}
$$

Now we proceed with the right-hand side $\mathbf{Lt} \circ \mathbf{R}(Q) \cup \mathbf{Rt} \circ \mathbf{R}(Q')$, unfolding the definition of an intersection and applying $(\vec{x},c)$ to it:

$$
\begin{array}{lll}
 & (\mathbf{Lt} \circ \mathbf{R}(Q))(\vec{x},c) \vee (\mathbf{Rt} \circ \mathbf{R}(Q'))(\vec{x},c) & \\
\equiv & \exists a\ (\mathbf{Lt}(a) = c \wedge \mathbf{R}(Q)(\vec{x},a)) \vee \exists b\ (\mathbf{Rt}(b) = c \wedge \mathbf{R}(Q')(\vec{x},b)) & \text{def. 2}
\end{array}
$$

Since $\mathbf{Lt}(a) = c$ and $c = \mathbf{Lt}(a)$ are equivalent, we can conclude that the statement $\mathbf{R}(Q \cup Q') \equiv \mathbf{Lt} \circ \mathbf{R}(Q) \cup \mathbf{Rt} \circ \mathbf{R}(Q')$ holds.

For the statement (g) $\mathbf{R}(Q \cap P) \equiv \mathbf{R}(Q) \cap \Delta(\mathbf{H}(P))$, firstly, we work with the left-hand side:

$$
\begin{array}{lll}
 & \mathbf{R}(Q \cap P)(\vec{x},a) & \\
\equiv & a\,\mathbf{r}\,(Q \cap P)(\vec{x}) & \text{def. of } \mathbf{R} \\
\equiv & a\,\mathbf{r}\,(Q(\vec{x}) \wedge P(\vec{x})) & \text{unfolding def. of} \cap \\
\equiv & a\,\mathbf{r}\,Q(\vec{x}) \wedge \mathbf{H}(P)(\vec{x}) & \text{def. } a\,\mathbf{r}\,Q \wedge P
\end{array}
$$

Now we proceed with the right-hand side $\mathbf{R}(Q) \cap \Delta(\mathbf{H}(P))$, unfolding the definition of an intersection and applying $(\vec{x},a)$ to the both sides of the conjunction:

$$
\begin{array}{lll}
 & \mathbf{R}(Q)(\vec{x},a) \wedge \Delta\mathbf{H}(P)(\vec{x},a) & \\
\equiv & a\,\mathbf{r}\,Q(\vec{x}) \wedge \mathbf{H}(P)(\vec{x}) & \text{def. of } \mathbf{R} \text{ and } \Delta\mathbf{H}
\end{array}
$$

Therefore, we can conclude that the statement (g) holds.

To prove item (h), $\exists(\mathbf{R}(Q)) \cup \mathbf{H}(P) \equiv \exists(\mathbf{R}(Q \cup P))$, we proceed with the left-hand side:

$$\begin{aligned}
& \exists(\mathbf{R}(Q)) \cup \mathbf{H}(P) \\
= \ & \lambda x \ (\exists(\mathbf{R}(Q))(x) \vee \mathbf{H}(P)(x)) \\
= \ & \lambda x (\exists a \ \mathbf{R}(Q)(x,a) \vee \mathbf{H}(P)(x))
\end{aligned}$$

On the right-hand side:

$$\begin{aligned}
& \exists(\mathbf{R}(Q \cup P)) \\
= \ & \lambda x \ \exists c \ \mathbf{R}(\lambda y (Q(y) \vee P(y)))(x,c) \\
= \ & \lambda x \ \exists c \ ((\lambda (y,a) \ \mathbf{R}(Q(y) \vee P(y))(a))(x,c)) \\
= \ & \lambda x \ \exists c \ (\mathbf{R}(Q(x) \vee P(x))(c)) \\
= \ & \lambda x \ \exists c \ (\exists a \ (c = \mathbf{Lt}(a) \wedge \mathbf{R}(Q(x))(a)) \vee \exists b \ (c = \mathbf{Lt}(b) \wedge \mathbf{R}(P(x))(b))) \\
= \ & \lambda x \ \exists c \ (\exists a \ (c = \mathbf{Lt}(a) \wedge \mathbf{R}(Q(x))(a)) \vee \exists b \ (c = \mathbf{Lt}(b) \wedge \mathbf{H}(P(x)) \wedge b = \mathbf{Nil})) \\
\equiv \ & \lambda x \ \exists c \ (\exists a \ (c = \mathbf{Lt}(a) \wedge \mathbf{R}(Q(x))(a)) \vee (c = \mathbf{Lt}(\mathbf{Nil}) \wedge \mathbf{H}(P(x)))) \\
\equiv \ & \lambda x \ (\exists a \ \mathbf{R}(Q(x))(a) \vee \mathbf{H}(P(x))
\end{aligned}$$

The last two equivalences follow by easy equality reasoning.

□

As mentioned earlier, in this thesis we are proving an IFP' version of soundness. In a general sense, the theorem corresponds to the IFP version of soundness, where IFP proofs are replaced by the IFP' proofs. To ensure that IFP is indeed embedded in IFP', we prove the following two lemmas about monotonicity of s.p. operators in IFP' as well as the actual embedding lemma.

Lemma 9 IFP' proves that every strictly positive operator is monotone.

Proof. We show that for every s.p. operator Φ:

(a) IFP' proves $Mon(\Phi)$[1], i.e., $X \subseteq Y \to \Phi(X) \subseteq \Phi(Y)$, where X, Y are different predicate variables that are not free in Φ.

(b) Simultaneously, we prove that IFP' proves

  (i) $X \subseteq Y \to P \subseteq P[Y/X]$ and

[1] The algorithm that produces for each s.p. operator Φ the proof of Lemma 9(a) is part of the extraction procedure implemented in our proof assistant.

(ii) $X \subseteq Y \to A \to A[Y/X]$,

where P resp. A range over predicates resp. formulas that are s.p. in X and do not contain Y free.

The proof is by simultaneous induction on the syntactic complexity of Φ, P and A. For most cases the proof is straightforward, so here we present the most interesting cases, namely (b), where P is defined by induction or coinduction.

Firstly, we define $Mon_X(P)$ as $X \subseteq X' \to P \subseteq P[X'/X]$, where X' is a fresh variable accompanied with X. This means that for the operator $\Phi = \lambda X\, P$, $Mon(\Phi)$ is equivalent to $Mon_X(P)$. Hence, we proceed by structural induction on P. Assume $Mon_Y(Q)$ holds for every subterm Q of P and every variable Y. We need to show $Mon_X(P)$.

Case $P = \mu(\lambda Y\, Q)$. Assume $X \subseteq X'$. Show $\mu(\lambda Y\, Q) \subseteq \mu(\lambda Y\, Q[X'/X])$. We also assume that $Y \notin \{X, X'\}$. Using IFP' induction on $\mu(\lambda Y\, Q)$, we transform the goal and need to show that (i) $Q[\mu(\lambda Y\, Q[X'/X])/Y] \subseteq \mu(\lambda Y\, Q[X'/X])$ and (ii) $Mon(\lambda Y\, Q)$, i.e. $Mon_Y(Q)$. The statement of (ii) holds by the i.h., and $Mon_X Q$ also holds. Hence, $Q \subseteq Q[X'/X]$. By Lemma 3, $Q[\mu(\lambda Y\, Q[X'/X])/Y]$ is a subset of $Q[X'/X][\mu(\lambda Y\, Q[X'/X])/Y]$. Moreover, $Q[X'/X][\mu(\lambda Y\, Q[X'/X])/Y] \subseteq \mu(\lambda Y\, Q[X'/X])$ by closure. Hence, (i) holds.

Case $P = \nu(\lambda Y\, Q)$. Here the argument is completely dual if we replace $Mon_X P$ by the equivalent formula $X' \subseteq X \to P[X'/X] \subseteq P$. $\square$

Lemma 10 (Embedding). If $\Gamma \vdash_{IFP} A$, then $\Gamma \vdash_{IFP'} A$.

Proof. Proof by induction on the IFP derivation of A. For most rules the assertion follows trivially from the induction hypotheses. We present the most interesting cases.

IND. Assume $\mu(\Phi) \subseteq P$ has been derived in IFP by induction, that is, from a (shorter) IFP proof of $\Phi(P) \subseteq P$. By induction hypothesis IFP' proves $\Phi(P) \subseteq P$. Furthermore, by lemma 9, IFP' proves $Mon(\Phi)$. Hence, by the rule IND', IFP' proves $\mu(\Phi) \subseteq P$.

COIND. Assume $P \subseteq \nu(\Phi)$ has been derived in IFP by coinduction, that is, from a (shorter) IFP proof of $P \subseteq \Phi(P)$. Similarly to the above case, since IFP' proves $P \subseteq \Phi(P)$ and $Mon(\Phi)$, by the COIND' rule, IFP' proves $P \subseteq \nu(\Phi)$.

The remaining variations of induction and coinduction are proven in a similar way. □

Now we can begin proving the Soundness Theorem. Here we continue using infix notation $a \mathbf{r} A$ for readability. We use $\vec{a} \mathbf{r} \Gamma$ as a shorthand for $a_1 \mathbf{r} \Gamma_1 \dots a_n \mathbf{r} \Gamma_n$.

Theorem 1 If A can be derived from Δ and Γ in IFP', where Δ is a set of Harrop formulas and Γ is a set of non-Harrop formulas, then for every vector of distinct program variables $\vec{a}$ of the same length as Γ, there exists a program p with $\mathrm{FV}(p) \subseteq \vec{a}$, such that $\mathbf{H}(\Delta), \vec{a} \mathbf{r} \Gamma \vdash_{\mathrm{RIFP}} p \mathbf{r} A$.

Proof. By induction on the length of IFP' derivations.

We fix vector $\vec{a}$ of program variables of the same length as Γ.

Use rule. Assume $\Delta, \Gamma \vdash_{\mathrm{IFP'}} A$, where $A \in \Gamma$ in case A is non-Harrop or $A \in \Delta$ in case A is Harrop. We show $\mathbf{H}(\Delta), \vec{a} \mathbf{r} \Gamma \vdash_{\mathrm{RIFP}} s \mathbf{r} A$ for some s. For the non-Harrop case there is $a \mathbf{r} A \in \vec{a} \mathbf{r} \Gamma$, we apply the use rule and set $s = a$ to obtain $\mathrm{H}(\Delta), \vec{a} \mathbf{r} \Gamma \vdash_{\mathrm{RIFP}} a \mathbf{r} A$. In the Harrop case $s = \mathbf{Nil}$ and the proof is using the same approach.

For the axiom application the proof is straightforward as our axioms are always Harrop.

We first look at the introduction rules.

$\wedge^+$. Assume $\Delta, \Gamma \vdash_{\mathrm{IFP'}} (A \wedge B)$ has been derived from $\Delta, \Gamma \vdash_{\mathrm{IFP'}} A$ and $\Delta, \Gamma \vdash_{\mathrm{IFP'}} B$.

We proceed by looking at the different cases based on whether the subformulas are Harrop or non-Harrop.

If both A and B are non-Harrop, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} s\,\mathbf{r}\,(A \wedge B)$ for some s. Since neither A nor B are Harrop by definition of realizability $s\,\mathbf{r}\,(A \wedge B) = (\pi_{\mathbf{Lt}} s)\,\mathbf{r}\,A \wedge (\pi_{\mathbf{Rt}} s)\,\mathbf{r}\,B$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,A$ and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} q\,\mathbf{r}\,B$. So, using $\wedge^{+}$, we get $s\,\mathbf{r}\,(A \wedge B)$, where $s = \mathbf{Pair}(p,q)$ since $\pi_{\mathbf{Lt}} s = p$ and $\pi_{\mathbf{Rt}} s = q$.

If both A and B are Harrop, it suffices to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(A \wedge B)$. By definition of realizability $\mathbf{H}(A \wedge B) = \mathbf{H}(A) \wedge \mathbf{H}(B))$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(A)$ and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(B)$. Hence, by $\wedge^{+}$, $\mathbf{H}(A \wedge B)$ holds.

If A is Harrop and B is non-Harrop, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} s\,\mathbf{r}\,(A \wedge B)$ for some s. By definition of realizability, it suffices to show $\mathbf{H}(A) \wedge s\,\mathbf{r}\,B$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(A)$ and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} q\,\mathbf{r}\,B$. Hence, by $\wedge^{+}$, $s = q$ and $q\,\mathbf{r}\,(A \wedge B)$.

If A is non-Harrop and B is Harrop, using the same approach we show $p\,\mathbf{r}\,(A \wedge B)$ from i.h. $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,A$ and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(B)$.

$\vee_l^{+}$. Assume $\Delta, \Gamma \vdash_{\mathrm{IFP'}} (A \vee B)$ has been derived from $\Delta, \Gamma \vdash_{\mathrm{IFP'}} A$ by $\vee_l^{+}$.
A disjunctive formula is always non-Harrop, regardless whether A and B are Harrop or not. We show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} s\,\mathbf{r}\,(A \vee B)$. By definition of realizability $s\,\mathbf{r}\,(A \vee B) = \exists a (s = \mathbf{Lt}(a) \wedge a\,\mathbf{r}\,A \vee s = \mathbf{Rt}(a) \wedge a\,\mathbf{r}\,B)$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,A$, regardless of whether A is Harrop or non-Harrop. This is because if A is Harrop, then we can set $p = \mathbf{Nil}$. Hence, by $\vee^{+}$, we get $s\,\mathbf{r}\,(A \vee B)$, where $s = \mathbf{Lt}(p)$. The proof for $\vee_r^{+}$ is similar and we can show $s\,\mathbf{r}\,(A \vee B)$, where $s = \mathbf{Rt}(p)$.

$\to^{+}$. Assume $\Delta, \Gamma \vdash_{\mathrm{IFP'}} (A \to B)$ has been derived from $\Delta, \Gamma, A \vdash_{\mathrm{IFP'}} B$ by $\to^{+}$.
If both A and B are non-Harrop, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} s\,\mathbf{r}\,(A \to B)$ for some s. We apply $\to^{+}$ to the i.h. $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma, \mathbf{r}A \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,B$ and obtain $a\,\mathbf{r}\,A \to p\,\mathbf{r}\,B$. Since a is not free in the remaining assumptions, we apply $\forall^{+}$ and obtain $\forall a (a\,\mathbf{r}\,A \to p\,\mathbf{r}\,B)$. Since $(\lambda a\ p) a = p$, we set $s = \lambda a\ p$, so $(\lambda a\ p)\,\mathbf{r}\,(A \to B)$.

If both A and B are Harrop, is suffices to show $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(A \to B)$.
By definition of realizability $\mathbf{H}(A \to B) = \mathbf{H}(A) \to \mathbf{H}(B)$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma, \mathbf{H}(A) \vdash_{\mathrm{RIFP}} \mathbf{H}(B)$. Hence, by $\to^{+}$, we get $\mathbf{H}(A \to B)$.

If A is Harrop and B is non-Harrop, we need to show $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} s\,\mathbf{r}\,(A \to B)$ for some s. By definition of realizability, $b\,\mathbf{r}\,(A \to B) = \mathbf{H}(A) \to p\,\mathbf{r}\,B$. By i.h. $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma, \mathbf{H}(A) \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,B$. Hence, by $\to^{+}$, we get $p\,\mathbf{r}\,(A \to B)$ and $s = p$.

If $\underline{A \text{ is non-Harrop and } B \text{ is Harrop}}$ by the definition $\mathbf{H}(A \to B) = (\exists a\; a\mathbf{r}A) \to \mathbf{H}(B)$. We apply $\to^+$ to the i.h. $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma, \mathbf{r}A \vdash_{\text{RIFP}} \mathbf{H}(B)$ and obtain $a\mathbf{r}A \to \mathbf{H}(B)$. Since a is not free in the remaining assumptions, we apply $\forall^+$ and obtain $\forall a(a\mathbf{r}A \to \mathbf{H}(B))$. This is logically equivalent to $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma, p\mathbf{r}A \vdash (\exists a\; a\mathbf{r}A) \to \mathbf{H}(B)$ since a is not free in $\mathbf{H}(B)$. Hence, we can conclude that $\mathbf{H}(A \to B)$ holds.

$\underline{\forall^+}$. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} \forall x\, A$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} A$ by $\forall^+$.

If $\underline{A \text{ is non-Harrop}}$, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} s\,\mathbf{r}\,(\forall x\, A)$ for some s. By definition of realizability $a\,\mathbf{r}\,(\forall x\, A)$ is equal to $\forall x(a\mathbf{r}A)$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} p\mathbf{r}A$, so using $\forall^+$ (note that the variable x should not occur free in the context in this case), we get $p\,\mathbf{r}\,(\forall x\, A)$ and $s = p$.

If $\underline{A \text{ is Harrop}}$, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(\forall x\, A)$. Since A is Harrop, by definition of realizability $\mathbf{H}(\forall x\, A) = \forall x(\mathbf{H}(A))$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(A)$ and using $\forall^+$, we get $\mathbf{H}(\forall x\, A)$.

$\underline{\exists^+}$. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} \exists x\, A$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} A[t/x]$ by $\exists^+$.

If $\underline{A \text{ is non-Harrop}}$, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} s\,\mathbf{r}\,(\exists x\, A)$. By definition of realizability $a\,\mathbf{r}\,(\exists x\, A) = \exists x\; (a\mathbf{r}A)$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} p\,\mathbf{r}\,(A[t/x])$, which, by Lemma 5 is equivalent to $(p\mathbf{r}A)[t/x]$, so using $\exists^+$, we get $p\,\mathbf{r}\,(\exists x\, A)$.

If $\underline{A \text{ is Harrop}}$, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(\exists x\, A)$. By definition of realizability $\mathbf{H}(\exists x\, A) = \exists x\; \mathbf{H}(A)$. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(A[t/x])$, which by Lemma 5 is equivalent to $(\mathbf{H}(A))[t/x]$, so using $\exists^+$, we get $\mathbf{H}(\exists x\, A)$.

Now for the elimination rules we proceed accordingly.

$\underline{\wedge_l^-}$. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} A$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} A \wedge B$ by $\wedge_l^-$.

If $\underline{A \wedge B \text{ is non-Harrop}}$ then A is also non-Harrop, so we need to show an s such that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} s\mathbf{r}A$ in case A is non-Harrop and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(A)$ if A is Harrop. By the i.h. we have a program p such that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} p\mathbf{r}A \wedge B$.

- If A and B are both non-Harrop, by the definition of realizability we have $p\,\mathbf{r}\,(A \wedge B) = (\pi_{\mathbf{Lt}}p)\,\mathbf{r}A \wedge (\pi_{\mathbf{Rt}}p)\,\mathbf{r}B$. Hence, by $\wedge_l^-$, we get $(\pi_{\mathbf{Lt}}p)\,\mathbf{r}A$ and $s = (\pi_{\mathbf{Lt}}p)$.

- If A is Harrop and B is non-Harrop, by the definition of realizability we have $p\,\mathbf{r}\,(A \wedge B) = \mathbf{H}(A) \wedge p\mathbf{r}B$. Hence, by $\wedge_l^-$, we get $\mathbf{H}(A)$.

- If A is non-Harrop and B is Harrop, by the definition of realizability we have $p\,\mathbf{r}\,(A \wedge B) = p\,\mathbf{r}A \wedge \mathbf{H}(B)$. Hence, by $\wedge_l^-$, we get $(\pi_{\mathbf{Lt}}p)\,\mathbf{r}A$ and $s = (\pi_{\mathbf{Lt}}p)$.

If $A \wedge B$ is Harrop, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(A)$. Both A and B should also be Harrop in this case, so the definition of realizability is $\mathbf{H}(A \wedge B) = \mathbf{H}(A) \wedge \mathbf{H}(B)$. By the induction hypothesis $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(A \wedge B)$. Applying the $\wedge_l^-$ rule, we get $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(A)$.

We prove the theorem for the $\wedge^{-r}$ in the similar manner.

$\underline{\vee^-}$. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} C$ has been derived by $\vee^-$ from (a) $\Delta, \Gamma \vdash_{\text{IFP'}} A \vee B$, (b) $\Delta, \Gamma \vdash_{\text{IFP'}} A \to C$ and (c) $\Delta, \Gamma \vdash_{\text{IFP'}} B \to C$. If all the involved formulas are non-Harrop, then by i.h. we have three programs q, p_1 and p_2, such that

- $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} q\,\mathbf{r}\,(A \vee B)$
- $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} p_1\,\mathbf{r}\,(A \to C)$
- $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} p_2\,\mathbf{r}\,(B \to C)$

We expect that the extracted program s realizing C can be defined as $s =$ **case** q **of** $\{\mathbf{Lt}(a) \to p_1\ a;\ \mathbf{Rt}(b) \to p_2\ b\}$. To verify this we proceed as follows. By realizability interpretation, we know that $q\,\mathbf{r}A \vee B = (\exists a\ (q = \mathbf{Lt}(q) \wedge a\,\mathbf{r}A)) \vee (\exists b\ (q = \mathbf{Rt}(b) \wedge b\,\mathbf{r}B))$. We apply $\vee^-$ and distinguish two cases.

- Case $\exists a(q = \mathbf{Lt}(q) \wedge a\,\mathbf{r}A)$: Using $\exists^-$, we may assume that $q = \mathbf{Lt}(a)$ and $a\,\mathbf{r}A$ for some a. Using a program axiom it follows $s = p_1\ a$. However, since $p_1\,\mathbf{r}\,(A \to B)$ and $a\,\mathbf{r}A$, this implies $(p_1\ a)\,\mathbf{r}C$. Therefore, $s\,\mathbf{r}C$, by congruence.
- Case $\exists a(q = \mathbf{Rt}(q) \wedge q\,\mathbf{r}B)$ is similar.

If one of A or B or both are Harrop, the definition of s and the proof are similar to the above.

In case C is Harrop, it suffices to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(C)$. By the definition of realizability we know that $\mathbf{H}(\diamond \to C) = \mathbf{r}\,(\diamond) \to \mathbf{H}(C)$ regardless of whether $\diamond$ is Harrop or not. Here $\diamond \in \{A, B\}$. Using the $\vee^-$ rule with i.h. $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} q\,\mathbf{r}\,(A \vee B)$ and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} \mathbf{H}(\diamond \to C)$, we obtain our goal.

$\underline{\to^-}$. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} B$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} A \to B$ and $\Delta, \Gamma \vdash_{\text{IFP'}} A$ by $\to^-$.

If B is non-Harrop, we need to show $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\text{RIFP}} s\,\mathbf{r}B$ for some s.

If A is non-Harrop, then by the definition of realizability $f\,\mathbf{r}\,(A \to B) = \forall a(a\,\mathbf{r}A \to (f\ a)\,\mathbf{r}B)$. Using the $\to^-$ rule with the i.h. $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,(A \to B)$ and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} q\,\mathbf{r}A$, we get $(p\ q)\,\mathbf{r}B$, so $s = (p\ q)$.

In case A is Harrop, the proof is similar. We assign $s = p$ and show that $p\,\mathbf{r}B$.

If B is Harrop, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(B)$. Using the $\to^-$ rule with the i.h., regardless of whether A is Harrop or not, we obtain $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(B)$.

$\forall^-$. Assume $\Delta, \Gamma \vdash_{\mathrm{IFP'}} (A[t/x])$ has been derived from $\Delta, \Gamma \vdash_{\mathrm{IFP'}} \forall x\, A$ by $\forall^-$.

If A is non-Harrop, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} s\,\mathbf{r}\,(A[t/x])$. By i.h. we know that $p\,\mathbf{r}\,(\forall x\, A)$. Also, by definition of realizability $p\,\mathbf{r}\,(\forall x\, A) = \forall x\ (p\,\mathbf{r}A)$. Using $\forall^-$ rule, we get $(p\,\mathbf{r}A)[t/x]$, which is equivalent to $p\,\mathbf{r}\,(A[t/x])$ by Lemma 5, so $s = p$.

Similarly, for case that A is Harrop we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(A[t/x])$. By i.h. we know that $\mathbf{H}(\forall x\, A)$, which by definition of realizability is equal to $\forall x\ (\mathbf{H}(A))$. Using $\forall^-$, we get $(\mathbf{H}(A))[t/x]$ or, equivalently by Lemma 5, $\mathbf{H}(A[t/x])$.

$\exists^-$. Assume $\Delta, \Gamma \vdash_{\mathrm{IFP'}} B$ has been derived from $\Delta, \Gamma \vdash_{\mathrm{IFP'}} A \to \exists x\, A(x)$ and $\Delta, \Gamma \vdash_{\mathrm{IFP'}} \forall x(A(x) \to B)$ by $\exists^-$.

If B is non-Harrop, we need to show that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} s\,\mathbf{r}B$. Here we look at two subcases:

- $\exists x\, A(x)$ is non-Harrop: Here by i.h. we know $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,(\exists x\, A(x))$, the conclusion of which, by the definition of realizers, can be re-written as $\exists x(p\,\mathbf{r}A(x))$. Also, by i.h., $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} q\,\mathbf{r}\,(\forall x(A(x) \to B))$, the conclusion of which, by the definition of realizers, can be re-written as $\forall x(q\,\mathbf{r}\,((A(x) \to B))$. This can be rewritten further as $\forall x(\forall a(a\,\mathbf{r}A(x) \to (q\ a)\,\mathbf{r}B))$ and $\forall a, x(a\,\mathbf{r}A(x) \to (q\ a)\,\mathbf{r}B)$. By $\forall^-$ with $a = p$, we get $\forall x\ (p\,\mathbf{r}A(x) \to (q\ p)\,\mathbf{r}B)$. Using $\exists^-$, we get $(q\ p)\,\mathbf{r}B$.
- $\exists x\, A(x)$ is Harrop: similar. By i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(\exists x\, A(x))$ and $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} q\,\mathbf{r}\,(\forall x(A(x) \to B))$. Since $\exists x\, A(x)$ is Harrop and, consequently, so is $A(x)$, then by the definition of the realizability, this i.h. can be rewritten as $\forall x(\mathbf{H}(A(x)) \to q\,\mathbf{r}B)$, so applying the $\exists^-$ rule we get $q\,\mathbf{r}B$.

If B is Harrop, we need to show $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(B)$. Again, we look at two subcases:

- $\exists x\, A(x)$ is non-Harrop: by i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} p\,\mathbf{r}\,(\exists x\, A(x))$, the conclusion of which, by the definition of realizers, can be re-written

as $\exists x(p\,\mathbf{r}\,A(x))$. Since B is Harrop, then by the definition of realizers it means that $(\forall x(A(x) \to B))$ should also be Harrop. The conclusion of the i.h. $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(\forall x(A(x) \to B))$, can be re-written as $\forall x\ \mathbf{H}(A(x) \to B)$ and further as $\forall x(\exists a(a\,\mathbf{r}\,A(x) \to \mathbf{H}(B)))$. The latter is equivalent to the statement $\forall x\ \forall a(a\,\mathbf{r}\,A(x) \to \mathbf{H}(B))$. Applying the $\forall^-$ rule and using substitution, followed by the $\forall^+$ rule, we obtain $\forall x(p\,\mathbf{r}\,A(x) \to \mathbf{H}(B))$. Finally, we apply the $\exists^-$ rule to the latter and $\exists x(p\,\mathbf{r}\,A(x))$ to get $\mathbf{H}(B)$.

- $\exists x\, A(x)$ is Harrop: by i.h. we know that $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma \vdash_{\mathrm{RIFP}} \mathbf{H}(\exists x\, A(x))$, so by the definition of realizability $\exists x(\mathbf{H}(A(x))$. The proof is as above, where $p = \mathbf{Nil}$.

We now proceed with the proofs of soundness for different forms of induction and coinduction.

IND'. Assume $\Delta, \Gamma \vdash_{\mathrm{IFP'}} (\mu(\Phi) \subseteq P)$ has been derived from $\Delta, \Gamma \vdash_{\mathrm{IFP'}} (\Phi(P) \subseteq P)$ and $Mon(\Phi)$ by IND', where $\Phi(P) = Q[P/X]$. Here $Mon(\Phi)$ means $X \subseteq Y \to Q \subseteq Q[Y/X]$. We look at specific cases, depending on whether Φ and P are Harrop.

If Φ and P are non-Harrop:
Show:

$$
\begin{array}{lrl}
 & f\,\mathbf{r}\,(\mu(\Phi) \subseteq P) & \\
\equiv & \mathbf{R}(\mu(\Phi)) \subseteq f^{-1} \circ \mathbf{R}(P) & \text{Lem. 8} \\
= & \mathbf{R}(\mu(\lambda X\ Q)) \subseteq f^{-1} \circ \mathbf{R}(P) & \text{since } \Phi = \lambda X\ Q \\
= & (\mu(\lambda \tilde{X} \mathbf{R}(Q))) \subseteq f^{-1} \circ \mathbf{R}(P) & \text{since } \mathbf{R}(\mu(\Phi)) = \mu(\mathbf{R}(\Phi)) \\
 & & \text{and } \mathbf{R}(\lambda X\ Q) = \lambda \tilde{X}(\mathbf{R}(Q))
\end{array}
$$

By s.p. induction, it is enough to show

$$\mathbf{R}(Q)[f^{-1} \circ \mathbf{R}(P)/\tilde{X}] \subseteq f^{-1} \circ \mathbf{R}(P) \tag{10.1}$$

By i.h. we have $s\,\mathbf{r}\,(\Phi(P) \subseteq (P))$, which is, by Lemma 8 equivalent to

$$\mathbf{R}(Q[P/X]) \subseteq s^{-1} \circ \mathbf{R}(P) \tag{10.2}$$

By i.h. we also have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P/Y]) \tag{10.3}$$

Writing out $Mon(\Phi)[P/Y]$ we obtain $X \subseteq P \to Q \subseteq Q[P/X]$. Therefore, 10.3 can be rewritten as

$$\begin{array}{llr} & \forall g(g\,\mathbf{r}\,(X \subseteq P) \to (m\ g)\,\mathbf{r}\,(Q \subseteq Q[P/X])) & \\ \equiv & \forall g(\mathbf{R}(X) \subseteq g^{-1} \circ \mathbf{R}(P) \to \mathbf{R}(Q) \subseteq (m\ g)^{-1} \circ \mathbf{R}(Q[P/X])) & \text{Lem. 8} \\ = & \forall g(\tilde{X} \subseteq g^{-1} \circ \mathbf{R}(P) \to \mathbf{R}(Q) \subseteq (m\ g)^{-1} \circ \mathbf{R}(Q[P/X])) & \text{def. } \mathbf{R}(X) \end{array}$$

If we define g as f and $\tilde{X}$ as $f^{-1} \circ \mathbf{R}(P)$ and use Lemma 3, we get

$$\begin{array}{rll} \mathbf{R}(Q)[f^{-1} \circ \mathbf{R}(P)/\tilde{X}] & \subseteq (m\ f)^{-1} \circ \mathbf{R}(Q[P/X])) & \\ & \subseteq (m\ f)^{-1} \circ (s^{-1} \circ \mathbf{R}(P)) & \text{by 10.2} \\ & = (s\ \circ\ (m\ f))^{-1} \circ \mathbf{R}(P) & \text{Lem. 7} \end{array}$$

Hence, we define the realizer recursively as $f = s\ \circ\ (m\ f)$, that is, non-recursively, $f = \mathbf{rec}(\lambda f(s\ \circ\ (m\ f)))$. As a result, in the final statement we can replace the right-hand side $(s\ \circ\ (m\ f))^{-1} \circ \mathbf{R}(P)$ with $f^{-1} \circ \mathbf{R}(P)$ and, thus, show that eq. (10.1) holds.

If Φ and P are Harrop then $\mu(\Phi)$ and $Q[P/X]$ are also Harrop.

Show:

$$\begin{array}{lrl} & \mathbf{H}(\mu(\Phi) \subseteq P) & \\ \equiv & \mathbf{H}(\mu(\Phi)) \subseteq \mathbf{H}(P) & \text{Lem. 8} \\ = & \mathbf{H}(\mu\lambda X\ Q) \subseteq \mathbf{H}(P) & \text{since } \Phi = \lambda X\ Q \\ = & \mu(\lambda X\ \mathbf{H}_X(Q)) \subseteq \mathbf{H}(P) & \text{since } \mathbf{H}(\mu(\Phi)) = \mu\mathbf{H}(\Phi) \\ & & \text{and } \mathbf{H}(\lambda X\ Q) = \lambda X\ \mathbf{H}_X(Q) \end{array}$$

By s.p. induction, it is enough to show

$$\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \subseteq \mathbf{H}(P) \tag{10.4}$$

By i.h. we have: $\mathbf{H}(\Phi(P) \subseteq (P))$, which is, by Lemma 8 equivalent to

$$\mathbf{H}(Q[P/X]) \subseteq \mathbf{H}(P) \tag{10.5}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3 we can substitute X and Y the former implies

$$\mathbf{H}(Mon(\Phi)[\hat{X}/X][P/Y]) \tag{10.6}$$

Writing out $Mon(\Phi)[\hat{X}/X][P/Y]$ we obtain $\hat{X} \subseteq P \to Q[\hat{X}/X] \subseteq Q[P/X]$. If we define $\hat{X}$ as $\mathbf{H}(P)$, 10.6 can be rewritten as $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] \subseteq \mathbf{H}(Q[P/X])$ but also $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] = \mathbf{H}_X(Q)[\mathbf{H}(P)/X]$, so we obtain

$$\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \subseteq \mathbf{H}(Q[P/X]) \tag{10.7}$$

By 10.7 and 10.5 we get $\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \subseteq \mathbf{H}(P)$, which corresponds to our goal.

If Φ is non-Harrop and P is Harrop then $\mu(\Phi)$ and $Q[P/X]$ are non-Harrop.

Show:

$$\begin{array}{lll}
 & \mathbf{H}(\mu(\Phi) \subseteq P) & \\
\equiv & \mathbf{R}(\mu(\Phi)) \subseteq \Delta(\mathbf{H}(P)) & \text{Lem. 8} \\
= & \mathbf{R}(\mu\lambda X\ Q) \subseteq \Delta(\mathbf{H}(P)) & \text{since } \Phi = \lambda X\ Q \\
= & \mu(\lambda\tilde{X}\ \mathbf{R}(Q)) \subseteq \Delta(\mathbf{H}(P)) & \text{since } \mathbf{R}(\mu(\Phi)) = \mu\mathbf{R}(\Phi) \\
 & & \text{and } \mathbf{R}(\lambda X\ Q) = \lambda\tilde{X}\ \mathbf{R}(Q)
\end{array}$$

By s.p. induction, it is enough to show

$$\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] \subseteq \Delta(\mathbf{H}(P)) \tag{10.8}$$

By i.h. we have: $\mathbf{H}(\Phi(P) \subseteq (P))$, which is, by Lemma 8 equivalent to

$$\mathbf{R}(Q[P/X]) \subseteq \Delta(\mathbf{H}(P)) \tag{10.9}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P/Y]) \tag{10.10}$$

Writing out $Mon(\Phi)[P/Y]$ we obtain $X \subseteq P \to Q \subseteq Q[P/X]$. Since P is Harrop, 10.10 can be rewritten as

$$\begin{array}{lll}
 & \mathbf{H}(X \subseteq P) \to m\,\mathbf{r}\,(Q \subseteq Q[P/X])) & \\
\equiv & (\mathbf{R}(X) \subseteq \Delta(\mathbf{H}(P)) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[P/X]) & \text{Lem. 8} \\
= & \tilde{X} \subseteq \Delta(\mathbf{H}(P)) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[P/X])) & \text{def. } \mathbf{R}(X)
\end{array}$$

If we define $\tilde{X}$ as $\Delta(\mathbf{H}(P))$ and use Lemma 3, we get

$$\begin{array}{ll}
\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] \subseteq m^{-1} \circ \mathbf{R}(Q[P/X])) & \\
\qquad \subseteq m^{-1} \circ \Delta(\mathbf{H}(P)) & \text{by 10.9} \\
\qquad = \Delta(\mathbf{H}(P)) & \text{Lem. 7}
\end{array}$$

This corresponds to our goal.

If Φ is Harrop and P is non-Harrop then $\mu(\Phi)$ is Harrop.

Show:

$$\begin{array}{rrl}
 & a\,\mathbf{r}\,(\mu(\Phi) \subseteq P) & \\
\equiv & \mathbf{H}(\mu(\Phi)) \subseteq a^{-1} * \mathbf{R}(P) & \text{Lem. 8} \\
= & \mathbf{H}(\mu\lambda X\ Q) \subseteq a^{-1} * \mathbf{R}(P) & \text{since } \Phi = \lambda X\ Q \\
\text{if } X \in \mathrm{FV}(Q) = & \mu(\lambda X\ \mathbf{H}_X(Q)) \subseteq a^{-1} * \mathbf{R}(P) & \text{since } \mathbf{H}(\mu(\Phi)) = \mu\mathbf{H}(\Phi) \\
 & & \text{and } \mathbf{H}(\lambda X\ Q) = \lambda X\ \mathbf{H}_X(Q) \\
\text{if } X \notin \mathrm{FV}(Q) = & \mu(\lambda X\ \mathbf{H}(Q)) \subseteq a^{-1} * \mathbf{R}(P) & \text{as above but } \mathbf{H}_X = \mathbf{H}
\end{array}$$

We look at these two subcases:

(a) $X \notin \mathrm{FV}(Q)$

By s.p. induction it is suffices to show that $\mathbf{H}(Q) \subseteq a^{-1} * (\mathbf{R}(P))$. By i.h. we have $a\,\mathbf{r}\,(\Phi(P) \subseteq (P))$. By Lemma 8 the latter is equivalent to $\mathbf{H}(Q) \subseteq a^{-1} * (\mathbf{R}(P))$, which corresponds to our goal.

(b) $X \in \mathrm{FV}(Q)$

By s.p. induction, it is enough to show

$$\mathbf{H}_X(Q)[a^{-1} * (\mathbf{R}(P))/X] \subseteq a^{-1} * \mathbf{R}(P) \tag{10.11}$$

By i.h. we have: $s\,\mathbf{r}\,(\Phi(P) \subseteq (P))$, which is, by Lemma 8 equivalent to

$$\mathbf{R}(Q[P/X]) \subseteq s^{-1} \circ \mathbf{R}(P) \tag{10.12}$$

By i.h. we have $m\,\mathbf{r}\,(Mon(\Phi))$ and using Lemma 3, followed by Lemma 4(a), we obtain

$$m\,\mathbf{r}\,(Mon(\Phi)[\hat{X}/X][P/Y]) \tag{10.13}$$

Writing out $Mon(\Phi)[\hat{X}/X][P/Y]$ we obtain $\hat{X} \subseteq P \to Q[\hat{X}/X] \subseteq Q[P/X]$. Therefore, 10.13 can be rewritten as

$$\begin{array}{rll}
 & \forall b(a\,\mathbf{r}\,(\hat{X} \subseteq P) \to (m\ b)\,\mathbf{r}\,(Q[\hat{X}/X] \subseteq Q[P/X])) & \\
\equiv & \forall b(\mathbf{H}(\hat{X}) \subseteq b^{-1} * \mathbf{R}(P) \to \mathbf{H}_X(Q)[\hat{X}/X] \subseteq (m\ b)^{-1} * \mathbf{R}(Q[P/X])) & \text{Lem. 8} \\
= & \forall b(\hat{X} \subseteq b^{-1} * \mathbf{R}(P) \to \mathbf{H}_X(Q)[\hat{X}/X] \subseteq (m\ b)^{-1} * \mathbf{R}(Q[P/X])) & \text{def. } \mathbf{H}(\hat{X})
\end{array}$$

If we define b as a and $\hat{X}$ as $a^{-1} * \mathbf{R}(P)$, we obtain

$$\mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P)/X] \subseteq (m\ a)^{-1} * \mathbf{R}(Q[P/X]) \tag{10.14}$$

since $\mathbf{H}(Q[\hat{X}/X])[a^{-1} * (\mathbf{R}(P))/\hat{X}] = \mathbf{H}_X(Q)[a^{-1} * (\mathbf{R}(P))/X]$.

Now,

$$\begin{aligned}
\mathbf{H}_X(Q)[a^{-1} * (\mathbf{R}(P))/X] &\subseteq (m\ a)^{-1} * \mathbf{R}(Q[P/X]) \\
&\subseteq (m\ a)^{-1} * (s^{-1} \circ \mathbf{R}(P)) && \text{by 10.12} \\
&= (s(m\ a))^{-1} * \mathbf{R}(P)) && \text{Lem. 7}
\end{aligned}$$

Hence, we define the realizer recursively as $a = s(m\ a)$. As a result, in the final statement we can replace the right-hand side $(s(m\ a))^{-1} * \mathbf{R}(P))$ with $a^{-1} * \mathbf{R}(P)$ and, thus, show that eq. (10.11) holds.

HSI'. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} (\mu(\Phi) \subseteq P)$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} (\Phi(P) \cap \mu(\Phi) \subseteq P)$ and $Mon(\Phi)$ by HSI', where $\Phi(P) = Q[P/X]$. Here $Mon(\Phi)$ means $X \subseteq Y \to Q \subseteq Q[Y/X]$. We look at specific cases, depending on whether Φ and P are Harrop.

If Φ and P are non-Harrop then, as with the IND' case, we show $(\mu(\lambda\tilde{X}\mathbf{R}(Q))) \subseteq f^{-1} \circ \mathbf{R}(P)$. By s.p. half-strong induction, it is enough to show

$$(\mathbf{R}(Q)[f^{-1} \circ \mathbf{R}(P)/\tilde{X}]) \cap \mathbf{R}(\mu(\Phi)) \subseteq f^{-1} \circ \mathbf{R}(P) \tag{10.15}$$

By i.h. we have:

$$\begin{aligned}
& s\,\mathbf{r}\,(\Phi(P) \cap \mu(\Phi) \subseteq \mathbf{R}(P)) \\
\equiv \quad & \mathbf{R}(\Phi(P) \cap \mu(\Phi)) \subseteq s^{-1}\circ \subseteq \mathbf{R}(P)) && \text{Lem. 8} \\
\equiv \quad & \pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(\Phi(P)) \cap \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(\mu(\Phi)) \subseteq s^{-1}\circ \subseteq \mathbf{R}(P)) && \text{Lem. 8}
\end{aligned}$$

Since $\Phi(P) = Q[P/X]$, the above is equal to

$$\pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q[P/X]) \cap \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(\mu(\Phi)) \subseteq s^{-1}\circ \subseteq \mathbf{R}(P)) \tag{10.16}$$

$\mathbf{R}(\mu(\Phi)) = \mu\mathbf{R}(\Phi)$ and $\mathbf{R}(\lambda X\ Q) = \lambda\tilde{X}\ \mathbf{R}(Q)$, the above is equal to

$$\pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q[P/X]) \cap \pi_{\mathbf{Rt}}{}^{-1} \circ \mu(\lambda\tilde{X}\ \mathbf{R}(Q)) \subseteq s^{-1}\circ \subseteq \mathbf{R}(P)) \tag{10.17}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P/Y]) \tag{10.18}$$

Writing out $Mon(\Phi)[P/Y]$ we obtain $X \subseteq P \to Q \subseteq Q[P/X]$. Therefore, 10.18 can be rewritten as

$$\begin{array}{lll} & \forall g(g\,\mathbf{r}\,(X \subseteq P) \to (m\ g)\,\mathbf{r}\,(Q \subseteq Q[P/X])) & \\ \equiv & \forall g(\mathbf{R}(X) \subseteq g^{-1} \circ \mathbf{R}(P) \to \mathbf{R}(Q) \subseteq (m\ g)^{-1} \circ \mathbf{R}(Q[P/X])) & \text{Lem. 8} \\ = & \forall g(\tilde{X} \subseteq g^{-1} \circ \mathbf{R}(P) \to \mathbf{R}(Q) \subseteq (m\ g)^{-1} \circ \mathbf{R}(Q[P/X])) & \text{def. } \mathbf{R}(X) \end{array}$$

If we define g as f and $\tilde{X}$ as $f^{-1} \circ \mathbf{R}(P)$ and use Lemma 3, we get

$$\mathbf{R}(Q)[f^{-1} \circ \mathbf{R}(P)/\tilde{X}] \subseteq (m\ f)^{-1} \circ \mathbf{R}(Q[P/Y]) \tag{10.19}$$

Now,

$$\begin{array}{ll} \quad \mathbf{R}(Q)[f^{-1} \circ \mathbf{R}(P)/\tilde{X}] \,\cap\, \mathbf{R}(\mu(\Phi)) & \\ \subseteq ((m\ f)^{-1} \circ \mathbf{R}(Q[P/X])) \,\cap\, \mathbf{R}(\mu(\Phi)) & \\ = \langle (m\ f), \mathbf{id} \rangle^{-1} \circ (\pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q[P/X]) \,\cap\, \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(\mu(\Phi))) & \text{Lem. 7} \\ \subseteq \langle (m\ f), \mathbf{id} \rangle^{-1} \circ (s^{-1} \circ \mathbf{R}(P)) & \text{by 10.17} \\ = (s\ \circ \langle (m\ f), \mathbf{id} \rangle)^{-1} \circ \mathbf{R}(P) & \text{Lem. 7} \end{array}$$

Here and in the rest of the thesis **id** stands for identity.

We define the realizer recursively as $f = s\ \circ \langle (m\ f), \mathbf{id} \rangle$, that is, non-recursively, $f = \mathbf{rec}(\lambda f(s\ \circ \langle (m\ f), \mathbf{id} \rangle))$. As a result, in the final statement we can replace the right-hand side $(s\ \circ \langle (m\ f), \mathbf{id} \rangle)^{-1} \circ \mathbf{R}(P)$ with $f^{-1} \circ \mathbf{R}(P)$ and, thus, show that eq. (10.15) holds.

If Φ and P are Harrop then $\mu(\Phi)$ and $Q[P/X]$ are also Harrop. As with IND' case, we need to show $\mu(\lambda X\ \mathbf{H}_X(Q)) \subseteq \mathbf{H}(P)$. By s.p. half-strong induction, it suffices to show

$$\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \,\cap\, \mathbf{H}(\mu(\Phi)) \subseteq \mathbf{H}(P) \tag{10.20}$$

By i.h. we have: $\mathbf{H}((\Phi(P) \,\cap\, \mu(\Phi)) \subseteq P)$, which is, by Lemma 8 equivalent to

$$\mathbf{H}(Q[P/X]) \,\cap\, \mathbf{H}(\mu(\Phi)) \subseteq \mathbf{H}(P) \tag{10.21}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3 we can substitute X and Y, so the former implies

$$\mathbf{H}(Mon(\Phi)[\hat{X}/X][P/Y]) \tag{10.22}$$

Writing out $Mon(\Phi)[\hat{X}/X][P/Y]$ we obtain $\hat{X} \subseteq P \to Q[\hat{X}/X] \subseteq Q[P/X]$. If we define $\hat{X}$ as $\mathbf{H}(P)$, 10.22 can be rewritten as $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] \subseteq \mathbf{H}(Q[P/X])$. But also since $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] = \mathbf{H}_X(Q)[\mathbf{H}(P)/X]$, so we obtain

$$\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \subseteq \mathbf{H}(Q[P/X]) \tag{10.23}$$

Now, by 10.23 and 10.21 $\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \cap \mathbf{H}(\mu(\Phi))$ is a subset or it is equal to $\mathbf{H}(Q[P/X]) \cap \mathbf{H}(\mu(\Phi))$, so $\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \cap \mathbf{H}(\mu(\Phi)) \subseteq \mathbf{H}(P)$, which corresponds to our goal.

If Φ is non-Harrop and P is Harrop then $\mu(\Phi)$ and $Q[P/X]$ are non-Harrop. As with IND' case, we need to show $\mu(\lambda\tilde{X}\ \mathbf{R}(Q)) \subseteq \Delta(\mathbf{H}(P))$. By s.p. half-strong induction, it suffices to show

$$\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] \cap \mathbf{R}(\mu(\Phi)) \subseteq \Delta(\mathbf{H}(P)) \tag{10.24}$$

By i.h. we have:

$$\begin{array}{lcr}
 & \mathbf{H}(\Phi(P) \cap \mu(\Phi) \subseteq P) & \\
\equiv & \mathbf{R}(\Phi(P) \cap \mu(\Phi)) \subseteq \Delta(\mathbf{H}(P)) & \text{Lem. 8} \\
\equiv & \pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(\Phi(P)) \cap \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(\mu(\Phi)) \subseteq \Delta(\mathbf{H}(P)) & \text{Lem. 8}
\end{array}$$

Since $\Phi(P) = Q[P/X]$, the above is equal to

$$\pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q[P/X]) \cap \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(\mu(\Phi)) \subseteq \Delta(\mathbf{H}(P)) \tag{10.25}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P/Y]) \tag{10.26}$$

Since P is Harrop, similarly to the case of IND', 10.26 can be rewritten as $\tilde{X} \subseteq \Delta(\mathbf{H}(P)) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[P/X]))$.

$$\begin{array}{lcr}
 & \mathbf{H}(X \subseteq P) \to m\,\mathbf{r}\,(Q \subseteq Q[P/X]) & \\
\equiv & (\mathbf{R}(X) \subseteq \Delta(\mathbf{H}(P)) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[P/X]) & \text{Lem. 8} \\
= & \tilde{X} \subseteq \Delta(\mathbf{H}(P)) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[P/X]) & \text{def. } \mathbf{R}(X)
\end{array}$$

If we define $\tilde{X}$ as $\Delta(\mathbf{H}(P))$ and use Lemma 3, we get

$$\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] \subseteq m^{-1} \circ \mathbf{R}(Q[P/X])) \tag{10.27}$$

Now,

$$\begin{aligned}
&\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] \ \cap \ \mathbf{R}(\mu(\Phi)) \\
&\subseteq m^{-1} \circ \mathbf{R}(Q[P/X])) \ \cap \ \mathbf{R}(\mu(\Phi)) \\
&\subseteq \langle m, \mathbf{id} \rangle^{-1} \circ (\pi_{\mathbf{Lt}}{}^{-1} \circ \mathbf{R}(Q[P/X]) \ \cap \ \pi_{\mathbf{Rt}}{}^{-1} \circ \mathbf{R}(\mu(\Phi))) && \text{Lem. 7} \\
&\subseteq \langle m, \mathbf{id} \rangle^{-1} \circ \Delta(\mathbf{H}(P)) && \text{by 10.25} \\
&= \Delta(\mathbf{H}(P)) && \text{Lem. 7}
\end{aligned}$$

This corresponds to our goal.

If Φ is Harrop and P is non-Harrop then $\mu(\Phi)$ is Harrop.

We need to show $a\,\mathbf{r}\,(\mu(\Phi) \subseteq P)$, so as in case of IND', this goal is equivalent to the following:

$$\begin{aligned}
&\text{if } X \in \mathrm{FV}(Q): && \mu(\lambda X\ \mathbf{H}_X(Q)) \subseteq a^{-1} * \mathbf{R}(P) \\
&\text{if } X \notin \mathrm{FV}(Q): && \mu(\lambda X\ \mathbf{H}(Q)) \subseteq a^{-1} * \mathbf{R}(P)
\end{aligned}$$

We look at these two subcases:

(a) $X \notin \mathrm{FV}(Q)$

By s.p. half-strong induction it is suffices to show that $\mathbf{H}(Q) \ \cap \ \mathbf{H}(\mu(\Phi)) \subseteq a^{-1} * (\mathbf{R}(P))$. By i.h. we have $a\,\mathbf{r}\,((\Phi(P) \ \cap \ \mu(\Phi)) \subseteq P)$. By Lemma 8 the latter is equivalent to $\mathbf{H}(Q) \ \cap \ \mathbf{H}(\mu(\Phi)) \subseteq a^{-1} * \mathbf{R}(P)$, which corresponds to our goal.

(b) $X \in \mathrm{FV}(Q)$

By s.p. half-strong induction, it is enough to show

$$\mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P)/X] \ \cap \ \mathbf{H}(\mu(\Phi)) \subseteq a^{-1} * \mathbf{R}(P) \tag{10.28}$$

By i.h. we have: $s\,\mathbf{r}\,((\Phi(P) \ \cap \ \mu(\Phi)) \subseteq P)$, which is, by Lemma 8 equivalent to

$$\mathbf{R}(Q[P/X] \ \cap \ \mu(\Phi)) \subseteq s^{-1} \circ (\mathbf{R}(P)) \tag{10.29}$$

By i.h. we have $m\,\mathbf{r}\,(Mon(\Phi))$ and using Lemma 3, followed by Lemma 4(a), we obtain

$$m\,\mathbf{r}\,(Mon(\Phi)[\hat{X}/X][P/Y]) \tag{10.30}$$

Writing out $Mon(\Phi)[\hat{X}/X][P/Y]$ we obtain $\hat{X} \subseteq P \to Q[\hat{X}/X] \subseteq Q[P/X]$. Therefore, 10.30 can be rewritten as

$$\begin{array}{lll} & \forall b(a\,\mathbf{r}\,(\hat{X} \subseteq P) \to (m\ b)\,\mathbf{r}\,(Q[\hat{X}/X] \subseteq Q[P/X])) & \\ \equiv & \forall b(\mathbf{H}(\hat{X}) \subseteq b^{-1} * \mathbf{R}(P) \to \mathbf{H}_X(Q)[\hat{X}/X] \subseteq (m\ b)^{-1} * \mathbf{R}(Q[P/X])) & \text{Lem. 8} \\ = & \forall b(\hat{X} \subseteq b^{-1} * \mathbf{R}(P) \to \mathbf{H}_X(Q)[\hat{X}/X] \subseteq (m\ b)^{-1} * \mathbf{R}(Q[P/X])) & \text{def. } \mathbf{H}(\hat{X}) \end{array}$$

If we define b as a and $\hat{X}$ as $a^{-1} * (\mathbf{R}(P))$, we obtain

$$\mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P)/X] \subseteq (m\ a)^{-1} * \mathbf{R}(Q[P/X])) \tag{10.31}$$

since $\mathbf{H}(Q[\hat{X}/X])[a^{-1} * \mathbf{R}(P)/\hat{X}] = \mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P)/X]$. Now,

$$\begin{array}{ll} (\mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P)/X]) \cap \mathbf{H}(\mu(\Phi)) & \\ \subseteq ((m\ a)^{-1} * \mathbf{R}(Q[P/X])) \cap \mathbf{H}(\mu(\Phi)) & \\ = (m\ a)^{-1} * (\mathbf{R}(Q[P/X]) \cap \Delta(\mathbf{H}(\mu(\Phi)))) & \text{Lem. 7} \\ = (m\ a)^{-1} * \mathbf{R}(Q[P/X] \cap \mu(\Phi)) & \text{Lem. 7} \\ \subseteq (m\ a)^{-1} * (s^{-1} \circ \mathbf{R}(P)) & \text{by 10.29} \\ = (s(m\ a))^{-1} * \mathbf{R}(P) & \text{Lem. 7} \end{array}$$

Hence, we define the realizer recursively as $a = s(m\ a)$. As a result, in the final statement we can replace the right-hand side $((s(m\ a))^{-1} \circ \mathbf{R}(P)$ with $a^{-1} \circ \mathbf{R}(P)$ and, thus, show that eq. (10.28) holds.

SI'. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} (\mu(\Phi) \subseteq P)$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} (\Phi(P \cap \mu(\Phi)) \subseteq P)$ and $Mon(\Phi)$ by SI', where $\Phi(P \cap \mu(\Phi)) = Q[P \cap \mu(\Phi)/X]$. Here $Mon(\Phi)$ means $X \subseteq Y \to Q \subseteq Q[Y/X]$. We look at specific cases, depending on whether Φ and P are Harrop.

If Φ and P are non-Harrop then, as with IND' case, we show $(\mu(\lambda\tilde{X}\mathbf{R}(Q))) \subseteq f^{-1} \circ \mathbf{R}(P)$. By s.p. strong induction, it suffices to show

$$\mathbf{R}(Q)[f^{-1} \circ \mathbf{R}(P \cap \mu(\Phi))/\tilde{X}] \subseteq f^{-1} \circ \mathbf{R}(P) \tag{10.32}$$

By i.h. we have:

$$\begin{array}{lll} & s\,\mathbf{r}\,(\Phi(P \cap \mu(\Phi)) \subseteq \mathbf{R}(P)) & \\ \equiv & \mathbf{R}(\Phi(P \cap \mu(\Phi))) \subseteq s^{-1} \circ \mathbf{R}(P) & \text{Lem. 8} \end{array}$$

Since $\Phi(P \cap \mu(\Phi)) = Q[(P \cap \mu(\Phi))/X$:

$$\mathbf{R}(Q[(P \cap \mu(\Phi))/X]) \subseteq s^{-1} \circ \mathbf{R}(P) \tag{10.33}$$

By i.h. we have $m \mathbf{r} Mon(\Phi)$ and by Lemma 4(a) this implies

$$m \mathbf{r} (Mon(\Phi)[(P \cap \mu(\Phi))/Y]) \tag{10.34}$$

Writing out $Mon(\Phi)[P \cap \mu(\Phi)/Y]$ we obtain $X \subseteq (P \cap \mu(\Phi)) \to Q \subseteq Q[P \cap \mu(\Phi)/X]$. Therefore, 10.34 can be rewritten as

$$\begin{array}{lll} & \forall g(g\,\mathbf{r}(X \subseteq (P \cap \mu(\Phi))) \to (m\ g)\,\mathbf{r}(Q \subseteq Q[(P \cap \mu(\Phi))/X])) & \\ \equiv & \forall g(\mathbf{R}(X) \subseteq g^{-1} \circ \mathbf{R}(P \cap \mu(\Phi)) \to \mathbf{R}(Q) \subseteq (m\ g)^{-1} \circ \mathbf{R}(Q[(P \cap \mu(\Phi))/X])) & \text{Lem. 8} \\ = & \forall g(\tilde{X} \subseteq g^{-1} \circ \mathbf{R}(P \cap \mu(\Phi)) \to \mathbf{R}(Q) \subseteq (m\ g)^{-1} \circ \mathbf{R}(Q[(P \cap \mu(\Phi))/X])) & \text{def. } \mathbf{R}(X) \end{array}$$

If we define g as f and $\tilde{X}$ as $f^{-1} \circ \mathbf{R}(P \cap \mu(\Phi))$ and use Lemma 3, we get

$$\begin{array}{lll} \mathbf{R}(Q)[f^{-1} \circ \mathbf{R}(P \cap \mu(\Phi))/\tilde{X}] & \subseteq (m\ f)^{-1} \circ \mathbf{R}(Q[(P \cap \mu(\Phi))/X]) & \\ & \subseteq (m\ f)^{-1} \circ (s^{-1} \circ \mathbf{R}(P)) & \text{by 10.33} \\ & = (s \circ (m\ f))^{-1} \circ \mathbf{R}(P) & \text{Lem. 7} \end{array}$$

Hence, we define the realizer recursively as $f = s \circ (m\ f)$, that is, non-recursively, $f = \mathbf{rec}(\lambda f(s \circ (m\ f)))$. As a result, in the final statement we can replace the right-hand side $(s \circ (m\ f))^{-1} \circ \mathbf{R}(P)$ with $f^{-1} \circ \mathbf{R}(P)$ and, thus, show that eq. (10.32) holds.

If Φ and P are Harrop then $\mu(\Phi)$ and $Q[(P \cap \mu(\Phi))/X]$ are also Harrop. As with IND' case, we need to show $\mu(\lambda X\ \mathbf{H}_X(Q)) \subseteq \mathbf{H}(P)$. By s.p. strong induction, it suffices to show

$$\mathbf{H}_X(Q)[(P \cap \mu(\Phi))/X] \subseteq \mathbf{H}(P) \tag{10.35}$$

By i.h. we have: $\mathbf{H}((\Phi(P \cap \mu(\Phi)) \subseteq P))$, which is, by Lemma 8 equivalent to

$$\mathbf{H}(Q[(P \cap \mu(\Phi))/X]) \subseteq \mathbf{H}(P) \tag{10.36}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3 we can substitute X and Y, so the former implies

$$\mathbf{H}(Mon(\Phi)[\hat{X}/X][(P \cap \mu(\Phi))/Y]) \tag{10.37}$$

Writing out $Mon(\Phi)[\hat{X}/X][(P \cap (\mu(\Phi)))/Y]$ we obtain $\hat{X} \subseteq (P \cap (\mu(\Phi))) \to Q[\hat{X}/X] \subseteq Q[(P \cap (\mu(\Phi)))/X]$. If we define $\hat{X}$ as $\mathbf{H}(P \cap (\mu(\Phi)))$, 10.37 can be rewritten as $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] \subseteq \mathbf{H}(Q[P/X])$. Since $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P \cap (\mu(\Phi)))/\hat{X}]$ is equal to $\mathbf{H}_X(Q)[\mathbf{H}(P \cap (\mu(\Phi)))/X]$, we obtain

$$\mathbf{H}_X(Q)[\mathbf{H}(P \cap \mu(\Phi))/X] \subseteq \mathbf{H}(Q[(P \cap \mu(\Phi))/X]) \tag{10.38}$$

Now, by 10.38 and 10.36 we get $\mathbf{H}_X(Q)[(P \cap \mu(\Phi))/X] \subseteq \mathbf{H}(P)$, which corresponds to our goal.

If Φ is non-Harrop and P is Harrop then $\mu(\Phi)$ and $Q[(P \cap \mu(\Phi))/X]$ are non-Harrop. As with IND' case, we need to show $\mu(\lambda\tilde{X}\ \mathbf{R}(Q)) \subseteq \Delta(\mathbf{H}(P))$. By s.p. strong induction, it suffices to show

$$\mathbf{R}(Q)[\Delta(\mathbf{H}(P \cap \mu(\Phi)))/\tilde{X}] \subseteq \Delta(\mathbf{H}(P)) \tag{10.39}$$

By i.h. we have:

$$\begin{array}{llr} & \mathbf{H}(\Phi(P \cap \mu(\Phi)) \subseteq P) & \\ \equiv & \mathbf{R}(\Phi(P \cap \mu(\Phi))) \subseteq \Delta(\mathbf{H}(P)) & \text{Lem. 8} \end{array}$$

Since $\Phi(P \cap \mu(\Phi)) = Q[(P \cap \mu(\Phi))/X]$, the above is equal to

$$\mathbf{R}(Q)[(P \cap \mu(\Phi))/X] \subseteq \Delta(\mathbf{H}(P)) \tag{10.40}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[(P \cap \mu(\Phi))/Y]) \tag{10.41}$$

Writing out $Mon(\Phi)[(P \cap \mu(\Phi))/Y]$ we obtain $X \subseteq (P \cap \mu(\Phi)) \to Q \subseteq Q[(P \cap (\mu(\Phi)))/X]$. Since P is Harrop, 10.41 can be rewritten as

$$\begin{array}{lll} & \mathbf{H}(X \subseteq (P \cap (\mu(\Phi)))) \to m\,\mathbf{r}(Q \subseteq Q[(P \cap (\mu(\Phi)))/X])) & \\ \equiv & (\mathbf{R}(X) \subseteq \Delta(\mathbf{H}(P \cap (\mu(\Phi)))) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[(P \cap \mu(\Phi))/X]) & \text{Lem. 8} \\ = & \tilde{X} \subseteq \Delta(\mathbf{H}(P \cap (\mu(\Phi)))) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[(P \cap (\mu(\Phi)))/X])) & \text{def. } \mathbf{R}(X) \end{array}$$

If we define $\tilde{X}$ as $\Delta(\mathbf{H}(P \cap (\mu(\Phi))))$ and use Lemma 3, we get

$$\begin{array}{rll} \mathbf{R}(Q)[\Delta(\mathbf{H}(P \cap \mu(\Phi)))/\tilde{X}] & \subseteq m^{-1} \circ \mathbf{R}(Q[(P \cap \mu(\Phi))/X])) & \\ & \subseteq m^{-1} \circ \Delta(\mathbf{H}(P)) & \text{by 10.40} \\ & = \Delta(\mathbf{H}(P)) & \text{Lem. 7} \end{array}$$

This corresponds to our goal.

If Φ is Harrop and P is non-Harrop then $\mu(\Phi)$ is Harrop.

We need to show $a\,\mathbf{r}\,(\mu(\Phi) \subseteq P)$, so as in case of IND', this goal is equivalent to the following:

$$\begin{array}{ll} \text{if } X \in \mathrm{FV}(Q): & \mu(\lambda X\ \mathbf{H}_X(Q)) \subseteq a^{-1} * \mathbf{R}(P) \\ \text{if } X \notin \mathrm{FV}(Q): & \mu(\lambda X\ \mathbf{H}(Q)) \subseteq a^{-1} * \mathbf{R}(P) \end{array}$$

We look at these two subcases:

(a) $X \notin \mathrm{FV}(Q)$

By s.p. strong induction it is suffices to show that $\mathbf{H}(\Phi(P \cap \mu(\Phi)) \subseteq a^{-1} * (\mathbf{R}(P))$. By i.h. we have $a\,\mathbf{r}\,((\Phi(P \cap \mu(\Phi))) \subseteq P)$. By Lemma 8 the latter is equivalent to $\mathbf{H}(\Phi(P \cap \mu(\Phi)) \subseteq a^{-1} * (\mathbf{R}(P))$, which corresponds to our goal.

(b) $X \in \mathrm{FV}(Q)$

By s.p. strong induction, it is enough to show

$$\mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P \cap \mu(\Phi))/X] \subseteq a^{-1} * (\mathbf{R}(P)) \tag{10.42}$$

By i.h. we have: $s\,\mathbf{r}\,(\Phi(P \cap \mu(\Phi)) \subseteq P)$, which is, by Lemma 8 equivalent to

$$\mathbf{R}(Q[(P \cap \mu(\Phi))/X]) \subseteq s^{-1} \circ \mathbf{R}(P) \tag{10.43}$$

By i.h. we have $m\,\mathbf{r}\,(Mon(\Phi))$ and using Lemma 3, followed by Lemma 4(a), we obtain

$$m\,\mathbf{r}\,(Mon(\Phi)[\hat{X}/X][(P \cap \mu(\Phi))/Y]) \tag{10.44}$$

Writing out $Mon(\Phi)[\hat{X}/X][(P \cap \mu(\Phi))/Y]$ we obtain $\hat{X} \subseteq (P \cap \mu(\Phi)) \to Q[\hat{X}/X] \subseteq Q[(P \cap \mu(\Phi))/X]$. Therefore, 10.44 can be rewritten by Lem. 8 and with respect to the definition of $\mathbf{H}(\hat{X})$ as follows

$$\begin{array}{rl} & \forall b(a\,\mathbf{r}\,(\hat{X} \subseteq (P \cap \mu(\Phi))) \to (m\ b)\,\mathbf{r}\,(Q[\hat{X}/X] \subseteq Q[(P \cap \mu(\Phi))/X])) \\ \equiv & \forall b(\mathbf{H}(\hat{X}) \subseteq b^{-1} * \mathbf{R}(P \cap \mu(\Phi)) \to \mathbf{H}_X(Q)[\hat{X}/X] \subseteq (m\ b)^{-1} * \mathbf{R}(Q[(P \cap \mu(\Phi))/X])) \\ = & \forall b(\hat{X} \subseteq b^{-1} * \mathbf{R}(P \cap \mu(\Phi)) \to \mathbf{H}_X(Q)[\hat{X}/X] \subseteq (m\ b)^{-1} * \mathbf{R}(Q[(P \cap \mu(\Phi))/X])) \end{array}$$

If we define b as a and $\hat{X}$ as $a^{-1} * \mathbf{R}(P \cap \mu(\Phi))$ and since $\mathbf{H}(Q[\hat{X}/X])[a^{-1} * \mathbf{R}(P \cap \mu(\Phi))/\hat{X}] = \mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P \cap \mu(\Phi))/X]$, we obtain

$$
\begin{aligned}
\mathbf{H}_X(Q)[a^{-1} * \mathbf{R}(P \cap \mu(\Phi))/X] &\subseteq (m\ a)^{-1} * \mathbf{R}(Q[(P \cap \mu(\Phi))/X]) \\
&\subseteq (m\ a)^{-1} * (s^{-1} \circ \mathbf{R}(P)) && \text{by 10.43} \\
&= (s(m\ a))^{-1} * \mathbf{R}(P) && \text{Lem. 7}
\end{aligned}
$$

Hence, we define the realizer recursively as $a = s(m\ a)$. As a result, in the final statement we can replace the right-hand side $(s(m\ a))^{-1} * \mathbf{R}(P)$ with $a^{-1} * \mathbf{R}(P)$ and, thus, show that eq. (10.42) holds.

COIND'. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} (P \subseteq \nu(\Phi))$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} P \subseteq \Phi(P)$ and $Mon(\Phi)$ by COIND', where $\Phi(P) = Q[P/X]$. Here $Mon(\Phi)$ means $X \subseteq Y \to Q \subseteq Q[Y/X]$. We look at specific cases, depending on whether Φ and P are Harrop.

If Φ and P are non-Harrop:

Show:

$$
\begin{aligned}
& f\,\mathbf{r}\,(P \subseteq \nu(\Phi)) \\
\equiv\quad & f \circ \mathbf{R}(P) \subseteq \mathbf{R}(\nu(\Phi)) && \text{Lem. 8} \\
=\quad & f \circ \mathbf{R}(P) \subseteq \mathbf{R}(\nu(\lambda X\ Q)) && \text{since } \Phi = \lambda X\ Q \\
=\quad & f \circ \mathbf{R}(P) \subseteq \nu(\lambda \tilde{X} \mathbf{R}(Q)) && \text{since } \mathbf{R}(\nu(\Phi)) = \nu(\mathbf{R}(\Phi)) \\
& && \text{and } \mathbf{R}(\lambda X\ Q) = \lambda \tilde{X}(\mathbf{R}(Q))
\end{aligned}
$$

By s.p. coinduction, it is enough to show

$$
f \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q)[f \circ \mathbf{R}(P)/\tilde{X}] \tag{10.45}
$$

By i.h. we have: $s\,\mathbf{r}\,(P \subseteq \Phi(P))$, which is, by Lemma 8 equivalent to

$$
s \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q[P/X]) \tag{10.46}
$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$
m\,\mathbf{r}\,(Mon(\Phi)[P/X]) \tag{10.47}
$$

Writing out $Mon(\Phi)[P/X]$ we obtain $P \subseteq Y \to Q[P/X] \subseteq Q[Y/X]$. Therefore, 10.47 can be rewritten as

$$
\begin{aligned}
& \forall g(g\,\mathbf{r}\,(P \subseteq Y) \to (m\ g)\,\mathbf{r}\,(Q[P/X] \subseteq Q[Y/X]) \\
\equiv\quad & \forall g(g \circ \mathbf{R}(P) \subseteq \mathbf{R}(Y) \to (m\ g) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) && \text{Lem. 8} \\
=\quad & \forall g(g \circ \mathbf{R}(P) \subseteq \tilde{Y} \to (m\ g) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) && \text{def. } \mathbf{R}(Y)
\end{aligned}
$$

If we define g as f and $\tilde{Y}$ as $f \circ \mathbf{R}(P)$ and use Lemma 3, we get

$$\begin{aligned}
\mathbf{R}(Q)[f \circ \mathbf{R}(P)/\tilde{X}] &\supseteq (m\ f) \circ \mathbf{R}(Q[P/X])) & \\
&\supseteq (m\ f) \circ (s\ \circ \mathbf{R}(P)) & \text{by 10.46} \\
&= ((m\ f) \circ\ s) \circ \mathbf{R}(P) & \text{Lem. 7}
\end{aligned}$$

Hence, we define the realizer recursively as $f = (m\ f) \circ\ s$, that is, non-recursively, $f = \mathbf{rec}(\lambda f((m\ f) \circ\ s))$. As a result, in the final statement we can replace the right-hand side $((m\ f) \circ\ s) \circ \mathbf{R}(P)$ with $f \circ \mathbf{R}(P)$ and, thus, show that eq. (10.45) holds.

If Φ and P are Harrop then $\nu(\Phi)$ and $Q[P/X]$ are also Harrop.

Show:

$$\begin{aligned}
& \mathbf{H}(P \subseteq \nu(\Phi)) & \\
\equiv\ & \mathbf{H}(P) \subseteq \mathbf{H}(\nu(\Phi)) & \text{Lem. 8} \\
=\ & \mathbf{H}(P) \subseteq \mathbf{H}(\nu\lambda X\ Q) & \text{since } \Phi = \lambda X\ Q \\
=\ & \mathbf{H}(P) \subseteq \nu(\lambda X\ \mathbf{H}_X(Q)) & \text{since } \mathbf{H}(\nu(\Phi)) = \nu\mathbf{H}(\Phi) \\
& & \text{and } \mathbf{H}(\lambda X\ Q) = \lambda X\ \mathbf{H}_X(Q)
\end{aligned}$$

By s.p. coinduction, it is enough to show

$$\mathbf{H}(P) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P)/X] \tag{10.48}$$

By i.h. we have: $\mathbf{H}(P \subseteq \Phi(P))$, which is, by Lemma 8 equivalent to

$$\mathbf{H}(P) \subseteq \mathbf{H}(Q[P/X]) \tag{10.49}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3 we can substitute X and Y the former implies

$$\mathbf{H}(Mon(\Phi)[P/X][\hat{X}/Y]) \tag{10.50}$$

Writing out $Mon(\Phi)[P/X][\hat{X}/Y]$ we obtain $P \subseteq \hat{X} \to Q[P/X] \subseteq Q[\hat{X}/X]$. If we define $\hat{X}$ as $\mathbf{H}(P)$, 10.50 can be rewritten as $\mathbf{H}(Q[P/X]) \subseteq \mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$, which is the same as $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] = \mathbf{H}_X(Q)[\mathbf{H}(P)/X]$, so we obtain

$$\mathbf{H}(Q[P/X]) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P)/X] \tag{10.51}$$

By 10.51 and 10.49 we get $\mathbf{H}(P) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P)/X]$, which corresponds to our goal.

If Φ is non-Harrop and P is Harrop then $\nu(\Phi)$ and $Q[P/X]$ are non-Harrop.

Show:

$$
\begin{array}{lll}
 & a\,\mathbf{r}\,(P \subseteq \nu(\Phi)) & \\
\equiv & a * \mathbf{H}(P) \subseteq \mathbf{R}(\nu(\Phi)) & \text{Lem. 8} \\
= & a * \mathbf{H}(P) \subseteq \mathbf{R}(\nu(\lambda X\ Q)) & \text{since } \Phi = \lambda X\ Q \\
= & a * \mathbf{H}(P) \subseteq \nu(\lambda X\ \mathbf{R}(Q))) & \text{since } \mathbf{R}(\nu(\Phi)) = \nu\mathbf{R}(\Phi) \\
 & & \text{and } \mathbf{R}(\lambda X\ Q) = \lambda\tilde{X}\ \mathbf{R}(Q)
\end{array}
$$

By s.p. coinduction, it is suffices to show

$$a * \mathbf{H}(P) \subseteq \mathbf{R}(Q)[a * \mathbf{H}(P)/\tilde{X}] \tag{10.52}$$

By i.h. we have: $s\,\mathbf{r}\,(P \subseteq \Phi(P)) \subseteq (P))$, which is, by Lemma 8 equivalent to

$$s * \mathbf{H}(P) \subseteq \mathbf{R}(Q[P/X]) \tag{10.53}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P/X]) \tag{10.54}$$

Writing out $Mon(\Phi)[P/X]$ we obtain $P \subseteq Y \to Q[P/X] \subseteq Q[Y/X]$. Therefore, 10.54 can be rewritten as

$$
\begin{array}{lll}
 & \forall g(g\,\mathbf{r}\,(P \subseteq Y) \to (m\ g)\,\mathbf{r}\,(Q[P/X] \subseteq Q[Y/X]) & \\
\equiv & \forall g(g\ * \mathbf{H}(P) \subseteq \mathbf{R}(Y) \to (m\ g) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) & \text{Lem. 8} \\
= & \forall g(g\ * \mathbf{H}(P) \subseteq \tilde{Y} \to (m\ g) \circ\ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) & \text{def. } \mathbf{R}(Y)
\end{array}
$$

If we define g as a and $\tilde{Y}$ as $a\ * \mathbf{H}(P)$, we obtain

$$
\begin{array}{rll}
\mathbf{R}(Q)[a\ * \mathbf{H}(P)/\tilde{X}] & \supseteq (m\ a) \circ\ \mathbf{R}(Q[P/X]) & \\
 & \supseteq (m\ a) \circ\ (s * \mathbf{H}(P)) & \text{by 10.53} \\
 & = ((m\ a)\ s) * \mathbf{H}(P) & \text{Lem. 7}
\end{array}
$$

Hence, we define the realizer recursively as $a = (m\ a)\ s$. As a result, in the final statement we can replace the right-hand side $((m\ a)\ s) * \mathbf{R}(P)$ with $a * \mathbf{R}(P)$ and, thus, show that eq. (10.52) holds.

If Φ is Harrop and P is non-Harrop then $\nu(\Phi)$ is Harrop.

Show:

$$\begin{array}{rll} & \mathbf{H}(P \subseteq \nu(\Phi)) & \\ \equiv & \exists(\mathbf{R}(P)) \subseteq \mathbf{H}(\nu(\Phi)) & \text{Lem. 8} \\ \text{if } X \in \mathrm{FV}(Q) = & \exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\ \mathbf{H}_X(Q)) & \text{since } \mathbf{H}(\nu(\Phi)) = \nu\mathbf{H}(\Phi) \\ & & \text{and } \mathbf{H}(\lambda X\ Q) = \lambda X\ \mathbf{H}_X(Q) \\ \text{if } X \notin \mathrm{FV}(Q) = & \exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\ \mathbf{H}(Q)) & \text{as above but } \mathbf{H}_X = \mathbf{H} \end{array}$$

We look at these two subcases:

(a) $X \notin \mathrm{FV}(Q)$

By s.p. coinduction it is suffices to show that $\exists(\mathbf{R}(P)) \subseteq \mathbf{H}(Q)$. This corresponds to i.h.: $\mathbf{H}(P \subseteq Q)$, which by Lemma 8 is equal to $\exists(\mathbf{R}(P)) \subseteq \mathbf{H}(Q)$.

(b) $X \in \mathrm{FV}(Q)$. In this case Q, $Q[P/X]$ and $Mon(\Phi)[P/X][\hat{X}/Y]$ are non-Harrop.

By s.p. coinduction, it suffices to show

$$\exists(\mathbf{R}(P)) \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] \tag{10.55}$$

By i.h. we have: $s\,\mathbf{r}\,(P \subseteq \Phi(P))$, which is, by Lemma 8 equivalent to

$$s \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q[P/X]) \tag{10.56}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3, followed by Lemma 4(a), we obtain

$$\mathbf{H}(Mon(\Phi)[P/X][\hat{X}/Y]) \tag{10.57}$$

Writing out $Mon(\Phi)[P/X][\hat{X}/Y]$ we obtain $P \subseteq \hat{X} \to Q[P/X] \subseteq Q[\hat{X}/X]$. Therefore, 10.57 can be rewritten as $\exists(\mathbf{R}(P)) \subseteq \hat{X} \to \exists(\mathbf{R}(Q[P/X])) \subseteq \mathbf{H}(Q[\hat{X}/X])$. If we define $\hat{X} = \exists(\mathbf{R}(P))$ and use Lemma 3, we obtain

$$\begin{array}{rl} \exists(\mathbf{R}(Q[P/X])) \subseteq \mathbf{H}(Q[\hat{X}/X])[\exists(\mathbf{R}(P))/\hat{X}] & \\ \exists(\mathbf{R}(Q[P/X])) = \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] & \text{subst.} \\ \exists(s \circ \mathbf{R}(P)) \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] & \text{by 10.56} \\ \exists(\mathbf{R}(P)) = \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] & \text{Lem. 7} \end{array}$$

This corresponds to our goal.

HSC'. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} (P \subseteq \nu(\Phi))$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} (P \subseteq \Phi(P) \cup \nu(\Phi))$ and $Mon(\Phi)$ by HSC', where $P \subseteq Q[P/X] \cup \nu(\Phi)$. Here $Mon(\Phi)$ means $X \subseteq Y \to Q \subseteq Q[Y/X]$. We look at specific cases, depending on whether Φ and P are Harrop.

If Φ and P are non-Harrop then, as with COIND' case, we need to show $f\,\mathbf{r}\,(P \subseteq \Phi(P) \cup \nu(\Phi))$, which can be rewritten as $f \circ \mathbf{R}(P) \subseteq \nu(\lambda \tilde{X} \mathbf{R}(Q))$. By s.p. half-strong coinduction, it is enough to show

$$f \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q)[f \circ \mathbf{R}(P)/\tilde{X}] \cup \nu(\mathbf{R}(\Phi)) \tag{10.58}$$

By i.h. we have:

$$\begin{array}{lll} & s\,\mathbf{r}\,(\mathbf{R}(P) \subseteq \Phi(P) \cup \nu(\Phi)) & \\ \equiv & s \circ \mathbf{R}(P) \subseteq \mathbf{R}(\Phi(P) \cup \nu(\Phi)) & \text{Lem. 8} \\ \equiv & s \circ \mathbf{R}(P) \subseteq (\mathbf{Lt} \circ \mathbf{R}(\Phi(P)) \cup \mathbf{Rt} \circ \mathbf{R}(\nu(\Phi))) & \text{Lem. 8} \end{array}$$

Since $\Phi(P) = Q[P/X]$ and by definition of $\mathbf{R}(\mu(\Phi))$, the above is equal to

$$s \circ \mathbf{R}(P) \subseteq (\mathbf{Lt} \circ \mathbf{R}(Q[P/X]) \cup \mathbf{Rt} \circ \nu(\mathbf{R}(\Phi))) \tag{10.59}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P/X]) \tag{10.60}$$

Writing out $Mon(\Phi)[P/X]$ we obtain $P \subseteq Y \to Q[P/X] \subseteq Q[Y/X]$. Therefore, 10.60 can be rewritten as

$$\begin{array}{lll} & \forall g(g\,\mathbf{r}\,(P \subseteq Y) \to (m\ g)\,\mathbf{r}\,(Q[P/X] \subseteq Q[Y/X])) & \\ \equiv & \forall g(g \circ \mathbf{R}(P) \subseteq \mathbf{R}(Y) \to (m\ g) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) & \text{Lem. 8} \\ = & \forall g(g \circ \mathbf{R}(P) \subseteq \tilde{Y} \to (m\ g) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) & \text{def. } \mathbf{R}(X) \end{array}$$

If we define g as f and $\tilde{Y}$ as $f \circ \mathbf{R}(P)$ and use Lemma 3, we get

$$(m\ f) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q)[f \circ \mathbf{R}(P)/\tilde{X}] \tag{10.61}$$

Now,

$$\begin{array}{ll} \mathbf{R}(Q)[f \circ \mathbf{R}(P)/\tilde{X}] \cup \nu(\mathbf{R}(\Phi)) & \\ \supseteq (m\ f) \circ \mathbf{R}(Q[P/X]) \cup \nu(\mathbf{R}(\Phi)) & \\ = [(m\ f) + \mathbf{id}] \circ (\mathbf{Lt} \circ \mathbf{R}(Q[P/X]) \cup \mathbf{Rt} \circ \mathbf{R}(\nu(\Phi))) & \text{Lem. 7} \\ = [(m\ f) + \mathbf{id}] \circ (\mathbf{Lt} \circ \mathbf{R}(Q[P/X]) \cup \mathbf{Rt} \circ \nu(\mathbf{R}(\Phi))) & \text{by def. } \mathbf{R}(\mu(\Phi)) \\ \supseteq [(m\ f) + \mathbf{id}] \circ (s \circ \mathbf{R}(P)) & \text{by 10.59} \\ = ([(m\ f) + \mathbf{id}] \circ s) \circ \mathbf{R}(P) & \text{Lem. 7} \end{array}$$

Hence, we define the realizer recursively as $f = [(m\ f) + \mathbf{id}] \circ s$, that is, non-recursively, $f = \mathbf{rec}(\lambda f([(m\ f) + \mathbf{id}] \circ s))$. As a result, in the final statement we can replace the right-hand side $([(m\ f) + \mathbf{id}] \circ s) \circ \mathbf{R}(P)$ with $f \circ \mathbf{R}(P)$ and, thus, show that eq. (10.58) holds.

If both Φ and P are Harrop then we need to show $\mathbf{H}(P) \subseteq \nu(\lambda X\ \mathbf{H}_X(Q))$. By s.p. half-strong coinduction, it suffices to show

$$\mathbf{H}(P) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P)/X] \cup \mathbf{H}(\nu(\Phi)) \tag{10.62}$$

By i.h. we have: $\mathbf{H}(P \subseteq \Phi(P) \cup \nu(\Phi))$, which is, by Lemma 8 equivalent to

$$\mathbf{H}(P) \subseteq \mathbf{H}(Q[P/X]) \cup \mathbf{H}(\nu(\Phi)) \tag{10.63}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3 we can substitute X and Y the former implies

$$\mathbf{H}(Mon(\Phi)[P/X][\hat{X}/Y]) \tag{10.64}$$

Writing out $Mon(\Phi)[P/X][\hat{X}/Y]$ we obtain $P \subseteq \hat{X} \to Q[P/X] \subseteq Q[\hat{X}/X]$. If we define $\hat{X}$ as $\mathbf{H}(P)$, 10.64 can be rewritten as $\mathbf{H}(Q[P/X]) \subseteq \mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$, which is the same as $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] = \mathbf{H}_X(Q)[\mathbf{H}(P)/X]$, so we obtain

$$\mathbf{H}(Q[P/X]) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P)/X] \tag{10.65}$$

Now, by 10.65 and 10.63 $\mathbf{H}(Q[P/X]) \cup \mathbf{H}(\nu(\Phi)) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P)/X] \cup \mathbf{H}(\nu(\Phi))$ and, therefore, we get $\mathbf{H}(P) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P)/X] \cup \mathbf{H}(\nu(\Phi))$, which corresponds to our goal.

If Φ is non-Harrop and P is Harrop then (Φ) and $Q[P/X]$ are non-Harrop. As with COIND' case, we need to show $a\,\mathbf{r}\,(P \subseteq \nu(\Phi))$, i.e. $a * \mathbf{H}(P) \subseteq \nu(\lambda X\ \mathbf{R}(Q)))$. By s.p. half-strong coinduction, it suffices to show

$$a * \mathbf{H}(P) \subseteq \mathbf{R}(Q)[a * \mathbf{H}(P)/\tilde{X}] \cup \mathbf{R}(\nu(\Phi)) \tag{10.66}$$

By i.h. we have: $s\,\mathbf{r}\,(P \subseteq \Phi(P) \cup \nu(\Phi))$, which is, by Lemma 8 equivalent to

$$s * \mathbf{H}(P) \subseteq (\mathbf{Lt} \circ \mathbf{R}(Q[P/X]) \cup \mathbf{Rt} \circ \mathbf{R}(\nu(\Phi))) \tag{10.67}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P/X]) \tag{10.68}$$

Writing out $Mon(\Phi)[P/X]$ we obtain $P \subseteq Y \to Q[P/X] \subseteq Q[Y/X]$. Therefore, 10.68 can be rewritten as

$$
\begin{array}{lll}
 & \forall g(g\,\mathbf{r}\,(P \subseteq Y) \to (m\ g)\,\mathbf{r}\,(Q[P/X] \subseteq Q[Y/X]) & \\
\equiv & \forall g(g\ *\mathbf{H}(P) \subseteq \mathbf{R}(Y) \to (m\ g) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) & \text{Lem. 8} \\
= & \forall g(g\ *\mathbf{H}(P) \subseteq \tilde{Y} \to (m\ g) \circ\ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X])) & \text{def. } \mathbf{R}(Y)
\end{array}
$$

If we define g as a and $\tilde{Y}$ as $a\ *\mathbf{H}(P)$, we obtain

$$
\begin{aligned}
\mathbf{R}(Q)[a\ *\mathbf{H}(P)/\tilde{X}] \cup \mathbf{R}(\nu(\Phi)) &\supseteq ((m\ a) \circ\ \mathbf{R}(Q[P/X])) \cup \mathbf{R}(\nu(\Phi)) \\
(\text{by Lem. 7}) &= [(m\ a) + \mathbf{id}] \circ (\mathbf{Lt} \circ \mathbf{R}(Q[P/X]) \cup \mathbf{Rt} \circ \mathbf{R}(\nu(\Phi))) \\
(\text{by 10.67}) &\supseteq [(m\ a) + \mathbf{id}] \circ (s * \mathbf{H}(P)) \\
(\text{by Lem. 7}) &= ([(m\ a) + \mathbf{id}]\ s) * \mathbf{H}(P)
\end{aligned}
$$

Hence, we define the realizer recursively as $a = ([(m\ a) + \mathbf{id}]\ s)$. As a result, in the final statement we can replace the right-hand side $([(m\ a) + \mathbf{id}]\ s) * \mathbf{R}(P)$ with $a * \mathbf{R}(P)$ and, thus, show that eq. (10.66) holds.

If Φ is Harrop and P is non-Harrop then Φ is Harrop.

We need to show $\mathbf{H}(P \subseteq \nu(\Phi))$, so as in case of COIND', this goal is equivalent to the following:

$$
\begin{array}{ll}
\text{if } X \in \mathrm{FV}(Q) = & \exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\ \mathbf{H}_X(Q)) \\
\text{if } X \notin \mathrm{FV}(Q) = & \exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\ \mathbf{H}(Q))
\end{array}
$$

We look at these two subcases:

(a) $X \notin \mathrm{FV}(Q)$

By s.p. half-strong coinduction it is suffices to show that $\exists(\mathbf{R}(P)) \subseteq \mathbf{H}(Q \cup \nu(\Phi))$. This corresponds to i.h.: $\mathbf{H}(P \subseteq Q \cup (\nu(\Phi))$ or, by Lemma 8, $\exists(\mathbf{R}(P)) \subseteq \mathbf{H}(Q \cup \nu(\Phi))$.

(b) $X \in \mathrm{FV}(Q)$. In this case Q, $Q[P/X]$ and $Mon(\Phi)[P/X][\hat{X}/Y]$ are non-Harrop.

By s.p. half-strong coinduction, it suffices to show

$$\exists(\mathbf{R}(P)) \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] \cup \mathbf{H}(\nu(\Phi))) \tag{10.69}$$

By i.h. we have: $s\,\mathbf{r}\,(P \subseteq \Phi(P) \cup (\nu(\Phi)))$, which is, by Lemma 8 equivalent to

$$s \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q[P/X] \cup \nu(\Phi)) \tag{10.70}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3, followed by Lemma 4(a), we obtain

$$\mathbf{H}(Mon(\Phi)[P/X][\hat{X}/Y]) \tag{10.71}$$

Writing out $Mon(\Phi)[P/X][\hat{X}/Y]$ we obtain $P \subseteq \hat{X} \to Q[P/X] \subseteq Q[\hat{X}/X]$. Therefore, 10.71 can be rewritten as $\exists(\mathbf{R}(P)) \subseteq \hat{X} \to \exists(\mathbf{R}(Q[P/X])) \subseteq \mathbf{H}(Q[\hat{X}/X])$. If we define $\hat{X}$ as $\exists(\mathbf{R}(P))$ and use Lemma 3, we obtain

$$\begin{array}{lll} & \exists(\mathbf{R}(Q[P/X])) \subseteq \mathbf{H}(Q[\hat{X}/X])[\exists(\mathbf{R}(P))/\hat{X}] & \\ = & \exists(\mathbf{R}(Q[P/X])) \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] & \text{subst.} \end{array}$$

Now,

$$\begin{array}{rll} \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] \cup \mathbf{H}(\nu(\Phi)) & \supseteq (\exists(\mathbf{R}(Q[P/X]))) \cup \mathbf{H}(\nu(\Phi)) & \\ & = \exists(\mathbf{R}(Q[P/X] \cup \nu(\Phi))) & \text{Lem. 7} \\ & \supseteq \exists(s \circ \mathbf{R}(P)) & \text{by 10.70} \\ & = \exists(\mathbf{R}(P)) & \text{Lem. 7} \end{array}$$

This corresponds to our goal.

SC'. Assume $\Delta, \Gamma \vdash_{\text{IFP'}} (P \subseteq \nu(\Phi))$ has been derived from $\Delta, \Gamma \vdash_{\text{IFP'}} (P \subseteq \Phi(P \cup \nu(\Phi)))$ and $Mon(\Phi)$ by SC', where $\Phi(P \cup \nu(\Phi)) = Q[P \cup \nu(\Phi)/X]$. Here $Mon(\Phi)$ means $X \subseteq Y \to Q \subseteq Q[Y/X]$. We look at specific cases, depending on whether Φ and P are Harrop.

If Φ and P are non-Harrop then, as with COIND' case, we need to show $f \circ \mathbf{R}(P) \subseteq \nu(\lambda\tilde{X}\mathbf{R}(Q))$. By s.p. strong coinduction, it is enough to show

$$f \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q)[f \circ (P \cup \nu(\Phi))/\tilde{X}] \tag{10.72}$$

By i.h. we have:

$$\begin{array}{lll} & s\,\mathbf{r}\,(\mathbf{R}(P) \subseteq \Phi(P \cup \nu(\Phi))) & \\ \equiv & s \circ \mathbf{R}(P) \subseteq \mathbf{R}(\Phi(P \cup \nu(\Phi))) & \text{Lem. 8} \end{array}$$

Since $\Phi(P \cup \nu(\Phi)) = Q[(P \cup \nu(\Phi))/X]$:

$$s \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q[P \cup \nu(\Phi)/X]) \tag{10.73}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[(P \cup \nu(\Phi))/X]) \tag{10.74}$$

Writing out $Mon(\Phi)[(P \cup \nu(\Phi))/X])$ we obtain $(P \cup \nu(\Phi)) \subseteq Y \to Q[P \cup \nu(\Phi)/X] \subseteq Q[Y/X]$. Therefore, 10.74 can be rewritten, using Lem. 8, the definitions of $\mathbf{R}(Y)$ and $\mathbf{R}(\lambda X\ Q)$, as follows:

$$\begin{aligned}
& \forall g(g\,\mathbf{r}\,((P \cup \nu(\Phi)) \subseteq Y) \to (m\ g)\,\mathbf{r}\,(Q[(P \cup \nu(\Phi))/X] \subseteq Q[Y/X]) \\
\equiv\ & \forall g(g \circ \mathbf{R}(P \cup \nu(\Phi)) \subseteq \mathbf{R}(Y) \to (m\ g) \circ \mathbf{R}(Q[(P \cup \nu(\Phi))/X]) \subseteq \mathbf{R}(Q[Y/X])) \\
=\ & \forall g(g \circ \mathbf{R}(P \cup \nu(\Phi)) \subseteq \tilde{Y} \to (m\ g) \circ \mathbf{R}(Q[(P \cup \nu(\Phi))/X]) \subseteq \mathbf{R}(Q[Y/X])) \\
=\ & \forall g(g \circ \mathbf{R}(P \cup \nu(\Phi)) \subseteq \tilde{Y} \to (m\ g) \circ \mathbf{R}(Q[(P \cup \nu(\Phi))/X]) \subseteq \mathbf{R}(Q)[\tilde{Y}/\tilde{X}])
\end{aligned}$$

If we define g as f and $\tilde{Y}$ as $f \circ \mathbf{R}(P \cup \nu(\Phi))$ and use Lemma 3, we get

$$\begin{aligned}
\mathbf{R}(Q)[f \circ \mathbf{R}(P \cup \nu(\Phi))/\tilde{X}] &\supseteq (m\ f) \circ \mathbf{R}(Q[(P \cup \nu(\Phi))/X])) \\
&\supseteq (m\ f) \circ (s \circ \mathbf{R}(P)) && \text{by 10.73} \\
&= ((m\ f) \circ s) \circ \mathbf{R}(P) && \text{Lem. 7}
\end{aligned}$$

Hence, we define the realizer recursively as $f = (m\ f) \circ s$, that is, non-recursively, $f = \mathbf{rec}(\lambda f((m\ f) \circ s))$. As a result, in the final statement we can replace the right-hand side $((m\ f) \circ s) \circ \mathbf{R}(P)$ with $f \circ \mathbf{R}(P)$ and, thus, show that eq. (10.72) holds.

If Φ and P are Harrop then $\nu(\Phi)$ and $Q[P \cup \nu(\Phi)/X]$ are also Harrop. As in COIND' case, we need to show $\mathbf{H}(P) \subseteq \nu(\lambda X\ \mathbf{H}_X(Q))$. By s.p. strong coinduction, it suffices to show

$$\mathbf{H}(P) \subseteq \mathbf{H}_X(Q)[\mathbf{H}(P \cup \nu(\Phi))/X] \tag{10.75}$$

By i.h. we have: $\mathbf{H}(P \subseteq \Phi(P \cup \nu(\Phi)))$, which is, by Lemma 8 equivalent to

$$\mathbf{H}(P) \subseteq \mathbf{H}(Q[(P \cup \nu(\Phi))/X]) \tag{10.76}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3 we can substitute X and Y the former implies

$$\mathbf{H}(Mon(\Phi)[(P \cup \nu(\Phi))/X][\hat{X}/Y]) \tag{10.77}$$

Writing out $Mon(\Phi)[(P \cup \nu(\Phi))/X][\hat{X}/Y]$ we obtain $(P \cup \nu(\Phi)) \subseteq \hat{X} \to Q[(P \cup \nu(\Phi))/X] \subseteq Q[\hat{X}/X]$.

If we define $\hat{X}$ as $\mathbf{H}(P \cup \nu(\Phi))$, 10.77 can be rewritten as

$$\begin{aligned}
\mathbf{H}(Q[(P \cup \nu(\Phi))/X]) &\subseteq \mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P \cup \nu(\Phi))/\hat{X}] \\
\mathbf{H}(Q[(P \cup \nu(\Phi))/X]) &\subseteq \mathbf{H}_X(Q)[\mathbf{H}(P \cup \nu(\Phi))/X] && \text{subst.} \\
\mathbf{H}(P) &\subseteq \mathbf{H}_X(Q)[\mathbf{H}(P \cup \nu(\Phi))/X] && \text{by 10.76}
\end{aligned}$$

This corresponds to our goal.

If Φ is non-Harrop and P is Harrop then $\nu(\Phi)$ and $Q[(P \cup \nu(\Phi))/X]$ are non-Harrop. As in the regular COIND' case, we need to show $a * \mathbf{H}(P) \subseteq \nu(\lambda X\ \mathbf{R}(Q)))$. By s.p. coinduction, it is suffices to show

$$a * \mathbf{H}(P) \subseteq \mathbf{R}(Q[P \cup \nu(\Phi)/X]) \tag{10.78}$$

By i.h. we have: $s\,\mathbf{r}\,(P \subseteq \Phi(P \cup \nu(\Phi)) \subseteq P)$, which is, by Lemma 8 equivalent to

$$s * \mathbf{H}(P) \subseteq \mathbf{R}(Q[P \cup \nu(\Phi)/X]) \tag{10.79}$$

By i.h. we have $m\,\mathbf{r}\,Mon(\Phi)$ and by Lemma 4(a) this implies

$$m\,\mathbf{r}\,(Mon(\Phi)[P \cup \nu(\Phi)/X]) \tag{10.80}$$

Writing out $Mon(\Phi)[P \cup \nu(\Phi)/X]$ we obtain $P \cup \nu(\Phi) \subseteq Y \to Q[P \cup \nu(\Phi)/X] \subseteq Q[Y/X]$. Therefore, 10.80 can be rewritten, using Lem. 8, the definition of $\mathbf{R}(Y)$ and $\mathbf{R}(\lambda X\ Q)$, as follows:

$$\begin{array}{rl}
 & \forall g(g\,\mathbf{r}\,(P \cup \nu(\Phi) \subseteq Y) \to (m\ g)\,\mathbf{r}\,(Q[P \cup \nu(\Phi)/X] \subseteq Q[Y/X]) \\
\equiv & \forall g(g * \mathbf{R}(P \cup \nu(\Phi)) \subseteq \mathbf{R}(Y) \to (m\ g) \circ \mathbf{R}(Q[P \cup \nu(\Phi)/X]) \subseteq \mathbf{R}(Q[Y/X])) \\
= & \forall g(g * \mathbf{R}(P \cup \nu(\Phi)) \subseteq \tilde{Y} \to (m\ g) \circ \mathbf{R}(Q[P \cup \nu(\Phi)/X]) \subseteq \mathbf{R}(Q[Y/X])) \\
= & \forall g(g * \mathbf{R}(P \cup \nu(\Phi)) \subseteq \tilde{Y} \to (m\ g) \circ \mathbf{R}(Q[P \cup \nu(\Phi)/X]) \subseteq \mathbf{R}(Q)[\tilde{Y}/\tilde{X}])
\end{array}$$

If we define g as $[\mathbf{id} + a]$ (since P is Harrop but $\nu(\Phi)$ is not) and $\tilde{Y} = [\mathbf{id} + a]\ \mathbf{R}(P \cup \nu(\Phi))$, we obtain

$$\begin{array}{rlr}
\mathbf{R}(Q)[[\mathbf{id}+a] * \mathbf{R}(P \cup \nu(\Phi))/\tilde{X}] & \supseteq (m\ [\mathbf{id}+a]) \circ \mathbf{R}(Q[(P \cup \nu(\Phi))/X]) & \\
 & \supseteq (m\ [\mathbf{id}+a]) \circ (s * \mathbf{H}(P)) & \text{by 10.79} \\
 & = ((m\ [\mathbf{id}+a])\ s) * \mathbf{H}(P) & \text{Lem. 7}
\end{array}$$

Hence, we define the realizer recursively as $a = ((m\ [\mathbf{id}+a])\ s)$. As a result, in the final statement we can replace the right-hand side $((m\ [\mathbf{id}+a])\ s) * \mathbf{R}(P)$ with $a * \mathbf{R}(P)$ and, thus, show that eq. (10.78) holds.

If Φ is Harrop and P is non-Harrop then $\nu(\Phi)$ is Harrop.

We need to show $\mathbf{H}(P \subseteq \nu(\Phi))$, so as in case of COIND', the goals are the following:

$$\begin{array}{ll}
\text{if } X \in \mathrm{FV}(Q): & \exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\ \mathbf{H}_X(Q)) \\
\text{if } X \notin \mathrm{FV}(Q): & \exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\ \mathbf{H}(Q))
\end{array}$$

We look at these two subcases:

(a) $X \notin \mathrm{FV}(Q)$

By s.p. strong coinduction it is suffices to show that $\exists(\mathbf{R}(P)) \subseteq \mathbf{H}(Q)$. This corresponds to i.h.: $\mathbf{H}(\exists(\mathbf{R}(P)) \subseteq \Phi(\exists(\mathbf{R}(P)) \cup \nu(\Phi)))$, which is $\exists(\mathbf{R}(P)) \subseteq \mathbf{H}(Q)$ by Lemma 8.

(b) $X \in \mathrm{FV}(Q)$. In this case Q, $Q[(P \cup \nu(\Phi))/X]$ and $Mon(\Phi)[(P \cup \nu(\Phi))/X][\hat{X}/Y]$ are non-Harrop.

By s.p. strong coinduction, it suffices to show

$$\exists(\mathbf{R}(P)) \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P) \cup \nu(\Phi)/X] \tag{10.81}$$

By i.h. we have: $s\,\mathbf{r}\,(P \subseteq \Phi(P \cup \nu(\Phi)))$, which is, by Lemma 8 equivalent to

$$s \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q[(P \cup \nu(\Phi))/X]) \tag{10.82}$$

By i.h. we have $\mathbf{H}(Mon(\Phi))$ and using Lemma 3, followed by Lemma 4(a), we obtain

$$\mathbf{H}(Mon(\Phi)[(P \cup \nu(\Phi))/X][\hat{X}/Y]) \tag{10.83}$$

Writing out $Mon(\Phi)[(P \cup \nu(\Phi))/X][\hat{X}/Y]$ we get $P \subseteq \hat{X} \to Q[(P \cup \nu(\Phi))/X] \subseteq Q[\hat{X}/X]$. Therefore, 10.83 can be rewritten as $\exists(\mathbf{R}(P \cup \nu(\Phi))) \subseteq \hat{X} \to \exists(\mathbf{R}(Q[(P \cup \nu(\Phi))/X])) \subseteq \mathbf{H}(Q[\hat{X}/X])$. If we define $\hat{X}$ as $\exists(\mathbf{R}(P \cup \nu(\Phi)))$ and use Lemma 3, we obtain

$$\begin{array}{rll} \exists(\mathbf{R}(Q[(P \cup \nu(\Phi))/X])) & \subseteq \mathbf{H}(Q[\hat{X}/X])[\exists(\mathbf{R}(P \cup \nu(\Phi)))/\hat{X}] & \\ \exists(\mathbf{R}(Q[(P \cup \nu(\Phi))/X])) & \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P \cup \nu(\Phi)))/X] & \text{subst.} \\ \exists(s \circ \mathbf{R}(P)) & \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P \cup \nu(\Phi)))/X] & \text{by 10.82} \\ \exists(\mathbf{R}(P)) & \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P \cup \nu(\Phi)))/X] & \text{Lem. 7} \end{array}$$

This corresponds to the goal. □

The Soundness Theorem covers the denotational semantics aspect, showing how we can construct programs from a given proof. An overview of realizers for selected inference rules in IFP' is given in fig. 10.2. The cases presented in this table are for non-Harrop formulas.

The next chapter describes operational semantics of our system and presents a proof of the adequacy theorem, showing the correspondence between the denotational semantics and the operational semantics.

Natural deduction:

Introduction	Elimination
$\dfrac{\diamond \vdash p\,\mathbf{r}\,A \qquad \diamond \vdash q\,\mathbf{r}\,B}{\diamond \vdash \mathbf{Pair}(p,q)\,\mathbf{r}\,A \wedge B}\ \wedge^{+}$	$\dfrac{\diamond \vdash p\,\mathbf{r}\,A \wedge B}{\diamond \vdash (\pi_{\mathbf{Lt}}\,p)\,\mathbf{r}\,A}\ \wedge^{-}_{l}$ $\quad$ $\dfrac{\diamond \vdash p\,\mathbf{r}\,A \wedge B}{\diamond \vdash (\pi_{\mathbf{Rt}}\,p)\,\mathbf{r}\,B}\ \wedge^{-}_{r}$
$\dfrac{\diamond, a\,\mathbf{r}\,A \vdash p\,\mathbf{r}\,B}{\diamond \vdash (\lambda a\, p)\,\mathbf{r}\,A \to B}\ \to^{+}$	$\dfrac{\diamond \vdash p\,\mathbf{r}\,A \to B \qquad \diamond \vdash q\,\mathbf{r}\,A}{\diamond \vdash (p\,q)\,\mathbf{r}\,B}\ \to^{-}$
$\dfrac{\diamond \vdash p\,\mathbf{r}\,A}{\diamond \vdash \mathbf{Lt}(p)\,\mathbf{r}\,A \vee B}\ \vee^{+}_{l}$ $\quad$ $\dfrac{\diamond \vdash p\,\mathbf{r}\,B}{\diamond \vdash \mathbf{Rt}(p)\,\mathbf{r}\,A \vee B}\ \vee^{+}_{r}$	$\dfrac{\diamond \vdash q\,\mathbf{r}\,A \vee B \qquad \diamond \vdash p_1\,\mathbf{r}\,A \to C \qquad \diamond \vdash p_2\,\mathbf{r}\,B \to C}{\diamond \vdash \mathbf{case}\ q\ \mathbf{of}\ \{\mathbf{Lt}(a) \to (p_1\,a); \mathbf{Rt}(b) \to (p_1\,b)\}\,\mathbf{r}\,C}\ \vee^{-}$
$\dfrac{\diamond \vdash p\,\mathbf{r}\,A(x)}{\diamond \vdash p\,\mathbf{r}\,\forall x\, A(x)}\ \forall^{+}$	$\dfrac{\diamond \vdash p\,\mathbf{r}\,\forall x\, A(x)}{\diamond \vdash p\,\mathbf{r}\,A(t)}\ \forall^{-}$
$\dfrac{\diamond \vdash p\,\mathbf{r}\,A(t)}{\diamond \vdash p\,\mathbf{r}\,\exists x\, A(x)}\ \exists^{+}$	$\dfrac{\diamond \vdash p\,\mathbf{r}\,\exists x\, A(x) \qquad q\,\mathbf{r}\,\forall x\, (A(x) \to B)}{\diamond \vdash (q p)\,\mathbf{r}\,B}\ \exists^{-}$

Induction and Coinduction:

$\dfrac{\diamond \vdash s\,\mathbf{r}\,\Phi(P) \subseteq P \qquad \diamond \vdash m\,\mathbf{r}\,Mon(\Phi)}{\diamond \vdash \mathbf{rec}(\lambda f (s \circ (m\ f)))\,\mathbf{r}\,\mu\,\Phi \subseteq P}\ \text{IND'}$	$\dfrac{\diamond \vdash s\,\mathbf{r}\,P \subseteq \Phi(P) \qquad \diamond \vdash m\,\mathbf{r}\,Mon(\Phi)}{\diamond \vdash \mathbf{rec}(\lambda f ((m\ f) \circ s))\,\mathbf{r}\,P \subseteq \nu\,\Phi}\ \text{COIND'}$
$\dfrac{\diamond \vdash s\,\mathbf{r}\,\Phi(P \cap \mu\,\Phi) \subseteq P \qquad \diamond \vdash m\,\mathbf{r}\,Mon(\Phi)}{\diamond \vdash \mathbf{rec}(\lambda f (s \circ (m\ f)))\,\mathbf{r}\,\mu\,\Phi \subseteq P}\ \text{SI'}$	$\dfrac{\diamond \vdash s\,\mathbf{r}\,P \subseteq \Phi(P \cup \nu\,\Phi) \qquad \diamond \vdash m\,\mathbf{r}\,Mon(\Phi)}{\diamond \vdash \mathbf{rec}(\lambda f ((m\ f) \circ s))\,\mathbf{r}\,P \subseteq \nu\,\Phi}\ \text{SC'}$
$\dfrac{\diamond \vdash s\,\mathbf{r}\,\Phi(P) \cap \mu\,\Phi \subseteq P \qquad \diamond \vdash m\,\mathbf{r}\,Mon(\Phi)}{\diamond \vdash \mathbf{rec}(\lambda f (s\ \circ \langle (m\ f), \mathbf{id} \rangle))\,\mathbf{r}\,\mu\,\Phi \subseteq P}\ \text{HSI'}$	$\dfrac{\diamond \vdash s\,\mathbf{r}\,P \subseteq \Phi(P) \cup \nu\,\Phi \qquad \diamond \vdash m\,\mathbf{r}\,Mon(\Phi)}{\diamond \vdash \mathbf{rec}(\lambda f ([(m\ f) + \mathbf{id}]\ \circ\ s))\,\mathbf{r}\,P \subseteq \nu\,\Phi}\ \text{HSC'}$

$\diamond$ stands for $\mathbf{H}(\Delta), \vec{a}\,\mathbf{r}\,\Gamma$. $\forall^{+}$ is subject to a variable condition that $x \notin FV(\Gamma)$. $\exists^{-}$ is also subject to a variable condition that $x \notin FV(B)$. **id** stands for identity.

Figure 10.2: Selected IFP' inference rules with realizers

Chapter 11

Operational semantics and adequacy

Contents

11.1 Finiteness, totality and approximation **112**

11.2 Big-step semantics **116**

11.3 Adequacy for finite computations **119**

11.4 Small-step semantics **124**

11.5 Adequacy for infinite computations **129**

Whilst the Soundness theorem confirms that the denotational semantics of an extracted program is indeed a correct realizer of a formula proven in IFP, we still need to introduce operational semantics to define how the data that realizes this formula is calculated. This section gives an overview of the operational semantics as defined by Berger and Tsuiki in [24] and presents the Adequacy Theorem, showing the direct relation between denotational and operational semantics.

Traditionally, in operational semantics we distinguish between *natural semantics*, which deals with the big-step reduction relation between closed programs and values, and *structural operational semantics*, which looks into how small step computations are performed. The latter is specifically interesting from the point of view of infinite data computation.

In the initial drafts of [24], Berger considered the use of rules with closures as in [16]. This approach meant that substitution at each evaluation step was avoided through keeping track of the changes of the environment. While reviewing the definitions, however, we came across an issue because the information

contained in the environment component was not always preserved when small-step rules came into play.

Therefore, in the published version of this paper transition rules do not use closures. They rely on substitution instead. This makes the rules easier to read. Nevertheless, that comes at the price of a trickier implementation and potential computational blow-up due to substitution.

To solve this issue, in this thesis we return to the approach with closures. Using closures as defined in [16] is not sufficient for our purposes. Hence, we introduce the notion of an *extended closure*, which will be explained later in section 11.4, redefine the small-step rules accordingly, and prove that big-step semantics and small-step semantics are equivalent. The proofs of adequacy from [16] and [24] are revisited and adjusted to match this new approach. Here we closely follow the structure of the operational semantics section in [24], modifying the old definitions, adding new ones, and adjusting proofs with respect to these alternations where needed.

11.1 Finiteness, totality and approximation

Before looking at the reduction rules, we define the notion of *data*. This definition is based on the definition of the Scott domain of realizers / programs, as presented in section 9.3.

$$D = (\mathbf{Nil} + \mathbf{Lt}(D) + \mathbf{Rt}(D) + \mathbf{Pair}(D \times D) + \mathbf{F}(D \to D))_{\perp}$$

We introduce a domain E of *data* elements, which is a subset of D:

$$E = (\mathbf{Nil} + \mathbf{Lt}(E) + \mathbf{Rt}(E) + \mathbf{Pair}(E \times E))_{\perp}$$

The domain of *data* elements is visually represented in fig. 11.1.

Inductive and coinductive definitions occur in proofs and programs often and, therefore, we introduce special operators to represent the least and the greatest fixed points. This allows us to be more concise.

The least and the greatest fixed points of the operators Φ and $\Phi_{\perp}$ defined below are used to refer to different kinds of data, including

- *arbitrary* ($E \stackrel{\text{Def}}{=} \nu(\Phi_{\perp})$),
- *finite* ($E_{\text{f}} \stackrel{\text{Def}}{=} \mu(\Phi_{\perp})$),
- *total* ($E_{\text{t}} \stackrel{\text{Def}}{=} \nu(\Phi)$), and
- *finite total data* ($E_{\text{ft}} \stackrel{\text{Def}}{=} \mu(\Phi)$), where

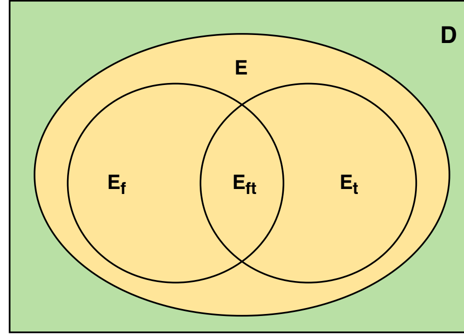

Figure 11.1: Visualisation of the domain of realisers, the domain of data and its sub-domains

$$\Phi(X)(a) \stackrel{\text{Def}}{=} \bigvee_{\substack{C\\ \text{constructor}}} \left(\exists a_1,\ldots,a_k \;\; a = C(a_1,\ldots,a_k) \wedge \bigwedge_{i\leq k} X(a_i)\right)$$

and $\Phi_\perp$ is a version of Φ with an additional option of $\perp$

$$\Phi_\perp(X)(a) \stackrel{\text{Def}}{=} a = \perp \vee \Phi(X)(a)\,.$$

Technically, it is possible to define special inference rules of a form, e.g.,

$$\frac{a_1 \in E_\mathrm{f},\ldots,a_k \in E_\mathrm{f}}{C(a_1,\ldots,a_k)}$$

for every constructor C. Since E_f also includes $\perp$, an additional rule $\perp \in E_\mathrm{f}$ is needed. For E_ft, this rule is not included. Rules for E and E_t are be build in the same way but there may be infinite derivations. Binary versions of Φ and $\Phi_\perp$ are defined as follows:

$$\Phi^2(X)(a,b) \stackrel{\text{Def}}{=} \bigvee_C \left(\begin{array}{r}\exists a_1,\ldots,a_k,b_1,\ldots,b_k \; a = C(a_1,\ldots,a_k) \wedge \\ b = C(b_1,\ldots,b_k) \wedge \bigwedge_{i\leq k} X(a_i,b_i)\end{array}\right)$$

$$\Phi^2_\perp(X)(a,b) \stackrel{\text{Def}}{=} a = \perp \vee \Phi^2(X)(a,b)$$

$$\Phi^2_{\perp,\perp}(X)(a,b) \stackrel{\text{Def}}{=} a = b = \perp \vee \Phi^2(X)(a,b)$$

These binary operators are used in the following relation:

$$a \sqsubseteq_E b \stackrel{\text{Def}}{=} \nu(\Phi^2_\bot)(a,b) \qquad \text{(domain ordering on } E\text{)}$$
$$\text{appr}(a,b) \stackrel{\text{Def}}{=} \mu(\Phi^2_\bot)(a,b) \qquad \text{(finite approximation)}$$
$$\text{eq}(a,b) \stackrel{\text{Def}}{=} \nu(\Phi^2_{\bot,\bot})(a,b) \qquad \text{(bisimilarity)}$$
$$\text{feq}(a,b) \stackrel{\text{Def}}{=} \mu(\Phi^2)(a,b) \qquad \text{(finite bisimilarity)}$$
$$\text{teq}(a,b) \stackrel{\text{Def}}{=} \nu(\Phi^2)(a,b) \qquad \text{(total bisimilarity)}$$

We refrain from defining $\sqsubseteq$ here for simplicity and instead state the properties of the domain ordering on D.

(i) $\bot \sqsubseteq a$

(ii) $C(\vec{a}) \sqsubseteq C(\vec{b}) \leftrightarrow \vec{a} \sqsubseteq \vec{b}$ for every constructor C

(iii) $\mathbf{F}(f) \sqsubseteq \mathbf{F}(g) \leftrightarrow \forall a \in D(f(a) \sqsubseteq g(a))$

(iv) $C(\vec{a}) \not\sqsubseteq C'(\vec{b})$ for any two different constructors

(v) $C(\vec{a}) \not\sqsubseteq \mathbf{F}(f) \wedge \mathbf{F}(f) \not\sqsubseteq C(\vec{a})$ for every constructor C

In terms of domain ordering, for $a \in E$, $a \sqsubseteq_E b$ holds iff $a \sqsubseteq b$, where the latter is the domain ordering on D. With respect to $\text{appr}(a,b)$, $\text{teq}(a,b)$ and $\text{feq}(a,b)$, there are the following equivalences:

$$\text{appr}(a,b) \leftrightarrow a \sqsubseteq_E b \wedge a \in E_{\text{f}}$$
$$\text{teq}(a,b) \leftrightarrow \text{eq}(a,b) \wedge a,b \in E_{\text{t}}$$
$$\text{feq}(a,b) \leftrightarrow a = b \wedge a,b \in E_{\text{f}} \leftrightarrow \text{eq}(a,b) \wedge a,b \in E_{\text{f}}$$

$\text{eq}(a,b)$ does not logically imply $a = b$ but rather expresses the fact that the elements of E are fully determined by their finite approximations, i.e. that E is an algebraic domain. Therefore, we add the new axiom to RIFP:

$$\forall a,b\,(\text{eq}(a,b) \to a = b) \tag{11.1}$$

This axiom enables one to derive the equivalence of $(a = b \wedge a,b \in E)$ and $(a \sqsubseteq_E b \wedge b \sqsubseteq_E a)$, as well as the fact that if $a \sqsubseteq_E b$ and $a \in E_{\text{t}}$, then a and b are equal.

Some of the observations mentioned above are included in the following lemma, covering the usual examples of (co)inductive proofs on data.

Lemma 11

(a) $\text{appr}(a,b) \leftrightarrow (a \sqsubseteq_E b \wedge a \in E_{\text{f}})$.

(b) $a \sqsubseteq_E b \leftrightarrow (\forall d(\text{appr}(d,a) \to \text{appr}(d,b)) \ \wedge a \in E)$.

(c) $\text{teq}(a,b) \leftrightarrow \text{eq}(a,b) \wedge a,b \in E_{\text{t}}$

(d) $\text{feq}(a,b) \leftrightarrow a = b \wedge a,b \in E_{\text{f}} \leftrightarrow \text{eq}(a,b) \wedge a,b \in E_{\text{f}}$

Proof. The proofs of (a) and (b) are as in Lemma 24 in [24], i.e, (a) left to right is proven by induction on $\text{appr}(a,b)$ and right to left by induction on $E_{\text{f}}(a)$; (b) left to right is by (a) and right to left by coinduction on $a \sqsubseteq_E b$.

(c) Left to right. Firstly, we show $\text{teq}(a,b) \to \text{eq}(a,b)$ for all a, b is by coinduction on eq. Let $P(a,b) = \text{teq}(a,b) = \nu(\Phi^2)(a,b)$. By coinduction it suffices to show $\forall a,b(P(a,b) \to \nu(\Phi^2_{\perp,\perp})(a,b))$. Assume $P(a,b)$. Using the definition of $\Phi^2_{\perp,\perp}$, it suffices to show that $a = b = \perp \vee \Phi^2(P)(a,b)$. P is a fixed point of Φ^2, which means that $\Phi^2(P) = P$ and so $a = b = \perp \vee P(a,b)$. By $\vee_r^+$, we it suffices to show $P(a,b)$, which matches the above assumption. Secondly, we need to show (i) $(\exists b \ \text{teq}(a,b)) \to a \in E_{\text{t}}$ and (ii) $(\exists a \ \text{teq}(a,b)) \to b \in E_{\text{t}}$ by coinduction on E_{t}. For (i), assume $P(a) = \exists b \ \text{teq}(a,b)$. Since E_{t} is the largest fixed point of Φ, by coinduction it suffices to show $P(a) \to \Phi(P)(a)$. Assume $a = C(a_1,\dots a_k)$ and $b = C(b_1,\dots b_k)$, where all arguments are pairwise equal, i.e., $\text{teq}(a_i,b_i)$, where $1 \leq i \leq k$. We have to find C', a'_i such that $a = C'(\vec{a'_i}) \wedge P(a'_i)$. We choose $C' = C$ and $a'_i = a_i$. To show $P(a_i)$ we have to find b'_i such that $\text{teq}(a_i, b'_i,)$. We choose $b'_i = b_i$, so $\text{teq}(a_i,b_i,)$, which corresponds to the assumption. For (ii) the proof is symmetric.

Right to left by coinduction on $\text{teq}(a,b)$. Assume $P(a,b)$, that is, $\text{eq}(a,b) \wedge a,b \in E_{\text{t}}$. Our goal is $\forall a,b(P(a,b) \to \nu(\Phi^2)(a,b))$, so by coinduction it suffices to show $\forall a,b(P(a,b) \to \Phi^2(P)(a,b))$. Since $a,b \in E_{\text{t}}$, $a = C(a_1,\dots,a_k)$ and $b = C'(b_1,\dots,b_k)$ with a_i,b_i in E_{t}, where $1 \leq i \leq k$, for some constructors C, C'. Since $\text{eq}(a,b)$, it follows that $C = C'$ and $\text{eq}(a_i,b_i)$. Therefore, $P(a_i,b_i)$, which entails $\Phi^2(P)(a,b)$.

(d) Left to right is covered by (i) and (ii) and right to left by (iii) and (iv).

(i) $\text{feq}(a,b) \to a = b \wedge a,b \in E_{\text{f}}$. Proof is by induction on $\text{feq}(a,b)$.

(ii) $a = b \wedge a,b \in E_{\text{f}} \to \text{eq}(a,b) \wedge a,b \in E_{\text{f}}$. Proof is by coinduction on $\text{eq}(a,b)$.

(iii) $a = b \wedge a,b \in E_{\text{f}} \to \text{feq}(a,b)$. Since $a = b$, it suffices to show $a \in E_{\text{f}} \to \text{feq}(a,a)$. Proof is by induction on E_{f}.

(iv) $\text{eq}(a,b) \wedge a,b \in E_{\text{f}} \to a = b \wedge a,b \in E_{\text{f}}$. Assume $a \in E_{\text{f}}$. It suffices to show $\forall b(\text{eq}(a,b) \to a = b)$. Proof is by induction on E_{f}.

□

With the initial definitions covered, we now proceed to more specific aspects of operational semantics, starting with big-step semantics.

11.2 Big-step semantics

One of the ways to define operational semantics is using the big-step approach. *Big-step reduction rules* in our approach use inductively defined closures.

A *closure* is a pair of a program with the corresponding environment, for instance (M,η), where *environment* is a finite mapping from object variables to closures.

A *value* is a closure (M,η), where the program M begins with a constructor.

A *closed program M* is a closure $(M,\emptyset)$ where $\emptyset$ is the empty mapping.

The *big-step reduction relation* ($\Downarrow$) from closures U to values V is defined as follows:

$$\text{(I, big)} : V \Downarrow V$$

$$\text{(II, big)} : \frac{\eta(a) \Downarrow V}{(a,\eta) \Downarrow V}.$$

$$\text{(III, big)} : \frac{(M,\eta) \Downarrow (C(M_1,\dots,M_k),\eta') \quad (N,\eta[u_1 \mapsto (M_1,\eta'),\dots,u_k \mapsto (M_k,\eta')]) \Downarrow V}{(\mathbf{case}\ M\ \mathbf{of}\ \{\dots C(u_1,\dots,u_k) \to N;\dots\},\eta) \Downarrow V}$$

$$\text{(IV, big)} : \frac{(M,\eta) \Downarrow (\lambda a\, M',\eta') \qquad (M',\eta'[a \mapsto (N,\eta)]) \Downarrow V}{(MN,\eta) \Downarrow V}$$

$$\text{(V, big)} : \frac{(M(\mathbf{rec}\ M),\eta) \Downarrow V}{(\mathbf{rec}\ M,\eta) \Downarrow V}$$

The above is a succinct way of representing the rules originally presented in [16] and an alternative interpretation of the rules defined in [24]. These versions can be interchanged.

The use of closures is beneficial as it provides a more detailed insight into what happens during evaluation. This approach also makes subsequent implementation process simpler, since there is no need to introduce additional substitution procedures for programs.

Lemma 12 For a closure U, there is at most one value V such that $U \Downarrow V$.

Proof. There is at most one big step reduction rule can be applied to a closure. □

In the framework of big-step semantics, reduction stops at constructors due to (I, big). However, to obtain data the evaluation process should proceed beyond that point and continue reduction under constructors.

Therefore, we define new operators Φ^{op} and $\Phi^{\mathrm{op}}_{\perp}$ with respect to the corresponding operators Φ^2 and $\Phi^2_{\perp}$ as follows:

$$\Phi^{\mathrm{op}}(X)(U,a) \stackrel{\mathrm{Def}}{=} \bigvee_{C} \left(\begin{array}{r} \exists M_1,\dots,M_k,\eta,a_1,\dots,a_k\ (U \Downarrow (C(M_1,\dots,M_k),\eta) \\ \wedge a = C(a_1,\dots,a_k) \wedge \bigwedge_{i\leq k} X((M_i,\eta),a_i)) \end{array} \right)$$

$$\Phi^{\mathrm{op}}_{\perp}(X)(U,a) \stackrel{\mathrm{Def}}{=} a = \perp \vee \Phi^{\mathrm{op}}(X)(U,a)\,.$$

The least and the greatest fixed points of these operators are used to define further four reduction relations from a closure U to data a:

$$\stackrel{\mu}{\Longrightarrow} \stackrel{\mathrm{Def}}{=} \mu(\Phi^{\mathrm{op}}) \qquad \stackrel{\nu}{\Longrightarrow} \stackrel{\mathrm{Def}}{=} \nu(\Phi^{\mathrm{op}})$$

$$\stackrel{\mu\perp}{\Longrightarrow} \stackrel{\mathrm{Def}}{=} \mu(\Phi^{\mathrm{op}}_{\perp}) \qquad \stackrel{\nu\perp}{\Longrightarrow} \stackrel{\mathrm{Def}}{=} \nu(\Phi^{\mathrm{op}}_{\perp})\,.$$

Each reduction corresponds to inductively defined rules. For instance, the reduction $U \stackrel{\star}{\Longrightarrow} a$ can be achieved using the below rules, where $\star$ stands for either μ or ν. In case of ν the rules are permitting infinite derivations.

$$\frac{U \Downarrow (\mathbf{Nil},\eta)}{U \stackrel{\star}{\Longrightarrow} \mathbf{Nil}} \qquad \frac{U \Downarrow (\mathbf{Pair}(M_1,M_2),\eta) \quad (M_1,\eta) \stackrel{\star}{\Longrightarrow} a_1 \quad (M_2,\eta) \stackrel{\star}{\Longrightarrow} a_2}{U \stackrel{\star}{\Longrightarrow} \mathbf{Pair}(a_1,a_2)}$$

$$\frac{U \Downarrow (\mathbf{Lt}(M),\eta) \quad (M,\eta) \stackrel{\star}{\Longrightarrow} a}{U \stackrel{\star}{\Longrightarrow} \mathbf{Lt}(a)} \qquad \frac{U \Downarrow (\mathbf{Rt}(M),\eta) \quad (M,\eta) \stackrel{\star}{\Longrightarrow} a}{U \stackrel{\star}{\Longrightarrow} \mathbf{Rt}(a)}$$

For $\stackrel{\mu\perp}{\Longrightarrow}$ and $\stackrel{\nu\perp}{\Longrightarrow}$ these rules also include the respective axioms $U \stackrel{\mu\perp}{\Longrightarrow} \perp$ and $U \stackrel{\nu\perp}{\Longrightarrow} \perp$.

Recalling the definitions of various kinds of data – arbitrary ($E \stackrel{\mathrm{Def}}{=} \nu(\Phi_{\perp})$), finite ($E_{\mathrm{f}} \stackrel{\mathrm{Def}}{=} \mu(\Phi_{\perp})$), total ($E_{\mathrm{t}} \stackrel{\mathrm{Def}}{=} \nu(\Phi)$), and finite total ($E_{\mathrm{ft}} \stackrel{\mathrm{Def}}{=} \mu(\Phi)$) – we define several important equivalences.

Lemma 13

(a) $U \stackrel{\mu}{\Longrightarrow} a$ iff $U \stackrel{\mu\perp}{\Longrightarrow} a \wedge a \in E_{\mathrm{ft}}$.

(b) $U \stackrel{\nu}{\Longrightarrow} a$ iff $U \stackrel{\nu\perp}{\Longrightarrow} a \wedge a \in E_{\mathrm{t}}$.

(c) $U \stackrel{\mu\perp}{\Longrightarrow} a$ iff $U \stackrel{\nu\perp}{\Longrightarrow} a \wedge a \in E_{\mathrm{f}}$.

(d) $U \stackrel{\nu\perp}{\Longrightarrow} a$ iff $\forall d\,(\mathrm{appr}(d,a) \to U \stackrel{\mu\perp}{\Longrightarrow} d) \wedge a \in E$.

Proof. Proceeds in the same way as Lemma 26 in [24] but works with closures instead of programs. We present the most interesting cases in more detail.

(a) Left to right by straightforward induction on $\stackrel{\mu}{\Longrightarrow}$. Right to left: the goal is logically equivalent to the formula $U \stackrel{\mu\perp}{\Longrightarrow} a \to (a \in E_{\mathrm{ft}} \to U \stackrel{\mu}{\Longrightarrow} a)$, which we prove by induction on $\stackrel{\mu\perp}{\Longrightarrow}$. Hence, it suffices to show that $\forall a, U(\Phi^{\mathrm{op}}_{\perp}(P)(U,a) \to P(U,a))$, where $P(U,a)$ is defined as $a \in E_{\mathrm{ft}} \to U \stackrel{\mu}{\Longrightarrow} a$. We unfold the definition of $\Phi^{\mathrm{op}}_{\perp}$ and $a = \perp \vee \Phi^{\mathrm{op}}(P)(U,a)$ to our assumptions. The reduction relation $\stackrel{\mu}{\Longrightarrow}$ is the least fixed point of Φ^{op} but since E_{ft} is defined as the least fixed point of Φ, a is of a constructor form. This allows us to prove the goal from the available assumptions.

(b) Left to right. We show $U \stackrel{\nu}{\Longrightarrow} a \to U \stackrel{\nu\perp}{\Longrightarrow} a$ by coinduction on $\stackrel{\nu\perp}{\Longrightarrow}$ and $U \stackrel{\nu}{\Longrightarrow} a \to a \in E_{\mathrm{t}}$ by coinduction on E_{t}. Right to left by coinduction on $\stackrel{\nu}{\Longrightarrow}$.

(c) Left to right by induction on $\stackrel{\mu\perp}{\Longrightarrow}$. Right to left: the goal is logically equivalent to $a \in E_{\mathrm{f}} \to (U \stackrel{\nu\perp}{\Longrightarrow} a \to U \stackrel{\mu\perp}{\Longrightarrow} a)$ and is proven by induction on E_{f}. Proof is similar as in (a).

(d) Left to right is proven as two sub-goals:

(i) $U \stackrel{\nu\perp}{\Longrightarrow} a \to a \in E$ proven by straightforward coinduction on E, and

(ii) $U \stackrel{\nu\perp}{\Longrightarrow} a \to \forall d(\mathrm{appr}(d,a) \to U \stackrel{\mu\perp}{\Longrightarrow} d)$, which is logically equivalent to $\forall d, a(\mathrm{appr}(d,a) \to \forall\, U(U \stackrel{\nu\perp}{\Longrightarrow} a \to U \stackrel{\mu\perp}{\Longrightarrow} d))$ and is proven by induction of $\mathrm{appr}(d,a)$. It suffices to show $\forall d, a(\Phi^2_{\perp}(P)(d,a) \to P(d,a))$, where $P(d,a) = \forall\, U(U \stackrel{\nu\perp}{\Longrightarrow} a \to U \stackrel{\mu\perp}{\Longrightarrow} d)$. Assume $\Phi^2_{\perp}(P)(d,a)$ and $U \stackrel{\nu\perp}{\Longrightarrow} a$. We have to show $U \stackrel{\mu\perp}{\Longrightarrow} a$. By the assumption $U \stackrel{\nu\perp}{\Longrightarrow} a$, $U \Downarrow (C(M_1,\ldots,M_k),\eta)$ and $a = C(a_1,\ldots,a_k)$ with $(M_i,\eta) \stackrel{\nu\perp}{\Longrightarrow} a_i$, for all

i (we use coclosure for $\stackrel{\nu\bot}{\Longrightarrow}$). By the assumption $\Phi^2_\bot(P)(d,a)$, we have $d = C(d_1,\ldots,d_k)$ and $P(d_i,a_i)$ for all i (the option $d = a = \bot$ is excluded since $a = C(a_1,\ldots,a_k)$). $(M_i,\eta) \stackrel{\nu\bot}{\Longrightarrow} a_i$ and $P(d_i,a_i)$ imply $(M_i,\eta) \stackrel{\mu\bot}{\Longrightarrow} a_i$ for all i. Therefore, $U \stackrel{\mu\bot}{\Longrightarrow} a$, by closure for $\stackrel{\mu\bot}{\Longrightarrow}$.

Right to left by coinduction on $\stackrel{\nu\bot}{\Longrightarrow}$. Suppose $a \in E$. Let $P(U,a)$ stand for $\forall d\,(\mathrm{appr}(d,a) \to U \stackrel{\mu\bot}{\Longrightarrow} d)$. We need to show $P(U,a) \to \Phi^{\mathrm{op}}_\bot(P)(U,a)$. Suppose that $P(U,a)$. Since $a \in E$, then either a is $\bot$ or is it of the form $C(a_1,\ldots,a_k)$ for $a_i \in E$. If $a = \bot$, then $\Phi^{\mathrm{op}}_\bot(P)(U,a)$ is straightforward from the definition of $\Phi^{\mathrm{op}}_\bot$. If $a = C(a_1,\ldots,a_k)$, then $\mathrm{appr}(C(\bot^k),a)$, which means that $U \stackrel{\mu\bot}{\Longrightarrow} C(\bot^k)$. Therefore, $U \Downarrow (C(M_1,\ldots,M_k),\eta)$ for some $M_1,\ldots,M_k$ and η. Now, for each $i \leq k$ we need to show that $P(M_i,a_i)$. Suppose that $\mathrm{appr}(d',a_i)$ and let $d = C(\bot^{i-1},d',\bot^{k-i})$. Since $\mathrm{appr}(d,a)$, we have $U \stackrel{\mu\bot}{\Longrightarrow} d$. Therefore, $M_i \stackrel{\mu\bot}{\Longrightarrow} d'$.

□

11.3 Adequacy for finite computations

To illustrate the link between the denotational and the operational semantics, computational adequacy needs to be shown. The initial version of the Adequacy Theorem was drafted in [16] and then modified in [24] to match the recent developments in the theory. The present version is very close to the one in [24] and it is adjusted to utilize closures.

Firstly, we define $\overline{U}$ to be the closed term represented by U, that is

$$\overline{(M,\eta)} = M[\overline{\eta(u)}/u \mid u \in \mathrm{FV}(M)]$$

It is possible to present computational adequacy in a general form using closures and closed terms. Instead, closed programs are used here as they suffice to show adequacy. A closed program M can be identified with the closure $(M,\emptyset)$, i.e., where the environment is empty.

Theorem 2 (Computational Adequacy I) :

(a) $M \stackrel{\mu}{\Longrightarrow} a$ iff $a = [\![M]\!] \wedge a \in E_{\mathrm{ft}}$.

(b) $M \overset{\mu\bot}{\Longrightarrow} a$ iff $a \sqsubseteq [\![M]\!] \wedge a \in E_{\mathrm{f}}$.

(c) $M \overset{\nu}{\Longrightarrow} a$ iff $a = [\![M]\!] \wedge a \in E_{\mathrm{t}}$.

(d) $M \overset{\nu\bot}{\Longrightarrow} a$ iff $a \sqsubseteq [\![M]\!] \wedge a \in E$.

Here M is a closed program. Part (a) represents the match between the denotational and the operational semantics for finite total data, that is the computational adequacy. A version for infinite total data is given in (c). The part (b) for finite partial data and the part (d) for partial arbitrary data.

Before we proceed with the proof of this theorem, we need to prove a number of supporting lemmas.

Lemma 14 (Correctness)

(a) If $U \Downarrow V$, then $[\![\overline{U}]\!] = [\![\overline{V}]\!]$.

(b) If $U \overset{\mu\bot}{\Longrightarrow} a$, then $\mathrm{appr}(a, [\![\overline{U}]\!])$.

(c) If $U \overset{\nu\bot}{\Longrightarrow} a$, then $a \sqsubseteq [\![\overline{U}]\!]$.

Proof. (a) by induction on $U \Downarrow V$.

(b) Let $P(U,a) \overset{\mathrm{Def}}{=} \mathrm{appr}(a, [\![\overline{U}]\!])$. To prove $U \overset{\mu\bot}{\Longrightarrow} a \to P(U,a)$ we proceed by induction and show $\Phi_{\bot}^{\mathrm{op}}(P)(U,a) \to P(U,a)$. Suppose $\Phi_{\bot}^{\mathrm{op}}(P)(U,a)$. If $a = \bot$, then $P(U,a)$ holds. If $\Phi^{\mathrm{op}}(P)(U,a)$, then $U \Downarrow (C(M_1,\dots,M_k),\eta)$, $a = C(a_1,\dots,a_k)$, and $P((M_i,\eta),a_i)$ for every $i \leq k$. Hence, by (a), $[\![\overline{U}]\!] = [\![\overline{(C(M_1,\dots,M_k),\eta)}]\!] = C([\![\overline{(M_1,\eta)}]\!],\dots,[\![\overline{(M_k,\eta)}]\!])$. Since $P((M_i,\eta),a_i)$, we have $\mathrm{appr}(a_i, [\![\overline{(M_i,\eta)}]\!])$ and, therefore, $\mathrm{appr}(a, [\![\overline{U}]\!])$).

(c) By Lemma 11(b), we need to show that $U \overset{\nu\bot}{\Longrightarrow} a$ and $\mathrm{appr}(d,a)$ implies $\mathrm{appr}(d, [\![\overline{U}]\!])$. The finite approximation is defined with the operator $\Phi_{\bot}^{2}$, so we can show that $U \overset{\nu\bot}{\Longrightarrow} a$ and $\mathrm{appr}(d,a)$ implies $U \overset{\nu\bot}{\Longrightarrow} d$. Since $U \overset{\nu\bot}{\Longrightarrow} d$ and $d \in E_{\mathrm{f}}$, we have $U \overset{\mu\bot}{\Longrightarrow} d$ by Lemma 13(b). Therefore, $\mathrm{appr}(d, [\![U]\!])$ by (b). □

We define D_0 as a the set of compact elements of D. Every element a in D_0 has a natural *rank*, $\mathbf{rk}(a) \in \mathbf{N}$, which satisfies the following:

rk1 If a has the form $C(a_1,\dots,a_k)$ for a data constructor C, then $a_1,\dots,a_k$ are compact and $\mathbf{rk}(a) > \mathbf{rk}(a_i)$ $(i \leq k)$.

rk2 If a has a form $\mathbf{F}(f)$, then for every $b \in D$, $f(b)$ is compact with $\mathbf{rk}(a) > \mathbf{rk}(f(b))$. Moreover, there exists a compact $b_0 \sqsubseteq b$ with $\mathbf{rk}(a) > \mathbf{rk}(b_0)$ and $f(b_0) = f(b)$.

Based on the above, for every compact a we assign a set of closed programs $\mathrm{Cl}(a)$ by induction on $\mathbf{rk}(a)$. Note that for $a \in E_0$ (i.e., $E_\mathrm{f}(a)$), $U \in \mathrm{Cl}(a)$ is equivalent to $U \stackrel{\mu\bot}{\Longrightarrow} a$. The definition is a generalized version of the one in [16] and is used in [24].

$$
\begin{array}{rcl}
\mathrm{Cl}(\bot) & = & \text{the set of all closures} \\
\mathrm{Cl}(C(a_1,\ldots,a_k)) & = & \{U \mid \exists M_1,\ldots,M_k,\eta,\, (U \Downarrow (C(M_1,\ldots,M_k),\eta) \wedge \\
& & \quad \bigwedge_{i\le k}(M_i,\eta) \in \mathrm{Cl}(a_i))\} \\
\mathrm{Cl}(\mathbf{F}(f)) & = & \{U \mid \exists x,M,\eta,\, (U \Downarrow (\lambda x M,\eta) \wedge \\
& & \quad \forall b \in D_0\,(\mathbf{rk}(b) < \mathbf{rk}(\mathbf{F}(f)) \to \\
& & \quad\quad \forall N \in \mathrm{Cl}(b)\,(M,\eta[N/x] \in \mathrm{Cl}(f(b)))))\}
\end{array}
$$

Lemma 15 (Monotonicity) For $a,b \in D_0$, if $a \sqsubseteq b$, then $\mathrm{Cl}(a) \supseteq \mathrm{Cl}(b)$.

Proof. The proof is by induction on $\mathbf{rk}(a)$. Assume $a \sqsubseteq b$.
Case $a = \bot$ is trivial as by the definition of $\mathrm{Cl}(\bot)$ for any b $\mathrm{Cl}(\bot) \supseteq \mathrm{Cl}(b)$.

Case $a = \mathbf{Nil}$. In this case b must also be $\mathbf{Nil}$. Hence, $\mathrm{Cl}(\mathbf{Nil}) = \mathrm{Cl}(\mathbf{Nil})$.

Case $a = \mathbf{Lt}(a)$. In this case b must also be of the form $\mathbf{Lt}(b)$. Show $\mathrm{Cl}(\mathbf{Lt}(a'))$ is a superset of $\mathrm{Cl}(\mathbf{Lt}(b'))$. Assume $U \in \mathrm{Cl}(\mathbf{Lt}(b'))$. This means that there is (M,η), s.t. $U \Downarrow (\mathbf{Lt}(M),\eta) \wedge (M,\eta) \in \mathrm{Cl}(b')$. By i.h., $\mathrm{Cl}(a') \supseteq \mathrm{Cl}(b')$ because by **rk**1, $\mathbf{rk}(a') < \mathbf{rk}(a)$. Therefore, the statement remains true if we replace b' by a' to get $U \Downarrow (\mathbf{Lt}(M),\eta) \wedge (M,\eta) \in \mathrm{Cl}(a')$. Hence, $U \in \mathrm{Cl}(\mathbf{Lt}(a'))$.

Cases $a = \mathbf{Rt}(a)$ and $a = \mathbf{Pair}(a,b)$. Same strategy as above.

Case $a = \mathbf{F}(f)$. Since $a \sqsubseteq b$, $b = \mathbf{F}(g)$ with $f \sqsubseteq g$, pointwise. Assume $U \in \mathrm{Cl}(b)$. We have to show $U \in \mathrm{Cl}(a)$. Since $U \in \mathrm{Cl}(b)$, there exist x,M,η such that (1) $U \Downarrow (\lambda x M,\eta)$ (2) whenever $\mathbf{rk}(c) < \mathbf{rk}(b)$, then for all $N \in \mathrm{Cl}(c)$, $(M,\eta[N/x]) \in \mathrm{Cl}(g(c))$. To show that $U \in \mathrm{Cl}(a)$, it suffices to show that (2′) whenever $\mathbf{rk}(c) < \mathbf{rk}(a)$, then for all $N \in \mathrm{Cl}(c)$, $(M,\eta[N/x]) \in \mathrm{Cl}(f(c))$. To prove (2′), assume $\mathbf{rk}(c) < \mathbf{rk}(a)$ and $N \in \mathrm{Cl}(c)$. We have to show $(M,\eta[N/x]) \in \mathrm{Cl}(f(c))$. By **rk**2, there exists a compact $c_0 \sqsubseteq c$ such that $\mathbf{rk}(c_0) < \mathbf{rk}(b)$ and $g(c_0) = g(c)$. Since $\mathbf{rk}(c) < \mathbf{rk}(a)$, it follows by induction hypothesis that $N \in \mathrm{Cl}(c_0)$. Since $\mathbf{rk}(c_0) < \mathbf{rk}(b)$, it follows

by (2) that $(M, \eta[N/x]) \in \mathrm{Cl}(g(c_0))$. Since $f \sqsubseteq g$ pointwise, it follows $f(c) \sqsubseteq g(c) = g(c_0)$. By **rk**2, $\mathbf{rk}(f(c)) < \mathbf{rk}(a)$ and $\mathbf{rk}(g(c_0)) < \mathbf{rk}(b)$. Hence, by the induction hypothesis, $(M, \eta[N/x]) \in \mathrm{Cl}(f(c))$.

□

Lemma 16 (Reducibility of closures) $U \in \mathrm{Cl}(a)$ iff $U \Downarrow V$ for some $V \in \mathrm{Cl}(a)$.

Proof. Immediate from the definition of $\mathrm{Cl}(a)$. □

Lemma 17 (Approximation) If a is compact and M is a closed program with $a \sqsubseteq [\![M]\!]$, then $(M, \eta) \in \mathrm{Cl}(a)$.

Proof. This is a special case of Lemma 15 in [16]. □

Lemma 18 If $a \in E_{\mathrm{f}}$ and $U \in \mathrm{Cl}(a)$, then $U \overset{\mu\perp}{\Longrightarrow} a$.

Proof. Proof by induction on E_{f}. We show $a \in E_{\mathrm{f}} \to \forall U\ (U \in \mathrm{Cl}(a) \to U \overset{\mu\perp}{\Longrightarrow} a)$. Let $P(a) = \forall U\ (U \in \mathrm{Cl}(a) \to U \overset{\mu\perp}{\Longrightarrow} a)$. By induction it suffices to show $\Phi_{\perp}(P)(a) \to P(a)$. Unfolding the definition of the operator, we have $a = \perp \vee \Phi(P)(a)$, where $\Phi(P)(a)$ means that there exist $a_1, \ldots, a_k$, such that $a = C(a_1, \ldots, a_k)$ and $P(a_i)$ for all i, such that $1 \leq i \leq k$. Assume $U \in \mathrm{Cl}(a)$. Show $U \overset{\mu\perp}{\Longrightarrow} a$. It suffices to show $a = \perp \vee \Phi^{\mathrm{op}}(\mu\Phi_{\perp}^{\mathrm{op}})(U, a)$ (by closure). In case $a = \perp$ the proof is trivial as we take left side of the disjunction. In case a is of a constructor form, we have the assumptions $P(a_i)$ as well as $U \in \mathrm{Cl}(a_1, \ldots, a_k)$, which means that $(M_i, \eta) \in \mathrm{Cl}(a_i)$. Hence, $(M_i, \eta) \overset{\mu\perp}{\Longrightarrow} a_i$. □

Now we have all the lemmas to prove the first Adequacy Theorem. For the reader's convenience let us reiterate the statements, replacing $\sqsubseteq$ by $\sqsubseteq_E$ since, taking into account the properties of domain ordering stated earlier, this does not change the meaning of the statements.

(a) $M \overset{\mu}{\Longrightarrow} a$ iff $a = [\![M]\!] \wedge a \in E_{\mathrm{ft}}$.

(b) $M \overset{\mu\perp}{\Longrightarrow} a$ iff $a \sqsubseteq_E [\![M]\!] \wedge a \in E_{\mathrm{f}}$.

(c) $M \overset{\nu}{\Longrightarrow} a$ iff $a = [\![M]\!] \wedge a \in E_{\mathrm{t}}$.

(d) $M \overset{\nu\perp}{\Longrightarrow} a$ iff $a \sqsubseteq_E [\![M]\!] \wedge a \in E$.

Proof. Recall that a closed program M is identified with the closure $(M,\emptyset)$ and that $M = \overline{(M,\emptyset)}$ and hence $[\![M]\!] = [\![\overline{(M,\emptyset)}]\!]$.

We first start by showing some connections between sub-parts of the theorem. Namely, in Thm. 2, part (b) implies part (a), and part (d) implies part (c) (more precisely its general form with closures U instead of closed terms M).

(b) implies (a): Assume (b). We show (a):

$M \stackrel{\mu}{\Longrightarrow} a$

iff

$M \stackrel{\mu\bot}{\Longrightarrow} a$ and $a \in E_{\mathrm{ft}}$ (by Lemma 13(a)

iff

$a \sqsubseteq_E [\![M]\!]$ and $a \in E_{\mathrm{ft}}$ (by (b) and since $E_{\mathrm{ft}} \subset E_{\mathrm{f}}$)

iff

$a = [\![M]\!]$ and $a \in E_{\mathrm{ft}}$ (since $E_{\mathrm{ft}} \subset E_{\mathrm{t}}$ and as remarked after the axiom (11.1), $a \sqsubseteq_E b$ and $a \in E_{\mathrm{t}}$ implies $a = b$)

(d) implies (c): Assume (d). We show (c):

$M \stackrel{\nu}{\Longrightarrow} a$

iff

$M \stackrel{\nu\bot}{\Longrightarrow} a$ and $a \in E_{\mathrm{t}}$ (by Lemma 13(b)

iff

$a \sqsubseteq_E [\![M]\!]$ and $a \in E_{\mathrm{t}}$ (by (d) and since $E_{\mathrm{t}} \subset E$)

iff

$a = [\![M]\!]$ and $a \in E_{\mathrm{t}}$ since $a \sqsubseteq_E b$ and $a \in E_{\mathrm{t}}$ implies $a = b$

Consequently, it suffices to prove (b) and (d).

(b): Left to right, i.e., $M \stackrel{\mu\bot}{\Longrightarrow} a \to a \sqsubseteq_E [\![M]\!] \wedge a \in E_{\mathrm{f}}$. Assume $M \stackrel{\mu\bot}{\Longrightarrow} a$. By lemma 14(b), $\mathrm{appr}(a,[\![M]\!])$ and by lemma 11(a), $a \in E_{\mathrm{f}}$.

Right to left. Assume $a \sqsubseteq_E [\![M]\!] \wedge a \in E_{\mathrm{f}}$. By lemma 17, $M \in \mathrm{Cl}(a)$ and by lemma 18, $M \stackrel{\mu\bot}{\Longrightarrow} a$.

(d): Left to right, i.e., $M \stackrel{\nu\bot}{\Longrightarrow} a \to a \sqsubseteq_E [\![M]\!] \wedge a \in E$ is proven by lemmas 14(c) and 13(d).

Right to left: $a \sqsubseteq_E [\![M]\!] \wedge a \in E \to M \stackrel{\nu\bot}{\Longrightarrow} a$. If $a \sqsubseteq_E [\![M]\!]$, then $a \sqsubseteq [\![M]\!]$. Therefore, by lemma 17, $[\![M]\!] \in \mathrm{Cl}(a)$. By lemma 18, $M \stackrel{\nu\bot}{\Longrightarrow} a$.

□

This concludes the proof of adequacy for finite computations.

11.4 Small-step semantics

In order to define structural operational semantics that corresponds to the above natural semantics, we introduce the notion of an *extended closure*, which is an application of a closure to a closure. Correspondingly, we give rules for evaluating these extended closures. An alternative would be using Plotkin-style environments [99] and rules which update environments in the premise. Instead, we update the environment for subsequent transitions, discarding the original environment. We find this approach cleaner and more suitable for our implementation.

- Every closure is an extended closure;
- if U is an extended closure and U_0 is a closure, then $U\,U_0$ is an extended closure;
- if U is an extended closure, $\vec{Cl}$ is a list of clauses, and η is an environment, then **case** U **of** $(\vec{Cl}, \eta)$ is an extended closure.

We use the letter V for values (as before), U, U' for extended closures, U_0, U_1 for closures, and we write $\eta[\vec{u} \mapsto (\vec{M}, \eta)]$ for $\eta[u_1 \mapsto (M_1, \eta), \ldots, u_k \mapsto (M_k, \eta)]$.

While big-step semantics stops at data constructors, we also need to look at infinite computations, which continues under data constructors. Constructors may have multiple arguments and therefore there is a need of computation that that runs in parallel to obtain the denotated value. This is why the small-step reduction rules are needed. We define small-step reduction relations ($\rightsquigarrow$) between closures. These relations terminate at values.

The rules of the small-step semantics are presented below. There is no rule that corresponds to (I, big). This is because the purpose of small-step semantics is to evaluate an expression one step at a time and since V is already a value, the reduction from $V \rightsquigarrow V$ is redundant.

The rest of the big-step rules are intuitively represented by the small-step ones. It is, however, necessary to add more precision to the small-step rules. We introduce three variations of the rules III and IV to make distinction between different forms of closures and handle them appropriately. The rules are ordered by letters with respect to their priority. A closure of the form (MN, η) can be broken down using (IV a, small). Then we look at the first closure to determine whether it needs to be broken down further, or it is already a value, and so on.

- (II, small) : $(a,\eta) \rightsquigarrow \eta(a)$

- (III a, small) : $(\textbf{case}\ M\ \textbf{of}\ \{Cl_1,...,Cl_n\},\eta) \rightsquigarrow \textbf{case}\ (M,\eta)\ \textbf{of}\ (\{Cl_1,...,Cl_n\},\eta)$

- (III b, small) : $\textbf{case}\ (C(M_1,...,M_k),\eta)\ \textbf{of}\ (\{...;C(u_1,...,u_k) \to N;...\},\eta_0) \rightsquigarrow (N,\eta_0[u_1 \mapsto (M_1,\eta),...,u_k \mapsto (M_k,\eta)])$

- (III c, small): $\dfrac{U \rightsquigarrow U'}{\textbf{case}\ U\ \textbf{of}\ (\{Cl_1,...,Cl_n\},\eta) \rightsquigarrow \textbf{case}\ U'\ \textbf{of}\ (\{Cl_1,...,Cl_n\},\eta)}$

- (IV a, small) : $(MN,\eta) \rightsquigarrow (M,\eta)(N,\eta)$

- (IV b, small) : $(\lambda a\ M,\eta)U_0 \rightsquigarrow (M,\eta[a \mapsto U_0])$

- (IV c, small) : $\dfrac{U \rightsquigarrow U'}{UU_0 \rightsquigarrow U'U_0}$

- (V, small) : $(\textbf{rec}\ M,\eta) \rightsquigarrow (M(\textbf{rec}\ M),\eta)$

There is exactly one applicable rule for each extended closure and, therefore, $\rightsquigarrow$ is a deterministic relation. We define $\rightsquigarrow^n$ as the n-times repetition of $\rightsquigarrow$, and $\rightsquigarrow^*$ as the transitive and reflexive closure of $\rightsquigarrow$.

Before proving the interchangeability between big and small-step rules, we prove specific characteristics of extended closures that are applications UU_0 or have the form $(\{Cl_1,\ldots,Cl_h\},\eta)$.

Lemma 19 If an extended closure UU_0 evaluates to a value V in n small steps, then U reduces to $(\lambda x\ M,\eta)$ in k steps and $(M,\eta[x \mapsto U_0])$ reduces to V in l steps for some term $\lambda x\ M$ and some environment η and program M, where $(k \le n) \wedge (l \le n)$.

Proof. (a) Let $UU_0 \rightsquigarrow^n V$, where $n \ge 0$. We proceed by induction on n and consider two cases:

Case $U = (\lambda x\ M,\eta)$. We have $UU_0 = (\lambda x\ M,\eta)U_0$. By (IV b, small) $(\lambda x\ M,\eta) \rightsquigarrow^1 (M,\eta[a \mapsto U_0])$, so by (a) $(M,\eta'[a \mapsto U_0]) \rightsquigarrow^{n-1} V$. Also, since $U = (\lambda x\ M,\eta)$ we know that U reduces to $(\lambda x\ M,\eta)$ in zero steps. Hence, $U \rightsquigarrow^0 (\lambda x\ M,\eta)$ and $(M,\eta[a \mapsto U_0])$ reduces to V in $n-1$ steps.

Case $U \neq (\lambda x\, M, \eta)$. We need to show $(U \leadsto^k (\lambda x\, M, \eta))$ and $(M, \eta[x \mapsto U_0]) \leadsto^l V$. By (IV c, small) $U \leadsto^1 U'$, so we know that $U'U_0 \leadsto^{n-1} V$. Hence, by i.h. we get the following:

(i) $k \leq n-1$

(ii) $l \leq n-1$

(iii) $U' \leadsto^k (\lambda x\, M, \eta)$

(iv) $(M, \eta[x \mapsto U_0]) \leadsto^l V$

Since U reduces to $U' \leadsto^k (\lambda x\ M, \eta)$ in one step, we know that U reduces to $(\lambda x\ M, \eta)$ in $K+1$ steps. Since we know (i) $k \leq n-1$ then also $k+1 \leq n$ holds. By (ii) and (iv) we know that $l \leq n-1$ but that also means that $l \leq n$. □

Lemma 20 If a **case** U **of** $(\{Cl_1, \ldots, Cl_h\}, \eta)$ evaluates to V in n small steps, then U reduces to $(C(M_1, \ldots, M_j), \eta')$ in k steps and $(N, \eta[u_1 \mapsto (M_1, \eta'), \ldots (u_j \mapsto M_j, \eta')])$ reduces to V in l steps for some term $C(M_1, \ldots, M_j)$, an environment η', a program N and a closure Cl, where $Cl \in \{Cl_1, \ldots, Cl_h\}$, $Cl = C(u_1, \ldots, u_j) \rightarrow N$, $(k \leq n)$, and $(l \leq n)$.

Proof. Let **case** U **of** $(\{Cl_1, \ldots, Cl_h\}, \eta) \leadsto^n V$, where $n \geq 0$. We proceed by induction on n and consider two cases:

Case $U = (C(M_1, \ldots, M_j), \eta')$. Let $(Cl \in \{Cl_1, \ldots, Cl_h\} \wedge Cl = C(u_1, \ldots, u_j) \rightarrow N)$. By (III b, small) it takes one step from **case** $(C(M_1, \ldots, M_j), \eta')$ **of** $(\{Cl_1, \ldots, Cl_h\}, \eta)$ to $(N, \eta[u_1 \mapsto (M_1, \eta'), \ldots, u_j \mapsto (M_j, \eta')])$. Since $U = (C(M_1, \ldots, M_j), \eta')$, we know that U reduces to $(C(M_1, \ldots, M_j), \eta')$ in 0 steps. Hence, $(N, \eta[u_1 \mapsto (M_1, \eta'), \ldots, u_j \mapsto (M_j, \eta')]) \leadsto^{n-1} V$. So in both cases the number of steps is smaller or equal to n.

Case $U \neq (C(M_1, \ldots, M_j), \eta')$. By (III c, small), **case** U' **of** $(\{Cl_1, \ldots, Cl_h\}, \eta) \leadsto^1 V$, so by i.h. we get the following:

(i) $k \leq n-1$

(ii) $l \leq n-1$

(iii) $Cl \in \{Cl_1, \ldots, Cl_h\} \wedge Cl = C(u_1, \ldots, u_j) \rightarrow N$

(iv) $U' \leadsto^k (C(M_1, \ldots, M_j), \eta')$

(v) $(N, \eta[u_1 \mapsto (M_1, \eta'), \ldots (u_j \mapsto M_j, \eta')]) \leadsto^l V$

Since U reduces to U' in one step and $U' \rightsquigarrow^k (C(M_1,\dots,M_j),\eta')$, we know that $U \rightsquigarrow^{k+1} (C(M_1,\dots,M_j),\eta')$. Since we know (i) $k \le n-1$ then also $k+1 \le n$ holds. By (ii) and (v) we know that $l \le n-1$ but that also means that $l \le n$.

□

The following lemma proves the interchangeability of big-step and small-step semantics.

Lemma 21 For a closure U and a value V, $U \Downarrow V$ iff $U \rightsquigarrow^* V$.

Proof. First, we proceed left to right ($U \Downarrow V \Rightarrow U \rightsquigarrow^* V$) by induction on derivation.

Case $U = V$. By (I, big) we know $V \Downarrow V$ and also $V \rightsquigarrow^0 V$ by reflexivity.

Case $U = (a,\eta)$. We need to show $(a,\eta) \rightsquigarrow^* V$. By (II, small) $(a,\eta) \rightsquigarrow \eta(a)$, hence by i.h. $(\eta(a) \Downarrow V \Rightarrow \eta(a) \rightsquigarrow^* V)$ and (II, big) we get $(a,\eta) \rightsquigarrow^* V$.

Case $U = (\textbf{case}\ M\ \textbf{of}\ \{\dots;C(u_1,\dots,u_k) \to N;\dots\},\eta)$. We first need to show that $(\textbf{case}\ M\ \textbf{of}\ \{\dots C(u_1,\dots,u_k) \to N;\dots\},\eta)$ reduces to V in some steps $(*)$. By (III a, small), $(\textbf{case}\ M\ \textbf{of}\ \{\dots C(u_1,\dots,u_k) \to N;\dots\},\eta)$ reduces in some steps $(*)$ to $\textbf{case}\ (M,\eta)\ \textbf{of}\ (\{\dots C(u_1,\dots,u_k) \to N;\dots\},\eta)$. By i.h. (M,η) reduces in $(*)$ steps to $(C(M_1,\dots,M_k),\eta')$. Hence, by applying (III c, small) multiple times it suffices to show:

$$\textbf{case}\ (C(M_1,\dots,M_k),\eta')\ \textbf{of}\ (\{\dots C(u_1,\dots,u_k) \to N;\dots\},\eta) \rightsquigarrow^* V \qquad (11.2)$$

We apply (III b, small) to $\textbf{case}\ (C(M_1,...,M_k),\eta)\ \textbf{of}\ (\{...;C(u_1,...,u_k) \to N;...\},\eta_0)$ to get $(N,\eta_0[u_1 \mapsto (M_1,\eta),...,u_k \mapsto (M_k,\eta)])$ in one reduction step. This new closure reduces to V by i.h. $(N,\eta[u_1 \mapsto (M_1,\eta'),\dots,u_k \mapsto (M_k,\eta')]) \rightsquigarrow^* V$. Therefore, we can conclude that eq. (11.2) holds.

Case $U = (MN,\eta)$. We need to show $(MN,\eta) \rightsquigarrow^* V$. By (IV a, small) we reduce (MN,η) to $(M,\eta)(N,\eta)$. Then by i.h.(a) and applying (IV c, small) multiple times it suffices to show that $(\lambda a\ M',\eta')(N,\eta) \rightsquigarrow^* V$. Since $(\lambda a\ M',\eta')(N,\eta)$ reduces to $(M',\eta'[a \mapsto (N,\eta)])$ by (IV b, small) and by i.h $M',\eta'[a \mapsto (N,\eta)] \rightsquigarrow^* V$. We can conclude that $(\lambda a\ M',\eta)(N,\eta)$ also reduces to V.

Case $U = (\textbf{rec}\ M,\eta)$. Show $(\textbf{rec}\ M,\eta) \rightsquigarrow^* V$. By (V, small) $(\textbf{rec}\ M,\eta) \rightsquigarrow (M(\textbf{rec}\ M),\eta)$. Hence, it suffices to show $(M(\textbf{rec}\ M),\eta) \rightsquigarrow^* V$, which corresponds to the i.h..

Now we proceed right to left ($U \rightsquigarrow^n V \Rightarrow U \Downarrow V$) by induction on n.

Base case: $n = 0$. Let $U = V$. By (I, big) $V \Downarrow V$.
Step: $n > 0$.
I.h. $\forall n.(n > 0) \rightarrow U \rightsquigarrow^1 U' \wedge U' \rightsquigarrow^{n-1} V \Rightarrow U' \Downarrow V$. We proceed by cases:

- Case $U = (a, \eta)$. Let $(a, \eta) \rightsquigarrow^n V$. Show: $(a, \eta) \Downarrow V$. Since (a, η) reduces to $\eta(a)$ in one step by (II, small) then $\eta(a) \rightsquigarrow^{n-1} V$. Consequently, by i.h. $\eta(a) \Downarrow V$, Finally, applying (II, big) to the latter we get $(a, \eta) \Downarrow V$.

- Case $U = (MN, \eta)$. Let $(MN, \eta) \rightsquigarrow^n V$. Show: $(MN, \eta) \Downarrow V$. By (IV a, small) $(MN, \eta) \rightsquigarrow^1 (M, \eta)(N, \eta)$, so by i.h. $(M, \eta)(N, \eta) \rightsquigarrow^{n-1} V$. Applying lemma 19 to the latter we we get:

  (i) $k \leq n-1$

  (ii) $l \leq n-1$

  (iii) $(M, \eta) \rightsquigarrow^k (\lambda x\, M', \eta')$

  (iv) $(M', \eta'[x \mapsto (N, \eta)]) \rightsquigarrow^l V$

  Since $(M, \eta) \rightsquigarrow^k (\lambda x\, M', \eta')$ and $k \leq n-1$ then by i.h. $(M, \eta) \Downarrow (\lambda x\, M', \eta')$. Similarly, since $l \leq n-1$ we know that $(M', \eta'[x \mapsto (N, \eta)]) \Downarrow V$. Hence by (IV, big) we get $(MN, \eta) \Downarrow V$.

- Case $U = (\textbf{case}\ M\ \textbf{of}\ Cl_1, \ldots, Cl_m, \eta)$. Let $(\textbf{case}\ M\ \textbf{of}\ Cl_1, \ldots, Cl_m, \eta) \rightsquigarrow^n V$. Show: $(\textbf{case}\ M\ \textbf{of}\ Cl_1, \ldots, Cl_m, \eta) \Downarrow V$. By (III a, small) we can reduce this closure to **case** (M, η) **of** $(Cl_1, \ldots, Cl_m, \eta)$ in one step, so by i.h. this reduces to V in $n-1$ steps. Applying lemma 20 to the the latter we get:

  (i) $k \leq n-1$

  (ii) $l \leq n-1$

  (iii) $Cl \in \{Cl_1, \ldots, Cl_h\} \wedge Cl = C(u_1, \ldots, u_j) \rightarrow N$

  (iv) $(M, \eta) \rightsquigarrow^k (C(M_1, \ldots, M_j), \eta')$

  (v) $(N, \eta[u_1 \mapsto (M_1, \eta'), \ldots (u_j \mapsto M_j, \eta')]) \rightsquigarrow^l V$

  Since $(M, \eta) \rightsquigarrow^k (C(M_1, \ldots, M_j), \eta')$ and $k \leq n-1$ then $(M, \eta) \Downarrow (C(M_1, \ldots, M_j)$. Similarly, since $(N, \eta[u_1 \mapsto (M_1, \eta'), \ldots (u_j \mapsto M_j, \eta')])$ reduces to V in l steps and $l \leq n-1$, we get $(N, \eta[u_1 \mapsto (M_1, \eta'), \ldots (u_j \mapsto M_j, \eta')]) \Downarrow V$. So, by (III, big) $(\textbf{case}\ M\ \textbf{of}\ Cl_1, \ldots, Cl_m, \eta) \Downarrow V$.

- Case $U = (\mathbf{rec}\, M, \eta)$ Let $(\mathbf{rec}\, M, \eta) \rightsquigarrow^n V$. Show: $(\mathbf{rec}\, M, \eta) \Downarrow V$. We know that $(\mathbf{rec}\, M, \eta)$ reduces to $(M(\mathbf{rec}\, M), \eta)$ in one step by (V, small). Thus, $(M(\mathbf{rec}\, M), \eta)$ reduces to V in $n-1$ steps. Consequently, by i.h. we know that $(M(\mathbf{rec}\ M), \eta) \Downarrow V$. Finally, applying (V, big) to the latter, we get $(\mathbf{rec}\, M, \eta) \Downarrow V$.

□

Now that the small semantics is defined we can proceed to proof adequacy for infinite computations.

11.5 Adequacy for infinite computations

Continuing computation under constructors may be tricky. If a constructor has multiple arguments, it may happen that some of them diverge, while others terminate. To solve this issue one needs to perform parallel computation in order to obtain a result that is close enough to the semantic value. With this is mind, special constructs called *closure-terms* are defined inductively as follows:

- every extended closure is a closure-term;
- if C is a constructor with arity k and W_i $(i \leq k)$ are closure-terms, then $C(W_1, \ldots, W_k)$ is a closure-term.

Note that a closure and a finite total data are both closure-terms. For a closure-term W, we use $W_\perp$ to denote the finite data obtained by replacing all the extended closures in W with $\perp$.

We extend $\rightsquigarrow$ from a relation between extended closures to a relation between closure-terms by adding the following two rules.

(VI) $(C(M_1, \ldots, M_k), \eta) \rightsquigarrow C((M_1, \eta), \ldots, (M_k, \eta))$

$$\text{(VII)}\quad \frac{W_i \rightsquigarrow W_i' \ (i = 1, \ldots, k)}{C(W_1, \ldots, W_k) \rightsquigarrow C(W_1', \ldots, W_k')}$$

Here, C ranges over all constructors. Note that **Nil** $\rightsquigarrow$ **Nil** by rule (VII). This set of rules is non-overlapping and covers all the cases. Therefore, there is exactly one applicable rule for each closure-term. A closure-term $W^{(n)}$ denotes W' such that $W \rightsquigarrow^n W'$.

Lemma 22 (Accumulation) If $W \rightsquigarrow W'$, then $W_\perp \sqsubseteq W'_\perp$. Therefore, ${W^{(n)}}_\perp \sqsubseteq {W^{(m)}}_\perp$ for $n \leq m$.

Proof. Induction on the structure of W.

Case W is an extended closure. We replace every extended closure with $\bot$, which means $W_\bot = \bot$ and, by the property of domain ordering (i), $\bot \sqsubseteq W'_\bot$.

Case W is of a form $C(W_i,\dots,W_k)$. Since W is of a closure-term of a constructor form, $W \rightsquigarrow C(W'_i,\dots,W'_k)$ by (VII). By i.h. if $W_i \rightsquigarrow W'_i$ $(i = 1,\dots,k)$, then $W_{i\bot} \sqsubseteq W'_{i\bot}$. Hence, $W_1 \sqsubseteq W'_1,\dots,W_k \sqsubseteq W'_k$. Applying the properly of the domain ordering (ii), $C(W_i,\dots,W_k) \sqsubseteq C(W'_i,\dots,W'_k)$. $\square$

For a closure U, finite approximation of the value of U after n computation steps is denoted by $U^{(n)}{}_\bot$. The following lemma shows that every finite approximation of the value of U is obtained sooner or later, making the computation is complete.

Lemma 23 (Adequacy for finite values) $U \stackrel{\mu\bot}{\Longrightarrow} a \to \exists n\,(a \sqsubseteq U^{(n)}{}_\bot)$.

Proof. Let $P(U,a) \stackrel{\text{Def}}{=} \exists n\,(a \sqsubseteq U^{(n)}{}_\bot)$. We need to show $U \stackrel{\mu\bot}{\Longrightarrow} a \to P(U,a)$. By induction, it suffices to show $\Phi^{\text{op}}_\bot(P)(U,a) \to P(U,a)$. If $a = \bot$, the statement of the lemma holds by the property of domain ordering (i). If we have

$$(U \Downarrow (C(M_1,\dots,M_k),\eta)) \wedge a = C(a_1,\dots,a_k) \wedge \bigwedge_{i\le k} (\exists n_i\; a_j \sqsubseteq (M_i,\eta)^{(n_i)}{}_\bot),$$

then, for n, the maximum of n_i $(i \le k)$, $C(a_1,\dots,a_k) \sqsubseteq C((M_1,\eta),\dots,(M_k,\eta))^{(n)}{}_\bot$ by Lemma 22. Also, we have $U^{(m)} = (C(M_1,\dots,M_k),\eta)$ for some m by Lemma 21, and $(C(M_1,\dots,M_k),\eta) \rightsquigarrow C((M_1,\eta),\dots,(M_k,\eta))$ by (VI). Hence, $a \sqsubseteq U^{(m+n+1)}{}_\bot$. $\square$

Since $(U^{(n)}{}_\bot)_n$ is an increasing sequence by Lemma 22, we define

$$U^{(\infty)} = \bigsqcup_n U^{(n)}{}_\bot$$

and we say that the closure U *infinitely computes* the data $U^{(\infty)}$.

We extend the definition of $\overline{U}$ for a closure U to a closure-term W and define $\overline{W}$ as the closed term represented by W.

Theorem 3 (Computational Adequacy II) If $[\![M]\!] \in E$, then $M^{(\infty)} = [\![M]\!]$.

Proof. Closure-terms have the following properties, which can be shown by induction on the definition $\rightsquigarrow$.

(a) For closure-terms W and W', $W \rightsquigarrow W' \to [\![\overline{W}]\!] = [\![\overline{W'}]\!]$.

(b) For a closure-term W, $W_\bot \sqsubseteq [\![\overline{W}]\!]$.

Here we need these properties only for a special case of closed terms. We start with M and reduce it to obtain $M^{(n)}$ and taking into account the above properties, we have ${M^{(n)}}_\perp \sqsubseteq [\![M]\!]$. Since this holds for every n, we have $M^{(\infty)} \sqsubseteq [\![M]\!]$.

By Theorem 2 (d), $M \overset{\nu\perp}{\Longrightarrow} [\![M]\!]$. Therefore, $\forall d\ (\mathrm{appr}(d, [\![M]\!]) \to M \overset{\mu\perp}{\Longrightarrow} d)$ by Lemma 13(d), considering the case that $U = (M, \emptyset)$. Therefore, by Lemma 23, we have $\forall d\ (\mathrm{appr}(d, [\![M]\!]) \to \exists n\ d \sqsubseteq {M^{(n)}}_\perp)$. Since $d \sqsubseteq {M^{(n)}}_\perp \to \mathrm{appr}(d, M^{(\infty)})$, we have $\forall d\ (\mathrm{appr}(d, [\![M]\!]) \to \mathrm{appr}(d, M^{(\infty)}))$. From this it follows that $[\![M]\!] \sqsubseteq M^{(\infty)}$ by Lemma 11(b). □

Part III

Program extraction in practice

Chapter 12

Development of the proof system

Contents

12.1 General structure of PRAWF 135
12.2 Logic component 139
12.3 Realizability and program extraction component 143
12.4 Execution component 145

Here we present PRAWF, an interactive proof assistant for program extraction that is build based on IFP.

Originally PRAWF was developed as an education tool for teaching natural deduction. In the first version it supported natural deduction for first order logic, enabling students to build proof trees and pretty print them using LaTeX. With the focus shifting towards program extraction, proofs became longer and, therefore, visualisation through LaTeX started to be inefficient and was dropped. The proof assistant is implemented in Haskell, a language, where side effects are minimized through separation of its purely functional part, which has no side effects, from its IO (input-output) part, which can have them. Functional style of programming is suitable for algorithmically complex programs like this one. Creation of data structures which closely correspond to the theory that we outlined earlier, for instance formulas and proofs, is easy and well supported in Haskell.

12.1 General structure of PRAWF

PRAWF encompasses the following components, namely:

- the *logic* component (LC)

- the *realizablity and program extraction* component (RPEC)
- the *execution* component (EC)

The logic component forms the base of the system. It covers first order logic, dual rules for induction and coinduction, closure and coclosure, as well as the usual equational reasoning rules like symmetry, reflexivity and congruence. In other words, this component corresponds to the IFP (and IFP'). The inference rules defined in this module are used for proof construction.

Realizability should be distinguished from program extraction. In the first case we consider elements of the Scott domain of realizers and in the second case we are dealing with program constructs that are used to represent these realizers. Realizability in this case is associated with RIFP and it serves as a link between the logic part of the system and the program extraction part. In terms of implementation, realizability and program extraction are integrated into one component, i.e., RPEC.

The execution component is an extra facility enabling users to execute their extracted programs.

From a user perspective, LC and RPEC correspond to the "proving-extracting" mode, where proofs are constructed and programs are extracted; EC corresponds to the execution mode, where one can run the extracted programs.

PRAWF consists of 19 modules, which can be divided based on their primary purpose into (a) logic-related modules, (b) realizability and extraction-related modules, (c) execution module, (d) IO modules, and (d) support modules (see fig. 12.1). IO modules cover interaction with the user, and the main modules here are `Mode` and `Prover`. There are also two IO modules that also fall into the support group - `ReadShow`, which processes the input and shows the output, and `Langandctxpreload`, which helps to load content for creating a proof environment.

Intuitively, the first three groups of modules correspond to the three components outlined above. Support modules are not necessarily directly related to any specific component. In terms of implementation, these groups of modules are closely intertwined. IO modules "orchestrate" interaction between various parts of the system. For instance, the `Prover` module links all the logic modules together. It also connects LC and RPEC through a function, which triggers program extraction. `Prover` includes functions that help creating a proof environment, work with recorded proof tactics, save theorems, etc. The `Mode` module is on a structurally higher level, guiding the user to select between the "proving-extracting" mode and the execution mode in PRAWF.

PRAWF uses goal-directed proof construction approach, which is common for most proof assistants. A proof begins with a goal formula. A user can apply

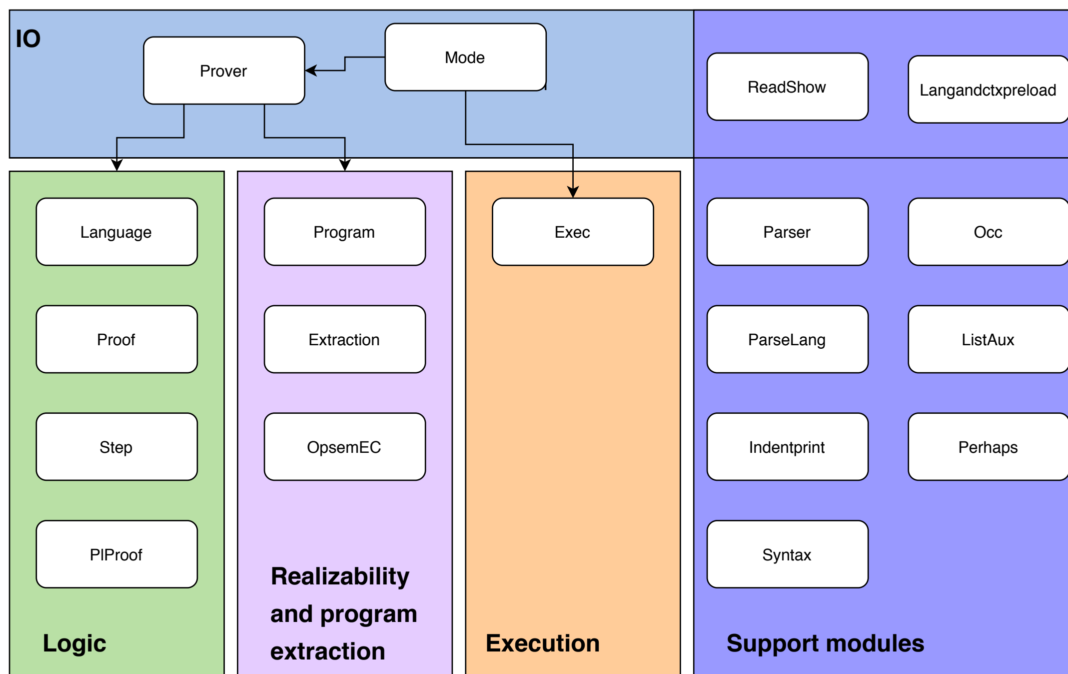

Figure 12.1: Intuitive structure of PRAWF

admissible IFP rules to transform the goal or break it down into smaller sub-goals (see fig. 12.2). Rules are applied in a backwards fashion. Each of the new sub-goals is put into the goal stack. Goals are processed one by one as shown by purple arrows in fig. 12.3. Once the end of a branch is reached (purple nodes), the previous available (unprocessed) internal node becomes the next goal. Goal labels inside of the nodes in fig. 12.3 correspond to goal labels as they would be displayed in PRAWF.

Generally proof steps are low level and correspond directly to the proof rules. However, there are minor automation implement to improve usability. There is no notion of a *tactic* as a composition of rules as in, for example, *Coq*. However, we use the notion of tactic record of a sequence of proof steps. This record can then be used to “replay” the proof. In this respect, PRAWF does not as many functionalities as the established proof assistants. However, the main focus of our proof assistant in not on functionalities but rather on a proof structure that works well for extraction.

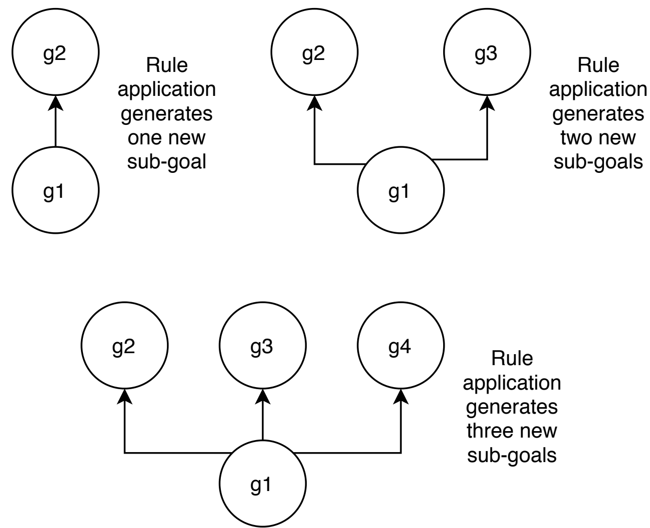

Figure 12.2: Generation of new goals through rule application

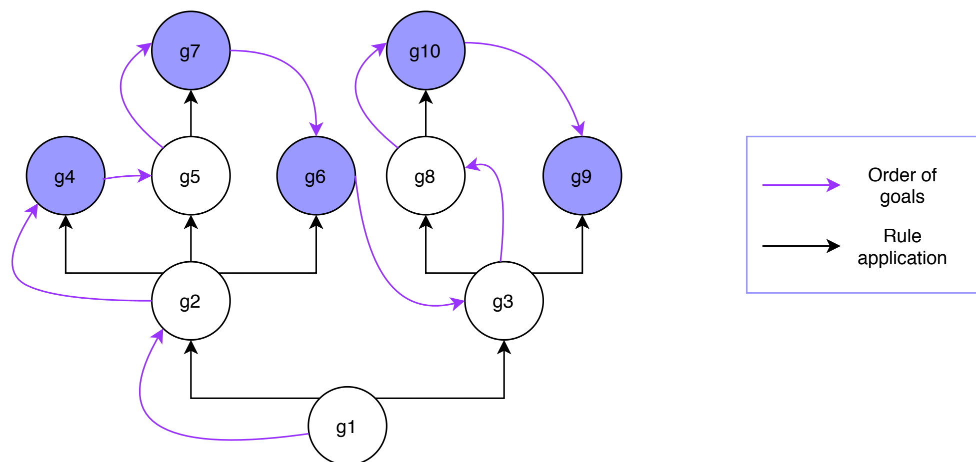

Figure 12.3: Order of goals

12.2 Logic component

The logic-related modules contain the machinery for constructing proofs. Here the notions like language, proof and rule are defined. This group of modules consists of `Prover`, `Language`, `Proof`, `PlProof` and `Step`.

Language In this module the core concepts like language, context, formula, predicate and term are defined.

A `Language` is a record that contains a list of sorts, constants, functions, predicates and operators:

```
data Language = Lang {
                       sorts      :: [String] ,
                       constants  :: [(String,Sort)] ,
                       functions  :: [(String,Type)] ,
                       predicates :: [(String,Arity)] ,
                       operators  :: [(String,Arity)]
                     }
           deriving (Show, Read)
```

Here `Sort` is represented by a string, `Arity` is a list of sorts, and `Type` is a tuple, consisting of an arity and a sort.

The use of the record type enables the users to define their own language and, hence, set up their specific proof environment. Aside of the language, the proof environment also comprises of a context, axioms, and declarations. The `Contex` is also defined as a record and includes fields for object variables, predicate variables, and operator variables:

```
data Context = Ctxt {
                      variables :: [(String,Sort)] ,
                      pvars     :: [(String,Arity)] ,
                      ovars     :: [(String,Arity)]
                    }
           deriving (Show, Read)
```

To add axioms and declarations, one needs a syntax for terms, formulas, predicates, and operators. These are defined as follows:

```
data Term = Var String | Const String | Fun String [Term]
           deriving (Show,Eq,Read)

data Formula = Predic Predicate [Term]
```

```
             | And Formula Formula
             | Or Formula Formula
             | Imp Formula Formula
             | Bot
             | All (String,Sort) Formula
             | Ex (String,Sort) Formula
             | LetF Decl Formula
          deriving (Show,Eq,Read)

data Predicate = PrVar String
               | PrConst String
               | Compr [(String,Sort)] Formula
               | Mu Op
               | Nu Op
               | OpApp Op Predicate
          deriving (Show,Eq,Read)

data Op = OpV String
        | OpA (String,Arity) Predicate
 deriving (Show,Eq,Read)
```

The idea of introducing the language in the above format did not come straight away. In the initial stages only formulas and terms were defined. That turned out to be inefficient as with the growing complexity of the logical structures and introduction of declarations and axioms the need for consistency and correctness checks become apparent. These checks are performed whenever a proof environment is loaded as well as at certain stages of proof construction and program extraction. Formulas, predicates, operators and their sub-parts are checked against the language and the context to ensure that the correct sorts, arities and types are used at all times.

Implementation of variable substitution was one of the trickiest tasks that required constant re-evaluation and updating. Certain inference rules require substitution and it needs to be performed on all kinds of expressions and, therefore, constant tracking of free and bound variables is essential. Since atomic formulas use a predicate in their structure:

```
(Predic Predicate [Term])
```

and a predicate can be a λ-abstraction, which uses a formula in its structure:

`(Compr [(String,Sort)] Formula)`,

keeping track of the variables is not a trivial matter because of constant switching between the predicates and formulas. Predicates and operators are also defined in terms of each other, causing the same difficulty during substitution and variable tracking.

Step, Proof, PlProof The rules used in PRAWF are defined in `Step` in the following way:

```
data Rule = AssuR | UseWithR | UseThmR | UseThmWithR
          | AndiR | AndelR | AnderR
          | OrilR | OrirR | OreR | OreAssuR
          | ImpiR | ImpeR
          | EfqR | RaaR
          | AlliR | AlleR | AlleRF
          | ExiR | ExeR
          | LetR | DeclR | NormR | UnfoldR
          | IndR | HSIndR | SIndR | ClR | CoclR
          | CoiR | HSCoiR | SCoiR
          | EqReflR | EqSymR | EqCongR | EqCongfR | EqCongruR
  deriving (Show, Read)
```

These rules correspond to the usual IFP rules and include some extra rules to enhance the interaction process for the users. Here `AssuR` corresponds to the use rule and the `Use...` rules are just its variations. `OreAssuR` is an enhanced variant of the disjunction elimination rule and so are the various versions of the congruence rule. `LetR` allows users to introduce declarations ad hoc and `NormR` and `UnfoldR` are used for normalization and unfolding of a formula accordingly. Here we also include *reductio ad absurdum* rule, which, although being a classical logic rule, can still be used in our system for non-computational formulas. Use of this rule with computational formulas is also possible but then extraction might not work anymore and just return an identity function. These rules are linked to how the users interact with the system. A very close notion related to these rules are proofs.

The data type of `Proof` is defined inductively, using constructors that correspond to the IFP proof rules, combined with the relevant sub-proof(s) or a label.

```
data Proof =
    AnProof Label
  | AndIntr Proof Proof
  ...
  | Ind Predicate Proof
  ...
```

`AnProof Label` stands for a use rule, where the assumption that is applied is referenced by `Label`. This label is a string, for example u. Other proofs are built up by combining smaller proofs under the corresponding proof constructor. For example, the rule for induction:

$$\frac{\Phi(P) \subseteq P}{\mu(\Phi) \subseteq P} \text{ is defined as } \texttt{Ind Predicate Proof}.$$

Here `Predicate` is expected to be of a form $\mu(\Phi)$ and `Proof` represents the connection between $\Phi(P) \subseteq P$ and $\mu(\Phi) \subseteq P$.

Proof trees in PRAWF are constructed in a backwards manner. Internally they are built by constantly recording the state of the system at every step of the proof. A `State` is a record consisting of number of fields, the most relevant of which is the `stplpproof` – partial proof. A partial proof is a record that contains a proof object, a list of goals that need to be proven for the proof to be complete, as well as lists of available assumptions, theorems, and declarations. Each goal is a record, consisting of a goal formula, a list of goal-specific assumptions, a context, and a list of declarations.

When a user builds a proof tree, the `stepI` function is utilized to move from one state to another, taking the user's input into consideration. The `stepI` function checks whether it is possible to apply the rule that the user has entered. This is done through refinement processes defined for each of the rules in the `PlProof` module. Going back to the induction rule, here the refinement process proves that one indeed can get $\Phi(P) \subseteq P$ from $\mu(\Phi) \subseteq P$. If that is the case, the state updates and the user can continue building the proof tree.

Once the proof is completed or whenever it is necessary to check that the proof tree is indeed built correctly, the `endFormula` function is used to go through the proof in the opposite (forward) direction. If the `endFormula` can recreate the initial goal, then the proof is correct.

The strategy for `endFormula` is straightforward. For the use rule, `endFormula` looks up the label it is given in the list of assumptions and if it is there, then it returns the corresponding formula. For the $\wedge^+$ rule, `endFormula` recursively calls itself to obtain the end formulas of the proofs it gets as an input. The obtained formulas are then merged using the `And` constructor. For the $\wedge_l^-$ rule, `endFormula` recursively calls itself to obtain the end formula of the proof it was given as an input. If the obtained formula is a conjunction, then the left side of this conjunction is returned. Other rules follow the same principle.

Once the proof is completed, there is an option to save it as a theorem, which can then be reused in more complex proofs.

While proofs are done in IFP, there is an internal procedure within the `transIFP` converts IFP proofs into IFP' proofs. It is a recursive function, which in case of

induction and coinduction adds an internally computed monotonicity proof to the premise of the inference rule.

12.3 Realizability and program extraction component

With the proof completed, one can start considering RPEC. The three core modules related to this component are:

(a) `Program`, in which the data type of `Program` is defined together with the procedures for program substitution and normalization;

(b) `OpsemEC`, which covers the operational semantics;

(c) `Extraction`, which contains the main program extraction algorithm `pe`.

Program To implement the notion of a program, we first define what a program constructor and a clause are:

```
data Constructor = Nil
                 | Lt
                 | Rt
                 | Pair
                 deriving (Show, Read, Eq)

type Clause = (Constructor, [String], Program)
```

In the `Clause` definition the list of strings refers to all local bound variables. For example, in `(Pair, ["a", "b"], prog)` the variables `"a"` and `"b"` are bound in `prog`. Now, we define `Program` inductively as follows:

```
data Program = ProgVar   String
             | ProgCon   Constructor [Program]
             | ProgCase  Program       [Clause]
             | ProgApp   Program       Program
             | ProgAbst  String            Program
             | ProgRec    Program
             deriving (Show, Read, Eq)
```

`ProgVar` corresponds to the program variable; `ProgCon Constructor [Program]` represents programs that use constructors. The constructor `Nil` is used when the program list is empty. A list of causes is used in `ProgCase Program  [Clause]`

to display all the possible options for the program M in an "or case", that is **case** M **of** $\{\ldots;\ C(\vec{x}) \to N;\ldots\}$. `ProgApp Program Program` is program application and `ProgRec Program` is a recursive program. `ProgAbst String Program` corresponds to $\lambda x\, M$, where M is a program.

OpsemEC In the `OpsemEC` module we define the notions of a closure, an environment and an extended closure.

```
data Closure = Clos Program Env

type Env = [(String,Closure)]
```

Extended closures are of three kinds:

- `ECCL Closure` corresponds to a simple extended closure U which is of a form (M,η);
- `ECCA EC [Clause] Env` is a way of expressing **case** U **of** $(\{Cl_1,\ldots,Cl_2\},\eta)$;
- `ECAP EC Closure` represents application of an extended closure to a closure.

This module also implements small step semantics that uses extended closures, as outlined in chapter 11. This approach enables to avoid complex substitution procedure during program evaluation. For our practical needs, big step semantics is redundant.

Extraction In this module the function `pe` (for program extraction) takes an IFP proof and converts it into an IFP' proof, using `transIFP` function defined in the `Proof` module. This new proof is fed into the main program extraction function, `pe'`.

Firstly, the proof is reconstructed to check whether it is possible to get its end formula. If that is the case, the formula is checked based on Harrop/non-Harrop criterion. For Harrop formulas, program extraction returns **Nil**. In case the end formula is non-Harrop, further distinction is made based on the proof. For instance, if it is a proof by assumption, then `ProgVar` with the label used in the proof is returned. In case of a proof by conjunction introduction, the end formulas of both sub-formulas of this conjunction are computed. If one of the sub-formulas is Harrop, the the other sub-formula is then in a recursive call to `pe'`. The program obtained from this call is then returned. If none of the sub-formulas is Harrop, then for each them there is a recursive call to `pe'` and a pair of the programs obtained in this way is returned.

For most of the proofs, procedures within `pe'` are fairly straightforward. The most interesting cases are proofs by induction and coinduction. In case induction, the form of the predicate provided as a part of the proof is checked. It is expected to be of a form $\mu(\Phi)$. If it is not, the program extraction fails. Otherwise, depending on whether Φ is Harrop or not, there are different scenarios.

If it is Harrop, then in all versions of induction, we have the same realizer, namely $s(m\ a)$ as seen in the soundness proof in chapter 10. Here s is a realizer of $\Phi(P) \subseteq P$ or corresponding formula in the other versions of induction, a is a realizer of $\mu(\Phi) \subseteq P$ and m is a realizer of the monotonicity proof.

If the operator is non-Harrop, there are different programs extracted for each type of induction. In case of IND', we get $s \circ (m\ f)$, for HSI' we get $s \circ \langle (m\ f), \mathbf{id} \rangle$, and for SI' we get $s \circ (m\ \langle f', \mathbf{id} \rangle)$ as per the soundness theorem and fig. 10.2.

In case of proofs by coinduction, firstly the end formula of the proof of $P \subseteq \Phi(P)$ is calculated, normalized with respect to the declarations in the proof environment, and stripped off the quantifiers. This resulting formula is expected to be an implication. The premise of this implication (corresponding to P) is then checked based on the Harrop/non-Harrop criterion.

If the formula is non-Harrop, then for COIND' we get $(m\ f) \circ s$, for HSC' we get $[(m\ f) + \mathbf{id}] \circ s$, and for SC' we get $(m\ [f' + \mathbf{id}]) \circ s$ as per the soundness theorem.

If the formula is Harrop, then for COIND' we get $(m\ a) \circ s$, for HSC' we get $[(m\ a) + \mathbf{id}] \circ s$, and for SC' we get $(m\ [\mathbf{id} + a]) \circ s$ as per the soundness theorem.

12.4 Execution component

The execution mode works on the principle of evaluating a program one step at a time. This is where the operational semantics comes into play. The users can chose how the result of evaluation is displayed, that is in a step-by-step manner or they can pick a specific number of steps to see the value at that point.

In order to provide a simple way of encoding numbers in the expected format, a special function for converting integers into programs is defined:

```
numgen :: Int -> Program
numgen 0 = ProgCon Lt [ProgCon Nil []]
numgen n = ProgCon Rt [numgen (n-1)]
```

Similarly, we include a function for defining a list of integers as a program:

```
listgen :: [Int] -> Program
listgen [] = ProgCon Lt [ProgCon Nil []]
listgen (x:xs) = ProgCon Rt [ProgCon Pair [numgen x, listgen xs]]
```

These two functions are created specifically for running our case studies with programs, which take numbers or lists of numbers as their input. This limited set of data types is sufficient for our purposes. However, this approach is rather rigid and requires users to add additional functions manually in order to work with other data types. Ideally, a more flexible solution, which does not require coding needs to be considered in the future.

The reader is welcome to watch a demo of PRAWF available at `https://www.youtube.com/watch?v=OY8dezi5054` as well as test the system on their own. The source files are stored in the public repository [100].

Chapter 13

Case studies

This chapter gives a walk-through from the stage when a proof is constructed to the stage wh en an extracted program is executed. We present a couple of simple programs in detail and outline a more complex example, a program for exact real numbers representations, which was presented in [22].

Before starting any proof in PRAWF, first a proof environment needs to be created. This means a language, a context, axioms, and declarations need to be defined.

For the first of the case studies, we create a proof environment for working with natural numbers defined in terms of real numbers. Real numbers are represented by the sort R. The language of reals includes constants 0,1 and 2 of the sort R, as well as a number of functions and predicates:

```
<functions>
+ : (R,R) -> R;
- : (R,R) -> R;
* : (R,R) -> R;
abs : (R) -> R;
sin : (R) -> R
<end functions>

<predicates>
A : ();
B : ();
C : ();
= : (R,R);
< : (R,R);
<< : (R,R);
```

```
<= : (R,R)
<end predicates>
```

Predicates are defined as names followed by the corresponding arities. Here arity is a list of sorts of elements, which are the arguments of a given predicate.

With respect to functions, notice that there is no difference between the functions `+ : (R,R) -> R` and `* : (R,R) -> R` apart from the function name. The way the functions are defined in PRAWF makes them truly abstract in a sense that there is no need to define what exactly addition, subtraction, or multiplication means. The only requirement is that these definitions state what is taken as an input and what is returned. In this case, the functions `+` and `*` take two real numbers as an input and output a real number. Similarly, the function for the absolute value `abs : (R) -> R` requires a real number as an input and the output is also a real number. Being abstract, all these functions rely on axioms, which define what is the actual operation performed by a certain function.

Here are some axioms for working with real numbers. The variables x, y and z are of sort `R`:

ax1 $\forall x(\forall y(((x+y)-1)=(x+(y-1))))$

ax2 $\forall x(\forall y((y=0)\rightarrow((x+y=x))))$

ax3 $\forall z(\forall y(\forall x((y+z)<x\rightarrow y<(x-z))))$

ax4 $\forall x(\forall y(x=(x+y)-y))$

ax5 $\forall x((x-x)=0)$

ax6 $abs(0)=0$

ax7 $0<1$

ax8 $\forall x((x*0)=0)$

ax9 $\forall x(((2*x)-1)-1=2*(x-1))$

A more comprehensive list of axioms used in proofs is available in the source code [100]. All the axioms that are added as a part of a proof environment need to be Harrop.

To define what *natural numbers* are, we declare a predicate `N` as the least fixed point of the operator Φ, where $\Phi=\lambda Y\lambda z\ (z=0\vee Y(z-1))$.

Case 1. Addition of natural numbers. To extract a program for addition, we formalize the goal as

$$\forall x(N(x)\rightarrow\forall y(N(y)\rightarrow N(x+y)))$$

and prove it in PRAWF by induction.

We begin with applying $\forall^+$ and adding $N(x)$ to the list of assumptions. By IND, it suffices to show $\forall y(y = 0 \vee N(x + (y - 1)) \to N(x + y))$. Again, we use $\forall^+$ and add $y = 0 \vee N(x + (y - 1)$ to the assumptions. We need to prove 2 sub-goals (a) $y = 0 \to N(x+y)$ and (b) $(N(x+(y-1)) \to N(x+y))$.

For (a) the proof is straightforward since $y = 0$, so $N(x + y) = N(x)$, which is justified by the ax2.

For (b) the goal can be rewritten as $N((x+y)-1) \to N(x+y)$, using the congruence rule. Again, this is possible due to ax1. We add the premise of this implication to the list of assumptions and apply $\to^-$, followed by $\forall^-$ to get two new sub-goals: (c) $\forall z(z = 0 \vee N(z - 1) \to N(z))$, where z corresponds to $(x + y)$, and (d) $(x + y = 0 \vee N((x + y) - 1))$. The closure rule proves (c); (d) is proven from assumptions by applying the $\vee_r^+$ rule. This concludes the proof.

Constructing a proof in PRAWF means building up a proof tree, which corresponds to the proof steps like the ones described above. Once the proof is completed, the proof tree is printed out in PRAWF. This structure is then used to extract a corresponding program:

```
Abstpr v1
   Recpr
      Abstpr f_mu
         Abstpr a_comp
            Apppr
                  Abstpr v2
                     Casepr
                        ProgVar v2
                        *{Lt ["a_ore"]
                           ProgVar v1
}                        *{Rt ["b_ore"]
                           Apppr
                                 Abstpr v4
                                    Apppr
                                          Abstpr a_id
                                             ProgVar a_id
                                          Conpr Rt
                                             ProgVar v4
                                 ProgVar b_ore
}
                  Apppr
                        Apppr
                              Abstpr xsuby
                                 Abstpr monPv
```

```
                                        Casepr
                                           ProgVar monPv
                                           *{Lt ["a_ore"]
                                              Conpr Lt
                                                 Conpr Nil
}                                           *{Rt ["b_ore"]
                                              Apppr
                                                    Abstpr v2
                                                       Conpr Rt
                                                          Apppr
                                                                ProgVar xsuby
                                                                ProgVar v2
                                                    ProgVar b_ore
}
                              ProgVar f_mu
                        ProgVar a_comp
```

This extracted program follows the proof steps from the proof tree and even though the non-computational content is filtered out, it still contains a lot of abstractions and applications. Therefore, there is a need to normalize this program by means of beta reduction. The application of the normalization procedure generates the following simplified program:

```
Abstpr v1
   Recpr
      Abstpr f_mu
         Abstpr a_comp
            Casepr
               ProgVar a_comp
               *{Lt ["a_ore"]
                  ProgVar v1
}               *{Rt ["b_ore"]
                  Conpr Rt
                     Apppr
                           ProgVar f_mu
                           ProgVar b_ore


}
```

Here Abstpr is a label for abstraction, Recpr for recursion, Casepr for a case constructor, Apppr for application and Conpr is a label for a program constructor.

Both programs, the full and the simplified one, are equivalent in terms of the output that they generate. In this example, the time difference in execution of the full program and the simplified program is not noticeable. However, for more complex proofs and programs that are extracted from them this may not be the case. Normally, full programs need to perform a lot of extra calculations, which is avoided in case of the simplified programs.

To execute the extracted program, we first need to encode the numbers that we are going to add into their program form. We use the `numgen` function, with the assumption that the input given is either 0 or a positive integer, for example:

```
*Mode> zero = numgen 0
*Mode> two = numgen 2
*Mode> three = numgen 3
```

Here `zero` corresponds to the program $\mathbf{Lt}(\mathbf{Nil})$, `two` to the program $\mathbf{Rt}(\mathbf{Rt}(\mathbf{Lt}(\mathbf{Nil})))$ and `three` to the program $\mathbf{Rt}(\mathbf{Rt}(\mathbf{Rt}(\mathbf{Lt}(\mathbf{Nil}))))$.

One way to calculate the result of executing the extracted program is by evaluating it step by step. Evaluation in this case begins with an undefined value $\perp$. The program is called recursively until the final expected result is shown. For instance, if we run the program with the inputs 3 and 2, we expect to get the number 5, i.e., $\mathbf{Rt}(\mathbf{Rt}(\mathbf{Rt}(\mathbf{Rt}(\mathbf{Rt}(\mathbf{Lt}(\mathbf{Nil}))))))$. Here is how the program execution looks like in PRAWF. Repetitive steps are omitted for convenience.

```
*Mode> applyme "additionsp" [three,two]
Type s to run the program step by step OR
enter a number to execute the program
for a certain number of steps >> s

1  bot
..
19  Rt(bot)
..
30  Rt(Rt(bot))
31  Rt(Rt(Rt(bot)))
32  Rt(Rt(Rt(Rt(bot))))
33  Rt(Rt(Rt(Rt(Rt(bot)))))
34  Rt(Rt(Rt(Rt(Rt(Lt(bot))))))
35  Rt(Rt(Rt(Rt(Rt(Lt(Nil))))))
```

In this case the result is calculated in 35 steps. If one is to proceed with the evaluation after this, the value will remain unchanged as there are no more undefined $\perp$ elements left to evaluate.

If one does not want to evaluate step by step, PRAWF also allows checking the result of evaluation after a specified number of steps, for instance:

```
*Mode> applyme "additionsp" [three,two]
Type s to run the program step by step OR
enter a number to execute the program
for a certain number of steps >> 36
36  Rt(Rt(Rt(Rt(Rt(Lt(Nil))))))
```

Case 2. Program for reversing lists. The goal of the second case study is to extract a program, which reverses a list of objects, regardless of their data type. This is an example of a polymorphic program that goes beyond the area of computable analysis.

Firstly, the proof environment needs to be redefined. The language now contains:

- sorts t for a list, and s a single element in a list.
- a constant e for an empty list
- functions $f:(s,t)\to t$ and $g:t\to t$
- predicates A of arity (t,t,t) and $=$ of arity (t,t)

Here the predicate $A(x,y,z)$ stands for `reverse (x) ++ y = z`. In other words, concatenation of the reverse of the first list x with the second list y results in the list z. Normally, A can be defined inductively as below, where Y has arity (t,t,t), a is of the sort s, and x' is of the sort t.

$$A=\mu\lambda Y\lambda(x,y,z)(x=e\wedge y=z)\vee\exists a,x'(x=f(a,x')\wedge A(x',f(a,y),z))$$

We do not use this definition as it would create additional computational content, which is not needed in practice.

We define a predicate L for a list as the least fixed point of the operator Φ, where $\Phi=\lambda Y\lambda x(x=e\vee(\exists a(\exists b(X(a)\wedge(x=f(a,b))\wedge Y(b)))))$. Here X and a are of sort s, while Y and x' are of sort t. We also define a predicate $R=\lambda x,t\,A(x,e,z)$, so $R(x,z)=A(x,e,z)$.

List reversal can be expressed by the following formula:

$$\forall x(L(x)\to\exists z(L(z)\wedge R(x,z)))$$

In PRAWF we prove this more generally, using the following claim, where y is defined as an empty list. This helps us to obtain a faster linear program.

$$\forall x(L(x)\to\forall y(L(y)\to\exists z(L(z)\wedge A(x,y,z))))$$

Prior to starting the proof, we also add the following axioms to the proof environment.

ax1' $\forall x(A(e,x,x))$

ax2' $\forall a(\forall x(\forall y(\forall z(A(x,f(a,y),z) \to A(f(a,x),y,z)))))$, where a is of sort s and x,y,z are of sort t

The proof is by induction. After applying the IND rule, unfolding the operator Φ and using $\forall^+$ and $\to^+$ to add $x = e \vee (\exists a(\exists b(X(a) \wedge (x = f(a,b) \wedge (\forall y(L(y) \to (\exists z(L(z) \wedge A(b,y,z)))))))))$ to assumptions, it suffices to show:

$$\forall y(L(y) \to \exists z(L(z) \wedge A(x,y,z))))$$

We use $\vee^-$ rule with the newly added disjunction assumption to get its left (*branch a*) and the right (*branch b*) parts as separate assumptions.

For the *branch a*, we use $\forall^+$ and $\to^+$ again to gain a new assumption $L(y)$. The new goal is $\exists z(L(z) \wedge A(x,y,z))$. This goal is broken into two sub-goals by applying $\exists^+$ and $\wedge^+$. The first sub-goal is $L(z)$. It corresponds to the assumption. The second sub-goal is $A(x,y,z)$. It is proven by rewriting it into $A(e,y,y)$, which holds by ax1'.

For the *branch b*, multiple application of the $\exists^-$ rule in combination with of $\forall^+$ and $\to^+$ allows striping off the existential quantifiers in assumptions and arriving back at the initial goal of $\exists z(L(z) \wedge A(x,y,z))$. This time, however, there are extra useful assumptions in the context, namely:

u: $L(y)$

v: $X(a) \wedge (x = f(a,b) \wedge (\forall y(L(y) \to (\exists z(L(z) \wedge A(b,y,z))))))$

The $\exists^-$ rule is applied again with $\exists z(L(z) \wedge A(b,f(a,y),z))$. Now, there are two new sub-goals $\exists z(L(z) \wedge A(b,f(a,y),z))$ in *branch c* and $\forall z((L(z) \wedge A(b,f(a,y),z)) \to (\exists z(L(z) \wedge A(x,y,z))))$ in *branch d*.

For the *branch c*, the $\to^-$ rule is used with $L(f(a,y))$, followed by $\forall^-$ and $\wedge_r^-$, which makes the goal formula match the assumption v. To close off this branch, several minor sub-goals generated by the application of above rules need to be proven. For instance, $\to^-$ is used with $\Phi(L)(f(a,y))$ and $\forall^-$ to transform the sub-goal $L(f(a,y))$ into the form of a closure, i.e. $\forall x(\Phi(L)(x) \to L(x))$. Further smaller sub-goals are also proven using several transformation steps that make them match with the assumptions u and v.

For the *branch d*, first the quantifier is stripped off and the assumption w : $L(z) \wedge A(b,f(a,y),z)$ is added to the context. The new goal is $\exists z(L(z) \wedge A(x,y,z))$.

Again, the quantifier stripped off and the resulting conjunction is split into two sub-goals.

For $L(z)$, the $\wedge_l^-$ rule is used with $A(b, f(a,y), z)$ and resulting formula is proven by the assumption w.

For $A(x,y,z)$, the goal is rewritten as $A(f(a,b),y,z)$, using the congruence rule and then by application of the $\rightarrow^-$ rule with $A(b, f(a,y), z)$, followed by quantification, the new goal matches the ax2.

All the extra goals introduced by the implication elimination and congruence transformed into a form which allows proving them from the available assumptions. These are small transformations; the exact order of steps is available in the source code of PRAWF namely in the tactics file called revlist.txt.

When the proof is complete a corresponding program can be extracted. The simplified version of the program for list reversal is given below.

```
Recpr
   Abstpr f_mu
      Abstpr a_comp
         Casepr
            ProgVar a_comp
            *{Lt ["a_ore"]
               Abstpr v3
                  ProgVar v3
}            *{Rt ["b_ore"]
               Abstpr v5
                  Casepr
                     ProgVar b_ore
                     *{Pair ["a_elr","b_elr"]
                        Apppr
                              Apppr
                                    ProgVar f_mu
                                    ProgVar b_elr
                              Conpr Rt
                                 Conpr Pair
                                    ProgVar a_elr
                                    ProgVar v5
}
}
```

To test that the program executes correctly, we define a list that needs to be reversed as a program. The type of the elements in the list can be arbitrary but for simplicity we use a list of positive integers `[1,3,5]` as an example. We also

define an empty list as a program, i.e. **Lt**(**Nil**). This is needed in order to perform evaluation with respect to our definition of a list, i.e. it is either an empty list or a pair (a,x), where a is a single element (the head of the list) and x is a list of an arbitrary length (a tail).

This time the evaluation takes 64 steps and we get the equivalent of `[5,3,1]` printed out in the program format. We deliberately skip steps in the below to be concise.

```
*Mode> empty = listgen []
*Mode> l = listgen [1,3,5]
*Mode> applyme "revlistsp" [l,empty]
Type s to run the program step by step OR
enter a number to execute the program for a certain number of steps
>> s

1  bot
..
53  Rt(bot)
54  Rt(Pair(bot, bot))
..
64  Rt(Pair(Rt(Rt(Rt(Rt(Rt(Lt(Nil)))))),
    Rt(Pair(Rt(Rt(Rt(Lt(Nil)))),
    Rt(Pair(Rt(Lt(Nil)), Lt(Nil)))))))
```

Since we are using type-free language, it is important to understand what **Lt**(**Nil**) means in each of the cases to understand the output of this example. The first three occurrences of **Lt**(**Nil**) stand for 0, which is a base for building numbers like 5, 3 and 1. The last occurrence of **Lt**(**Nil**) refers to an empty list. In a simpler representation, Haskell-like notation, the above stands for `5:3:1:[]`.

Case 3. Conversion of natural numbers into Cauchy. While the first two examples use only induction, in order to extract a program to convert natural numbers into their Cauchy representation, the proof requires the use of both, induction and coinduction. The statement that needs to be proven is as follows.

$$\forall x(N(x) \to C(x))$$

The proof environment is an extended version of the proof environment in the first case study. The predicate N for natural numbers is defined as before and C (for Cauchy) is the greatest fixed point of $\Phi_C = \lambda X \lambda x\ \exists y(N(y) \wedge abs(x-y) < 1 \wedge X(2*x))$.

Applying the coinduction rule and unfolding the operator the new goal to prove is $\forall x(N(x) \to (\exists y\ (N(y) \wedge (abs(x-y) < 1 \wedge N(2*x)))))$. After applying the $\forall^+$ rule we add $u_1 : N(x)$ to the list of assumptions. Using the $\exists^+$ with x and $\wedge^+$, we get three new sub-goals (a), (b) and (c).

(a) $N(x)$, which is proven from the assumption u_1.

(b) $abs(x-x) < 1$ is proven by rewriting this goal to $0 < 1$, which corresponds to the axiom ax7. This rewriting requires the use of the congruence rule and the axioms ax5 and ax6.

(c) $N(2*x)$ is transformed into $\forall x(N(x) \to N(2*x))$, using the $\to^-$ and $\forall^-$.

Now we need to prove $\forall x(N(x) \to N(2*x))$. The proof is by induction. Since N is defined in terms of Φ, the definition of Φ is unfolded, creating a new goal, i.e. $\forall x((x = 0 \vee N(2*(x-1))) \to N(2*x))$. Using the combination of $\forall^+$ and $\to^+$, we add a new assumption $u_2 : x' = 0 \vee N(2*(x'-1))$ to context. Then the $\vee^-$ rule is applied with this newly added assumption, creating two new sub-goals (d) and (e).

(d) $x' = 0 \to N(2*x')$. Here the $\to^+$ rule is used to add $u_3 : x' = 0$ to the context. The new goal $N(2*x')$ is rewritten by the congruence rule into $N(0)$. Using $\to^-$ and $\forall^-$, we rewrite the goal to $\forall x(\Phi(N)(x) \to N(x))$, which is a closure. Again, a number of trivial sub-proofs need to be performed to justify the use of the congruence rule. These sub-proofs are written out in detail in `cauchy.txt`, which is one of the tactics supplied with the PRAWF distribution.

(e) $N(2*(x'-1)) \to N(2*x')$. This goal is rewritten by congruence rule into $N((2*x'-1)-1) \to N(2*x')$. The premise of this implication becomes a new assumption u_4. Using $\to^-$ with $\forall x(N(x-1) \to N(x))$, followed by $\to^+$, we add a new assumption $u_5 : \forall x(N(x-1) \to N(x))$ to the context. $N(2*x')$ is then used with u_5 to transforms this goal exactly into the form, which matches the assumption u_4.

Adding $\forall x(N(x-1) \to N(x))$ as a part of the $\to^-$ rule requires justification. Using $\forall^+$, $\forall x(N(x-1) \to N(x))$ is transformed to $N(x''-1) \to N(x'')$. The premise of this implication is added as a new assumption u_6. Again, $\to^-$ is used with $x'' = 0 \vee N(x''-1)$, followed by $\forall^-$, transforming the current goal to $\forall x''((x'' = 0 \vee N(x''-1)) \to N(x''))$, which is a closure. The remaining sub-proofs created by the use of congruence are proven by the assumptions from the context and the axiom ax9.

This concludes the proof. The extraction generates the following recursive program:

```
Recpr
   Abstpr f_nu
      Abstpr a_comp
         Conpr Pair
            ProgVar a_comp
            Apppr
                  ProgVar f_nu
                  Apppr
                        Recpr
                           Abstpr f_mu
                              Abstpr a_comp
                                 Casepr
                                    ProgVar a_comp
                                    *{Lt ["a_ore"]
                                       Conpr Lt
                                          Conpr Nil
}                                    *{Rt ["b_ore"]
                                       Conpr Rt
                                          Conpr Rt
                                             Apppr
                                                   ProgVar f_mu
                                                   ProgVar b_ore
}
                        ProgVar a_comp
```

The below is a brief illustration of the evaluation process, which generates a stream representing the real number 2. This representation shows a stream after 100 evaluation steps. Since here we work with infinite Cauchy sequences, in order to present them we print finite results after a certain number of evaluation steps. This, however, requires the use of bot for undefined parts, which can be evaluated further if we decide to continue evaluation beyond the chosen number of steps.

```
*Mode> applyme "cauchysp" [two]
Type s to run the program step by step OR
enter a number to execute the program for a certain number of steps >> s

1   bot
..
```

```
5   Pair(bot, bot)
..
7   Pair(Rt(bot), bot)
8   Pair(Rt(Rt(bot)), bot)
9   Pair(Rt(Rt(Lt(bot))), bot)
10  Pair(Rt(Rt(Lt(Nil))), bot)
..
20  Pair(Rt(Rt(Lt(Nil))), Pair(bot, Pair(bot, bot)))
..
30  Pair(Rt(Rt(Lt(Nil))),
    Pair(Rt(Rt(bot)),
    Pair(bot, Pair(bot,
    Pair(bot, bot)))))
..
50  Pair(Rt(Rt(Lt(Nil))),
    Pair(Rt(Rt(Rt(Rt(Lt(Nil))))),
    Pair(Rt(Rt(Rt(Rt(bot)))),
    Pair(Rt(Rt(bot)),
    Pair(bot, Pair(bot, Pair(bot,
    Pair(bot, bot)))))))
..
100 Pair(Rt(Rt(Lt(Nil))),
    Pair(Rt(Rt(Rt(Rt(Lt(Nil))))),
    Pair(Rt(Rt(Rt(Rt(Rt(Rt(Rt(Rt(Lt(Nil))))))))),
    Pair(Rt(Rt(Rt(Rt(Rt(Rt(Rt(Rt(bot)))))))),
    Pair(Rt(Rt(Rt(Rt(Rt(Rt(bot)))))),
    Pair(Rt(Rt(Rt(Rt(bot)))),
    Pair(Rt(Rt(bot)),
    Pair(bot, Pair(bot, Pair(bot, Pair(bot, Pair(bot, Pair(bot,
    Pair(bot, Pair(bot,
    Pair(bot, bot)))))))))))))))
..
```

In a more concise form, this can be written as nested pairs $(2,(4,(8,..)))$. Further evaluation follows the expected pattern with 16 being the next number to be evaluated.

Summary and more complex case studies. Working on these examples has shown that the proof process in PRAWF can be tedious. A number of usability improvements were introduced to speed up the interaction, however, there is still a lot that can be improved in terms of practical usability. These improvements are mentioned in the final part of this thesis.

Whilst the case studies described earlier show that the program extraction approach we developed is working, they are rather small. A more complex case study was performed by Hideki Tsuiki, whose main goal was to show that real number represented as a signed digit can be translated into its Gray code representation. The corresponding theorem was proven manually in [24]. This proof was subsequently replicated in PRAWF in the environment, which was an extended version of the language of reals that we used in the previous cases.

Both ways of representing real numbers are expressed in a form of infinite streams. Logically, they rely on coinductively defined predicates. For instance, for the predicates S and G a realizer of $S(x)$ is an infinite stream of signed digits representing x and, similarly, a realizer of $G(x)$ represents x in Gray code.

The predicate S is defined as the largest fixed point of Φ_S, where $\Phi_S \stackrel{\text{Def}}{=} \lambda X \lambda x\ \exists d(SD(d) \wedge (abs(2*x-d) \leq 1) \wedge X(2*x-d))$, and the predicate SD (for signed digit) is defined as $\lambda x((x = m \vee x = 1) \vee x = 0)$. The infinite stream, which is the realizer of $S(x)$, consists of digits like $\mathbf{Lt}(\mathbf{Lt}(\mathbf{Nil}))$ for -1, $\mathbf{Lt}(\mathbf{Rt}(\mathbf{Nil})))$ for 1 and $\mathbf{Rt}(\mathbf{Nil})$ for 0. Each of these digits are actually realizers of $SD(y)$ for the corresponding signed digit y. Streams are presented in the form of infinitely nested pairs, for instance the real number -0.5 is written as $\mathbf{Lt}(\mathbf{Lt}(\mathbf{Nil})) : \mathbf{Rt}(\mathbf{Nil}) : \mathbf{Rt}(\mathbf{Nil}) : \ldots$, that is $-1 : 0 : 0 : \ldots$, where $a : b$ stands for $\mathbf{Pair}(a, b)$.

Gray code representation is special in a sense that a realizer of $G(x)$ is a stream that may have one element not defined. The predicate G is the greatest fixed point of Φ_G, defined as $\lambda X \lambda x\ (m \leq x \wedge x \leq 1) \wedge (D(x) \wedge X(t(x)))$. Here t is a tent function, which when applied to x is defined as $1 - 2 * abs(x)$. The predicate D is defined as $\lambda x\ (\neg(x = 0)) \rightarrow B(x)$, where $B = \lambda x\ (x \leq 0 \vee 0 \leq x)$. The way that D is defined allows us to admit a program as a realizer, even if its value is not defined like in case of an infinitely looping program. This is possible when the premise in D is false, that is when $x = 0$.

In order to extract a program for converting of a signed digit stream into a Gray code stream in PRAWF the statement $\forall x(S(x) \rightarrow G(x))$ needs to be proven. This proof is complex and involves a multitude of sub-proofs. The tactic for this proof is called `gray-stog` and should be run with the `gray` environment. This proof is included with PRAWF distribution. The most interesting points of the proof are highlighted in [22]. Should the reader be interested, the distribution also contains a few more tactics, which can be used to extract some simpler programs.

Part IV

Conclusions

Contents

The journey **163**
Summary **164**
Future work **165**

The journey

The journey that brought me to this point was not a straightforward one. In my first year I focused on natural language processing instead of program extraction. I was aiming to find a way of parsing proofs written in (a subset of) natural language efficiently in order to convert them into formal proofs in a proof assistant.

Natural language processing is not a new topic, however, this specific aspect of using it with mathematical proofs does not have a lot of related research apart from the Naproche project [41]. This project looks at a subset of natural language, Naproche CNL (controlled natural language), which is used for checking existing mathematical texts. My pursuit, on the other hand, was directed towards applications of natural language processing whilst mathematical content is being created. Consequently, a prototype system *Script* was developed. This system used a language that, although being formal, looked very close to natural language. *Script* used the basic PRAWF machinery to check the proofs and automatically deduce solutions. The idea was to use it on its own or when writing papers. For instance, if one were to write a proof using LaTeX , then *Script* would "read" this proof and check it for correctness. I presented the prototype at the *Proof, Computation, Complexity 2017* workshop.

In my second year, after the realization that the scope of my original research was too large and with the growing interest in program extraction, the direction of my research changed. This shift led to transformation of the previously developed proof assistant that I used before into a prototype tool for program extraction, which is now known as PRAWF.

PRAWF is a testing ground for various ideas and a good base for building numerous extensions. For instance, while this thesis is being written, my fellow research student is extending PRAWF to work with sequent calculus. PRAWF allows us to present IFP in action. Compared with the well-established proof assistants like *Coq*, *Isabel* or *Minlog*, our tool does not have enough strength in terms of the number of already proven theorems, usability, and certain additional functionalities. However, it is not meant to compete with them but rather

be a simple way of presenting how IFP can be implemented without getting distracted by the specifics of the established systems, which are not straightforwardly compatible with IFP. In fact, at a certain point I attempted to implement IFP in *Coq*. However, I encountered complications due to bound variables renaming. Hence, the decision was made to continue working on PRAWF instead to achieve more functionalities within the limited time. Implementation of IFP in *Coq* is worth revisiting in the future, provided there is sufficient time to tackle these complications.

Summary

This thesis developed a new approach to program extraction from proofs, which uses inductive and coinductive definitions. After reviewing the existing approaches of obtaining correct programs, with a specific focus on intuitionism and realizability interpretation, we set a semantic foundation for our approach and performed a rigorous proof of correctness.

The key system, IFP, served as a base of this research, allowing construction of proofs about abstract mathematical objects. An enhanced realizability interpretation based on Kleene's original research was used to enable extraction of computational content from these proofs. The distinction between Harrop and non-Harrop formulas allowed separating computationally meaningful constructs in a proof from the computationally irrelevant content. This thesis also gave an overview of the existing developments in the area of program extraction. IFP and PRAWF were specifically inspired by the *Minlog* system. The main difference is in the way induction and coinduction are defined in both systems as well as slightly different treatment of Harrop formulas. While in *Minlog* they are essentially treated as nc formulas, in our system there is distinction between nc formulas and Harrop formulas as their subset.

This thesis includes the following main contributions:

1. a detailed proof of soundness,

2. an alternative approach to operational semantics, which uses extended closures to avoid unnecessary substitution, and

3. development of a proof assistant for program extraction based on IFP.

The first part covered the theoretical framework and background related to program extraction.

The second part presented IFP, IFP' and RIFP, as well as the proof of soundness. The Soundness Theorem originally proven in [21] showed a direct link

between IFP and RIFP. However, this version of the Soundness Theorem was not included here as it is a restricted version, which requires predicates to be admissible. Instead, a detailed proof of the redefined theorem, drafted in [24], was included. This version of the Soundness Theorem links IFP' with RIFP. However, since IFP proofs can be "translated" into IFP' proofs, this theorem also shows that IFP is sound. The thesis also included a brief presentation of the denotational semantics and a larger overview of operational semantics, showing the adequacy between them. The Adequacy Theorem presented in this thesis is a variant of the Adequacy Theorem proven in [24], adjusted to match the updated operational semantics in order to show that even after the update the adequacy remained.

The third part of the thesis included a general description of the structure of PRAWF. Here several case studies were presented to showcase program extraction from IFP proofs in PRAWF. The reader is welcome to try this tool for themselves. The relevant software is available on Bitbucket [100]. The user manual and the tutorial are available at the official PRAWF website [101]. The tutorial follows the first one of the case studies.

PRAWF is a prototype system and requires improvements in order to become as powerful as the well-established proof assistants. Although we did not investigate how the efficiency of the programs extracted in PRAWF compares to that of the programs extracted in other tools, we did show that IFP works in practice, which was the main purpose of such an implementation. Successful program extraction leads us believe that it is viable to implement IFP in a more powerful proof assistant.

Future work

There are various aspects, both theoretical and practical, that can be used to extend and improve this research.

From the theoretical perspective, providing formal proofs of the Soundness and the Adequacy Theorems within one of the established systems will help to improve trust in IFP. Apart from that, since all axioms and rules of IFP (and RIFP) are true w.r.t. the classical notion of model, IFP does not contradict classical mathematics.

From the practical point of view, the use of Skolem functions for adding axioms should be considered. Skolem functions provide a systematic and controlled way of introducing new functions with prescribed (non-computational) properties. The Skolem scheme guarantees that such a new function with the prescribed property actually does exist in a classical model (for the existence

one uses the Axiom of Choice though). Hence, the use of Skolem functions could help us to ensure that the axioms added to PRAWF's proof environment are indeed meeting the requirement of being non-computational.

Another task would be updating the implementation to include a proof of correctness with each extracted program. This is done in other systems, e.g. *Minlog*, and the implementation should not be too complicated. This means that the proof of the Soundness Theorem needs to be implemented.

To make our system more applicable and scalable, the theorem database needs to be extended. This may include introduction of the notion of a *Theory*, which would include a set of theorems combined with the appropriate language, declarations and axioms.

As mentioned earlier, the signed digit representation into infinite Gray code conversion was one of the more complex case studies. However, it is only the half of a potentially bigger case study, exploring the reverse conversion from infinite Gray code conversion into the signed digit representation. This direction is more complicated and requires further extension of IFP and additional research into concurrency.

Another challenging area is linear algebra with exact real numbers, in particular matrix inversion. Usually matrix inversion is done using Gaussian elimination, which picks out a column in the matrix, which is not zero. This is a pivot problem because one cannot test if a real number is zero or not. For instance, if we need to solve two linear equations $ax+by=c$ and $dx+ey=f$. Naively we could try to solve the first equation for x, i.e., $x=(c-by)/a$. However, such an approach can only work if a is not zero. Therefore, one needs to find one of the coefficients, which is not zero. However, it is not possible to decide which one is definitively not a zero. For instance, in case the coefficients are given as Cauchy sequences, converging to very small values, such that it is hard to see if they are zero or not. Therefore, all coefficients need to be worked on in parallel until one is found to be non-zero. This requires non-determinism, which makes program extraction non-trivial.

Extraction of a SAT solver was previously done in *Minlog* in [20]. The way induction and coinduction are defined in *Minlog* required certain workarounds to make extraction efficient. Therefore, it is an interesting case to explore within IFP to see whether the extraction process can be more straightforward.

Additionally, from the usability perspective, the interface of the system needs to be improved to be more accessible. The issue that we encountered so far is that when GHCi updates certain minor functionalities, which are not directly related to IFP, PRAWF may stop working. Consequently, the system needs to be monitored to ensure that the users of the newer versions of GHCi can work with

the system. Another aspect is the need to add more flexibility in the execution mode. This may include introduction of an alternative way to add new data types seamlessly, without modifying the source code.

To sum up, PRAWF is a fairly new system. Hence, during its development we focused mainly on the core functionalities. This system is aimed to be an advocate for IFP in terms of bringing the theory behind IFP into the practical realm.

Bibliography

[1] S. Abramsky and A. Jung. “Domain Theory”. In: *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. USA: Oxford University Press, Inc., 1995, 1 — 168.

[2] *Agda website*. Accessed 26/07/2020. URL: `https://agda.readthedocs.io/en/v2.6.1.1/overview.html`.

[3] S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. “The Nuprl Open Logical Environment”. In: Dec. 2006, pp. 170 –176.

[4] M. van Atten and G. Sundholm. *L.E.J. Brouwer’s ‘Unreliability of the logical principles’. A new translation, with an introduction*. Nov. 2015. arXiv: `1511.01113 [math.HO]`.

[5] J. Avigad and S. Feferman. “Gödel’s functional (“Dialectica”) interpretation”. In: *Handbook of proof theory* 137 (1998), pp. 337 –405.

[6] J. Avigad, H. Towsner, et al. “Functional interpretation and inductive definitions”. In: *Journal of Symbolic Logic* 74.4 (2009), pp. 1100–1120.

[7] M. Baaz, S. Hetzl, A. Leitsch, C. Richter, and H. Spohr. “Cut-Elimination: Experiments with CERES”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by F. Baader and A. Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 481–495.

[8] M. Baaz, S. Hetzl, A. Leitsch, C. Richter, and H. Spohr. “Proof Transformation by CERES”. In: *Mathematical Knowledge Management*. Ed. by J. M. Borwein and W. M. Farmer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 82–93.

[9] M. Baaz and A. Leitsch. *Methods of cut-elimination*. Vol. 34. Springer Science & Business Media, 2011.

[10] H. P . Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.

[11] H. P. Barendregt. “Lambda Calculi with Types”. In: *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*. USA: Oxford University Press, Inc., 1993, 117–309.

[12] J.L. Bates. *A Logic for Correct Program Development*. TR 79-388. Cornell University, 1979.

[13] A. Bauer and I. Kavkler. "A constructive theory of continuous domains suitable for implementation". In: *Annals of Pure and Applied Logic* 159.3 (2009). Joint Workshop Domains VIII — Computability over Continuous Data Types, Novosibirsk, September 11–15, 2007, pp. 251 –267.

[14] A. Bauer, G.D. Plotkin, and D.S. Scott. "Cartesian closed categories of separable Scott domains". In: *Theor. Comput. Sci.* 546 (2014), pp. 17–29.

[15] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. "Proof theory at work: Program development in the Minlog system". In: *Automated Deduction - A Basis for Applications II*. Ed. by W. Bibel and P.H. Schmitt. 1998, pp. 41 –71.

[16] U. Berger. "Realisability for Induction and Coinduction with Applications to Constructive Analysis". In: *Journal of Universal Computer Science* 16.18 (2010), pp. 2535–2555.

[17] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. "Program Extraction from Normalization Proofs". In: *Studia Logica: An International Journal for Symbolic Logic* 82.1 (2006), pp. 25–49.

[18] U. Berger, J. Blanck, and P.K. Køber. "Domain representations of spaces of compact subsets". In: *Mathematical Structures in Computer Science* 20.2 (2010), 107–126.

[19] U. Berger and T. Hou. "A realizability interpretation of Church's simple theory of types". In: *Mathematical Structures in Computer Science* (2016), pp. 1 –22.

[20] U. Berger, A. Lawrence, F. Nordvall Forsberg, and M. Seisenberger. "Extracting verified decision procedures: DPLL and Resolution". In: *Logical Methods in Computer Science* 11.1 (2015).

[21] U. Berger and O. Petrovska. "Optimised Program Extraction for Induction and Coinduction". In: *Sailing Routes in the World of Computation: 14th Conference on Computability in Europe, CiE 2018, Kiel, Germany, July 30 – August 3*. 2018, pp. 70 –80.

[22] U. Berger, O. Petrovska, and H. Tsuiki. "PRAWF: An Interactive Proof System for Program Extraction". In: *Beyond the Horizon of Computability*. Ed. by M. Anselmo, G. Della Vedova, F. Manea, and A. Pauly. Cham: Springer International Publishing, 2020, pp. 137–148.

[23] U. Berger and M. Seisenberger. "Proofs, programs, processes". In: *Theory of Computing Systems* 51.3 (2012), pp. 313–329.

[24] U. Berger and H. Tsuiki. “Intuitionistic Fixed Point Logic”. In: *Annals of Pure and Applied Logic* 172.3 (2021).

[25] S. Berghofer. “Extracting a Normalization Algorithm in Isabelle/HOL”. In: *Proceedings of the 2004 International Conference on Types for Proofs and Programs*. TYPES’04. Jouy-en-Josas, France: Springer-Verlag, 2006, pp. 50–65.

[26] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. “First steps in synthetic guarded domain theory: step-indexing in the topos of trees”. In: *Logical Methods in Computer Science* Volume 8, Issue 4 (Oct. 2012).

[27] G. Birkhoff. *Lattice Theory*. American Mathematical Society colloquium publications v. 25, pt. 2. American Mathematical Society, 1940.

[28] E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill Book Company, 1967.

[29] E. Bishop. *Schizophrenia in Contemporary Mathematics*. American Mathematical Society, 1973.

[30] J. Blanck. “Interval Domains and Computable Sequences: A Case Study of Domain Reductions”. In: *The Computer Journal* 56.1 (Sept. 2012), pp. 45–52.

[31] J. Blanck, V. Stoltenberg-Hansen, and J.V. Tucker. “Domain representations of partial functions, with applications to spatial objects and constructive volume geometry”. In: *Theoretical Comp. Sci.* 284.2 (2002), pp. 207 –240.

[32] D. Bridges and E. Palmgren. “Constructive Mathematics”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2018. Metaphysics Research Lab, Stanford University, 2018.

[33] L.E.J. Brouwer. “1907 - On the Foundations of Mathematics”. In: *Philosophy and Foundations of Mathematics*. Ed. by A. Heyting. Vol. 1. North-Holland, 1975, pp. 11 –101.

[34] L.E.J. Brouwer. “1908 B - On the Foundations of Mathematics”. In: *Philosophy and Foundations of Mathematics*. Ed. by A. Heyting. Vol. 1. North-Holland, 1975, pp. 105 –106.

[35] L.E.J. Brouwer. *Brouwer’s Cambridge lectures on intuitionism*. Ed. by D. van Dalen. Cambridge University Press, 1981.

[36] C.M. Chuang. “Extraction of programs for exact number computation using Agda”. Accessed 15/05/2020. PhD thesis. Swansea University, 2011. URL: https://cronfa.swan.ac.uk/Record/cronfa42274.

[37] A. Church. “A Formulation of the Simple Theory of Types”. In: *J. Symbolic Logic* 5.2 (1940).

[38] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. USA: Prentice-Hall, Inc., 1986.

[39] R.L. Constable and J.L. Bates. *The nearly ultimate pearl*. Tech. rep. Cornell University, 1983.

[40] T. Coquand and G. Huet. “The calculus of constructions”. In: *Information and Computation* 76.2 (1988), pp. 95 –120.

[41] M. Cramer, B. Fisseni, P. Koepke, D. Kühlwein, B. Schröder, and J. Veldman. “The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts”. In: *Controlled Natural Language: Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8-10, 2009. Revised Papers*. Ed. by N.E. Fuchs. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[42] H. B. Curry. “Functionality in Combinatory Logic”. In: *Proceedings of the National Academy of Sciences* 20.11 (1934).

[43] J. Diller and A. S Troelstra. “Realizability and intuitionistic logic”. In: *Foundations: Logic, Language, and Mathematics*. Springer, 1984, pp. 253–282.

[44] A. Edalat and A. Lieutier. “Domain theory and differential calculus (functions of one variable)”. In: *Mathematical Structures in Computer Science* 14 (Dec. 2004), pp. 771–802.

[45] A. Edalat, A. Lieutier, and D. Pattinson. “A Computational Model for Multi-variable Differential Calculus”. In: *Foundations of Software Science and Computational Structures*. Ed. by V. Sassone. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 505–519.

[46] A. Edalat and P. Sünderhauf. “A domain-theoretic approach to computability on the real line”. In: *Theoretical Computer Science* 210.1 (1999). Real Numbers and Computers, pp. 73 –98.

[47] T. Erker, M.H. Escardo, and K. Keimel. “The way-below relation of function spaces over semantic domains”. In: *Topology and its Applications* 89 (Nov. 1998), pp. 61–74.

[48] Y. L. Ershov. *Theory of numberings (Teoriya numeratsiy)*. Nauka, 1977.

[49] M.H. Escardo and R.C. Flagg. “Semantic Domains, Injective Spaces and Monads: Extended Abstract”. In: *Electronic Notes in Theoretical Computer Science* 20 (1999). MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference, pp. 229 –244.

[50] M.H. Escardo and Ho W.K. “Operational domain theory and topology of sequential programming languages”. In: *Information and Computation* 207.3 (2009), pp. 411 –437.

[51] S. Feferman. “Constructive Theories of Functions and Classes”. In: *Logic Colloquium ’78*. Ed. by M. Boffa, D. van Dalen, and K. Mcaloon. Vol. 97. Studies in Logic and the Foundations of Mathematics. Elsevier, 1979, pp. 159 –224.

[52] M. Felleisen and R. Hieb. “The Revised Report on the Syntactic Theories of Sequential Control and State”. In: *Theoretical Computer Science* 103 (1992), pp. 235–271.

[53] R. W. Floyd. “Assigning Meanings to Programs”. In: *Proceedings of Symposia in Applied Mathematics* 19 (1967).

[54] H. Friedman. “Classically and intuitionistically provably recursive functions”. In: *Higher Set Theory*. Ed. by G. H. Müller and D. S. Scott. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 21–27.

[55] J. Friedman. “Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. Summaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd edn., Communications Research Div., Institute for Defense Analyses, Princeton, N. J., 1960, pp. 3–50. 3a-45a.” In: *Journal of Symbolic Logic* 28.4 (1963), 289–290.

[56] H. Geuvers. “Proof assistants: History, ideas and future”. In: *Sadhana* 34.1 (2009).

[57] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. 2003.

[58] J-Y. Girard. “Une extension de l’interprétation de Gödel à l’analyse, et son application à l’élimination des coupures dans l’analyse et la théorie des types”. In: *Proceedings of the Second Scandinavian Logic Symposium*. Ed. by J.E. Fenstad. 1971, pp. 63–92.

[59] K. Gödel. “Über eine bischer noch nicht benützte Erweiterung des finiten Standpunktes”. In: *Dialectica* 12.3-4 (1958), pp. 280–287.

[60] S. Goto. “Program Synthesis from Natural Deduction Proofs”. In: *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI’79. Tokyo, Japan: Morgan Kaufmann Publishers Inc., 1979, 339–341. ISBN: 0934613478.

[61] C.A. Gunter and D.S. Scott. “Semantic Domains”. In: *Formal Models and Semantics*. Ed. by Jan Van Leeuwen. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, 1990, pp. 633 –674.

[62] R. Harrop. “On disjunctions and existential statements in intuitionistic systems of logic”. In: *Mathematische Annalen* 132.4 (1956), pp. 347–361.

[63] S. Hayashi and H. Nakano. *PX: A Computational Logic*. Cambridge, MA, USA: MIT Press, 1989.

[64] H. Herbelin. “An Intuitionistic Logic that Proves Markov’s Principle”. In: *2010 25th Annual IEEE Symposium on Logic in Computer Science*. 2010, pp. 50–56.

[65] A Heyting. *Die formalen Regeln der intuitionistischen Logik*. Preussische Akademie für Wissenschaften, Phys.-math. Klasse, 1930.

[66] C.A.R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–583.

[67] *Isabelle*. Accessed on 25/05/2020. URL: https://isabelle.in.tum.de/.

[68] T. de Jong and M. H. Escardó. *Domain Theory in Constructive and Predicative Univalent Foundations*. 2020. arXiv: 2008.01422 [math.LO].

[69] G. Kahn. “Natural semantics”. In: *STACS 87*. Ed. by Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 22–39.

[70] K. Keimel. “Domain Theory its Ramifications and Interactions”. In: *Electronic Notes in Theoretical Computer Science* 333 (2017). The Seventh International Symposium on Domain Theory and Its Applications (ISDT), pp. 3 –16.

[71] S. C. Kleene. “On the Interpretation of Intuitionistic Number Theory”. In: *The Journal of Symbolic Logic* 10.4 (1945).

[72] S.C. Kleene. *The Foundations of Intuitionistic Mathematics*. Amsterdam: North-Holland Pub. Co., 1965.

[73] U. Kohlenbach. “Gödel’s functional interpretation and its use in current mathematics”. In: *dialéctica* 62.2 (2008), pp. 223–267.

[74] G. Kreisel. “Interpretation of Analysis by Means of Constructive Functionals of Finite Types”. In: *Constructivity in Mathematics*. Ed. by A. Heyting. Amsterdam: North-Holland Pub. Co., 1959, pp. 101–128.

[75] J-L. Krivine. “Classical logic, storage operators and second-order lambda-calculus”. In: 68 (1–3 1994), pp. 53–78.

[76] J-L. Krivine. “Dependent choice, ‘quote’ and the clock”. In: 308 (2003), pp. 259–276.

[77] J-L. Krivine. “Typed lambda-calculus in classical Zermelo-Fraenkel set theory”. In: 40 (2001), pp. 189–205.

[78] J-L. Krivine and M. Parigot. “Programming with proofs”. In: *EIK* 26 (1990), pp. 149–167.

[79] D. Leivant. “Reasoning about functional programs and complexity classes associated with type disciplines”. In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. 1983, pp. 460–469.

[80] J. Lockwood. *Nuprl. An Open Logical Programming Environment: a Practical Framework for Sharing Formal Models and Tools. Program Extraction*. Accessed 25/08/2020. 1998. URL: `http://www.nuprl.org/html/LPEPage/proposal/node28.html`.

[81] Z. Manna and R. Waldinger. “A Deductive Approach to Program Synthesis”. In: *ACM Trans. Program. Lang. Syst.* 2 (Jan. 1980), pp. 90–121.

[82] P. Martin-Lof. *Notes on Constructive Mathematics*. Almqvist and Wiksell, 1968.

[83] P. Martin-Löf. “Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions”. In: *Proceedings of the Second Scandinavian Logic Symposium*. Ed. by J.E. Fenstad. Vol. 63. Studies in Logic and the Foundations of Mathematics. Elsevier, 1971, pp. 179 –216.

[84] M. McKubre-Jordens. *Constructive Mathematics*. In: The Internet Encyclopedia of Philosophy. Accessed on 25/04/2020. URL: `https://iep.utm.edu/con-math`.

[85] E. Menzler-Trott. *Logic’s Lost Genius*. History of Mathematics. American Mathematical Society, 2016.

[86] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[87] F. E. Miranda-Perea. “Realizability for monotone and clausular (Co)inductive definitions”. In: *Electronic Notes in Theoretical Computer Science*. Vol. 123. 2005, pp. 179–193.

[88] K. Miyamoto. “Program Extraction from Coinductive Proofs and its Application to Exact Real Arithmetic”. PhD thesis. 2013.

[89] R. E Møgelberg, L. Birkedal, and G. Rossolini. “Synthetic domain theory and models of linear Abadi Plotkin logic”. In: *Annals of Pure and Applied Logic* 155.2 (2008), pp. 115 –133.

[90] J Moschovakis. “Intuitionistic Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2018. Metaphysics Research Lab, Stanford University, 2018.

[91] A. Nogin. “Writing Constructive Proofs Yielding Efficient Extracted Programs”. In: *Electronic Notes in Theoretical Computer Science* 37 (2000), pp. 1 –17.

[92] P. Oliva. “An analysis of Gödel’s Dialectica interpretation via linear logic”. In: *Dialectica* 62.2 (2008), pp. 269–290.

[93] P. Oliva. “Computational interpretations of classical linear logic”. In: *International Workshop on Logic, Language, Information, and Computation*. Springer. 2007, pp. 285–296.

[94] J. van Oosten. *Realizability: a historical essay*. Accessed on 10/03/2020. 2000. URL: https://dspace.library.uu.nl/bitstream/handle/1874/1945/1131.pdf?sequence=1&isAllowed=y.

[95] C. Parent. “Developing certified programs in the system Coq the program tactic”. In: *Types for Proofs and Programs*. Ed. by H. Barendregt and T. Nipkow. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 291–312.

[96] M. Parigot. “$\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction”. In: *Logic Programming and Automated Reasoning*. Ed. by A. Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 190–201.

[97] M. Parigot. “Recursive programming with proofs”. In: *Theoretical Computer Science* 94.2 (1992), pp. 335–356.

[98] C. Paulin-Mohring. “Extraction de programmes dans le Calcul des Constructions”. PhD thesis. 1989.

[99] G. Plotkin. “A Structural Approach to Operational Semantics”. In: *J. Log. Algebr. Program.* 60-61 (July 2004), pp. 17–139.

[100] PRAWF *source code*. Accessed on 20/09/2020. URL: https://bitbucket.org/olga_swansea/prawf/src/master/.

[101] PRAWF *website*. Accessed 26/09/2020. Swansea University. URL: https://prawftree.wordpress.com/.

[102] H. Rasiowa and R. Sikorski. "On existential theorems in non-classical functional calculi". In: *Fundamenta mathematicae* 41.1 (1955), pp. 21–28.

[103] M. Sato. "Classical Brouwer-Heyting-Kolmogorov interpretation". In: *Algorithmic Learning Theory*. Ed. by M. Li and A. Maruoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 176–196.

[104] M. Sato. "Towards a Mathematical Theory of Program Synthesis". In: *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'79. Tokyo, Japan: Morgan Kaufmann Publishers Inc., 1979, 757–762.

[105] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. USA: William C. Brown Publishers, 1986.

[106] H. Schwichtenberg. "Definierbare Funktionen im Lambda-Kalkül mit Typen". In: Arch. Math Logik, 1976, pp. 113–114.

[107] H. Schwichtenberg. "Dialectica interpretation of well-founded induction". In: *Mathematical Logic Quarterly* 54.3 (2008), pp. 229–239.

[108] H. Schwichtenberg and S. S. Wainer. *Proofs and Computations*. Perspectives in Logic. Cambridge University Press, 2011.

[109] D.S. Scott. "A type-theoretical alternative to ISWIM, CUCH, OWHY". In: *Theoretical Computer Science* 121.1 (1993), pp. 411 –440.

[110] D.S. Scott. "Data types as lattices". In: *ISILC Logic Conference*. Ed. by G. H. Müller, A. Oberschelp, and K. Potthoff. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 579–651.

[111] D.S. Scott. *Outline of a Mathematical Theory of Computation*. Programming Research Group, Oxford University Computing Laboratory. Oxford University Computing Laboratory, 1970.

[112] M.B. Smyth. *Category Theoretic Solution of Recursive Domain Equations*. The University of Warwick Theory of Computation Report no. 14. 1976.

[113] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains*. USA: Cambridge University Press, 1994.

[114] C. Strachey. *The varieties of programming language*. Oxford University Computing Laboratory, Programming Research Group, 1973.

[115] M. Tatsuta. "Realizability of Monotone Coinductive Definitions and Its Application to Program Synthesis". In: (1998), pp. 338–364.

[116] T. Teitelbaum and T. Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment". In: *Commun. ACM* 24.9 (1981), 563–573.

[117] *The Coq Proof Assistant*. Accessed 20/04/2020. URL: `https://coq.inria.fr/`.

[118] *The Minlog System*. Accessed 27/05/2020. URL: `http://www.mathematik.uni-muenchen.de/~logik/minlog/index.php`.

[119] H. Towsner. “A worked example of the functional interpretation”. In: *arXiv preprint arXiv:1503.05572* (2015).

[120] A. Troelsta. “Realizability and Functional Interpretations”. In: *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Vol. 344. 1. Springer-Verlag Berlin Heidelberg, 1973, pp. 175–275.

[121] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. 2nd ed. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000.

[122] T. Uustalu and V. Vene. “Least and greatest fixed points in intuitionistic natural deduction”. In: *Theoretical Computer Science*. Vol. 272. 1-2. 2002, pp. 315–339.