Engineering Formal Systems in Constructive Type Theory

Steven Schäfer

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
an der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

Saarbrücken, 2019

Berichterstatter:
Prof. Dr. Gert Smolka, Universität des Saarlandes
Prof. Dr. Derek Dreyer, Max Planck Institute for Software Systems

Dekan:
Prof. Dr. Sebastian Hack

Prüfungsauschuss:
TBD.

Tag des Kolloquiums:
TBD.

Textfassung vom 3. März 2019
© 2019 Steven Schäfer

Abstract

This thesis presents a practical methodology for formalizing the meta-theory of formal systems with binders and coinductive relations in constructive type theory.

While constructive type theory offers support for reasoning about formal systems built out of inductive definitions, support for syntax with binders and coinductive relations is lacking. We provide this support.

We implement syntax with binders using well-scoped de Bruijn terms and parallel substitutions. We solve substitution lemmas automatically using the rewriting theory of the σ-calculus. We present the Autosubst library to automate our approach in the proof assistant Coq.

Our approach to coinductive relations is based on an inductive tower construction, which is a type-theoretic form of transfinite induction. The tower construction allows us to reduce coinduction to induction. This leads to a symmetric treatment of induction and coinduction and allows us to give a novel construction of the companion of a monotone function on a complete lattice.

We demonstrate our methods with a series of case studies. In particular, we present a proof of type preservation for CC_ω, a proof of weak and strong normalization for System F, a proof that systems of weakly guarded equations have unique solutions in CCS, and a compiler verification for a compiler from a non-deterministic language into a deterministic language.

All technical results in the thesis are formalized in Coq.

Zusammenfassung

In dieser Dissertation beschreiben wir praktische Techniken um Formale Systeme mit Bindern und koinduktiven Relationen in Konstruktiver Typtheorie zu implementieren.

Während Konstruktive Typtheorie bereits gute Unterstützung für Induktive Definition bietet, gibt es momentan kaum Unterstützung für syntaktische Systeme mit Bindern, oder koinduktiven Definitionen.

Wir kodieren Syntax mit Bindern in Typtheorie mit einer de Bruijn Darstellung und zeigen alle Substitutionslemmas durch Termersetzung mit dem σ-Kalkül. Wir präsentieren die Autosubst Bibliothek, die unseren Ansatz im Beweisassistenten Coq implementiert.

Für koinduktive Relationen verwenden wir eine induktive Turmkonstruktion, welche das typtheoretische Analog zur Transfiniten Induktion darstellt. Auf diese Art erhalten wir neue Beweisprinzipien für Koinduktion und eine neue Konstruktion von Pous' "companion" einer monotonen Funktion auf einem vollständigen Verband.

Wir validieren unsere Methoden an einer Reihe von Fallstudien.

Alle technischen Ergebnisse in dieser Dissertation sind mit Coq formalisiert.

Acknowledgements

This thesis represents several years of work on disparate topics. It would not have been possible without the support of a number of people.

First and foremost I want to thank my advisor Gert Smolka for his constant support and trust. Thank you, for allowing me so much freedom to explore different topics over the years.

I also want to thank my current and former colleagues in the programming systems lab: Chad Brown, Christian Doczkal, Yannick Forster, Jonas Kaiser, Dominik Kirst, Fabian Kunze, Sigurd Schneider, Kathrin Stark, and Tobias Tebbi. Special thanks go to Tobias Tebbi, for being my office mate, for the long collaboration on Autosubst, and for coqdocjs. More special thanks go to Yannick Forster, for his coqtheorem LaTeX package.

I also want to take this opportunity to thank my parents, Beate and Frank, for all of their support throughout my life. Finally, I want to thank my wife Sahar for her unwavering support, love, and patience over the years.

Contents

1 Introduction **1**
- 1.1 Structure of the Thesis 5
- 1.2 Contributions 7
- 1.3 Published Results 8

2 Preliminaries **9**
- 2.1 Type Theory 9
- 2.2 Notations 10
- 2.3 Abstract Reduction Systems 11
- 2.4 Discussion 14

3 De Bruijn Representation of Syntax with Binders **17**
- 3.1 De Bruijn Representation of λ-Terms 17
- 3.2 Equational Theory of Substitutions 20
- 3.3 Case Study: Substitutivity of $\beta\eta$-Reduction 23
- 3.4 Discussion 24

4 Subject Reduction in CC_ω **27**
- 4.1 Syntax and Semantics of CC_ω 27
- 4.2 The Church-Rosser Theorem for CC_ω 30
- 4.3 Properties of Subtyping 34
- 4.4 Properties of Pretyping Judgments 36
- 4.5 Type Soundness and Subject Reduction 39
- 4.6 Discussion 42

5 Normalization in System F **45**
- 5.1 Polymorphic Types and Models 46
- 5.2 Syntax and Semantics of System F_{cbv} 47
- 5.3 Weak Normalization of System F_{cbv} 50
- 5.4 Syntax and Semantics of System F 53

5.5 Strong Normalization of System F 55
5.6 Discussion 59

6 Tower-based Companions for Coinduction **65**
6.1 Complete Lattices and Fixed-Points 65
6.2 The Tower-Based Companion Construction 70
6.3 Parameterized Tower Induction 74
6.4 Linearity and Well-Foundedness of the Tower Construction 76
6.5 Companions of Cocontinuous Functions 79
6.6 The Companion as a Greatest Fixed-Point 82
6.7 Discussion 84

7 Strong Bisimilarity in CCS **89**
7.1 Syntax and Semantics of CCS 89
7.2 Congruence Properties of Relative Bisimilarity 93
7.3 Properties of Strong Bisimilarity 97
7.4 Discussion 98

8 Axiomatic Semantics for Compiler Verification **101**
8.1 (Co-)Directed (Co-)Induction 103
8.2 Abstract Axiomatic Semantics 107
8.3 Guarded Commands 112
8.4 Imperative Continuations 118
8.5 Axiomatic Semantics of IC 119
8.6 Operational Correctness Statements 124
8.7 Translation from GC to IC 125
8.8 Discussion 131

9 Autosubst: Automation for de Bruijn Substitutions **135**
9.1 Automating de Bruijn Substitutions in Coq 136
9.2 Traversable Containers 140
9.3 Heterogeneous Substitutions 141
9.4 Automation Tactics 143
9.5 Related Work 143
9.6 Discussion 144

Bibliography **147**

A Representing Syntax with Binders **157**

Introduction

1

We present a practical methodology for formalizing the meta-theory of formal systems with binders and coinductive predicates in constructive type theory. Our approach to syntax with binders is based on the combination of (well-scoped) de Bruijn terms and parallel substitutions [30], with operations from the σ-calculus of explicit substitutions [1]. We implement this approach with the AUTOSUBST library for Coq [106]. Our approach to coinduction is based on an inductive tower construction and the companion construction of Pous [99]. We demonstrate the effectiveness of our approach with several case studies. All technical results are formalized using the Coq proof assistant [121] under the assumptions of functional and propositional extensionality.

A formal system consists of a collection of inductive types of terms along with inductive and coinductive relations over terms. Examples of formal systems include proof systems, such as Gentzen's inference rules for first-order logic [50], the static and dynamic semantics of programming languages [57], and process calculi [82]. In this thesis we consider the untyped λ-calculus [32], System F [51, 102], CC_ω [78], CCS [82] with recursive processes, and two idealized imperative programming languages GC [41] and IC [109].

We are interested in proving global properties of formal systems. For example, we show termination of all programs in System F. We recall a typical presentation of System F with explicitly named variables. The syntax of System F *types* (A, B) and *terms* (s, t) is given by the following grammar.

$$\begin{aligned} A, B &::= X \mid A \to B \mid \forall X.\ A \\ s, t &::= x \mid s\,t \mid \lambda x.\ s \mid s\,A \mid \Lambda X.\ s \end{aligned}$$

Here, X and x are type and term variables respectively. Informally, we think of variables as symbols drawn from a fixed countably infinite alphabet. New variables can be *bound* in types $(\forall X.\ A)$ and terms $(\lambda x.\ s$, $\Lambda X.\ s)$. Formally, terms have to be identified up-to renaming of bound variables (α-equivalence). A variable which is not bound is called free.

The main operation on terms is capture-avoiding instantiation of free variables by terms and types, written s^x_t and s^X_A. Instantiation s^x_t is the operation of replacing the free variable x in s by the term t while renaming bound variables in s so as to avoid

clashes with free variables in t.

Using instantiation we define the operational semantics of System F inductively by rules such as the following.

$$\frac{}{(\lambda x.\ s)\, t \rhd s^x_t} \qquad \frac{}{(\Lambda X.\ s)\, A \rhd s^X_A} \qquad \frac{s \rhd s'}{s\, t \rhd s'\, t} \qquad \dots$$

These rules define a relation $s \rhd t$ called *reduction*, which formally defines the evaluation of a program in System F. Typing for System F terms ($\Gamma \vdash s : A$) is another example of an inductive relation. Typing determines well-formed programs. The termination theorem (Corollary 5.29) for System F states that all well-typed terms are terminating, i.e., that there is no infinite reduction path starting from s whenever $\Gamma \vdash s : A$ holds.

Proving termination is delicate. For example, termination fails when extending System F with intentional features such as Girard's J operator [51], or the following variant due to Harper and Mitchell [58].

$$\begin{aligned} &J \ :\ \forall XY.\ (X \to X) \to Y \to Y \\ &J\, X\, X\, f \rhd f \\ &J\, X\, Y\, f \rhd \lambda x.\ x && \text{if } X \neq Y \text{ and } X, Y \text{ closed} \end{aligned}$$

Intuitively, $J\, X\, Y\, f\, x$ is a type conditional, which only applies the argument function when the two types X, Y match. If both input types are the same, then the application of J evaluates to $f\, x$, otherwise it returns x unchanged. In isolation, the operator J and its reduction rules are harmless. Yet the following term diverges [58].

$$\begin{aligned} A \ &:=\ \forall X.\ X \to X \\ \omega \ &:=\ \Lambda X.\ J\, A\, X\, (\lambda x.\ x\, A\, x) \\ \omega\, A\, \omega \ &\rhd\ J\, A\, A\, (\lambda x.\ x\, A\, x)\, \omega \ \rhd\ (\lambda x.\ x\, A\, x)\, \omega \rhd \omega\, A\, \omega \end{aligned}$$

This shows that termination is a global property and cannot be deduced by local analysis. Termination can fail due to the interplay of separate and at first glance unrelated features. In this case, intentional type analysis combined with impredicative quantification yields non-termination, even though either feature is harmless in isolation. This behavior is typical for global properties of formal systems.

At the same time, these delicate proofs quickly become tedious.

In order to show termination we have to analyze all terms and reduction paths. This is not a problem for System F, which is a small core language. Realistic programming

languages are much larger. For example, a mathematical presentation of Standard ML takes up 136 pages [83]. Similarly, the mechanized semantics for the big-step evaluation of JavaScript of Bodin et al. [28] has about 900 separate rules. In this case, establishing global properties of evaluation means analyzing all 900 cases. Many of these cases will not be interesting and yet a small mistake in any of them can ruin the whole endeavor.

This combination of subtle yet tedious proofs is fertile ground for errors.

For example, in 1997 Drossopoulou and Eisenbach [43] argued for the type soundness of Java. Yet in the same year, Saraswat [104] found that Java is unsound due to an obscure feature of the language. This mistake has been fixed and further extensions had been carefully scrutinized. Yet in 2016 Amin and Tate [14] showed that the type systems of both Java and Scala remain unsound.

Our long term goal is to eliminate mistakes in proofs about formal systems by mechanizing these proofs in a proof assistant based on constructive type theory. Constructive type theory is a formal system originating with Martin-Löf [80]. The type theory we use in this thesis is an extension of Martin-Löf's type theory with an impredicative universe, functional and propositional extensionality, and inductive types.

What makes type theory appealing for formalizations is that it offers good support for inductive definitions [34, 44]. This is in contrast to classical logics based on arithmetic or set theory, where inductive definitions (along with everything else) are obtained by finding a concrete encoding into numbers or sets. In type theory, inductive types are first-class objects of the theory. This makes type theory ideal for mechanizing proofs about formal systems which are, after all, proofs about inductive definitions.

However, type theory offers little support for reasoning about syntax with binders or coinductive relations. This is the topic we address in this thesis.

We use the Coq proof assistant [121] to mechanize our results. Coq is a mature proof assistant with excellent support for proof automation. In addition, the type theory underlying Coq (the calculus of inductive constructions) includes an impredicative universe of propositions, which is essential for the material from Chapter 6 onwards.

Representing Syntax with Binders. So far we have not discussed the precise definitions of variables, binders and capture avoiding instantiation. In a proof assistant we are forced to be precise.

Informal proofs are usually incomplete when it comes to the treatment of syntax with binders. Typically, variables are written with explicit names and proofs use the

"Barendregt convention" [18] to avoid talking about issues of α-equivalence. The Barendregt convention states that all newly introduced variables ought to be distinct from all variables currently contained in any term.

Formally, terms which differ only by renaming of bound variables should be identified and many inference rules have to carry "freshness" side-conditions which ensure that newly introduced names do not clash with existing names. These side conditions are usually glossed over in proofs "by the Barendregt convention".

This is not ideal for formalizations. In the first place, the Barendregt convention is just a convention, not a sound framework for dealing with variable binding. It is easy to come up with situations where the Barendregt convention leads to wrong results, especially in combination with inductive proofs [127].

A number of methods exist for formalizing syntax with binders, which can be roughly categorized into synthetic approaches (such as higher-order abstract syntax and nominal logic) and explicit approaches (such as de Bruijn and locally nameless terms).

In this thesis we use well-scoped de Bruijn representations [25] throughout. We argue that even on paper a nameless representation leads to short and clear proofs, which are comparable to proofs using a named representation with the Barendregt convention. In particular, we use the σ-calculus of Abadi et al. [1] to automate the proofs of substitution lemmas, which are one of the main sources of overhead compared to informal paper proofs.

Representing Coinductive Relations and Proofs. Coinduction is the order-theoretic dual of induction. While the values of inductive types are essentially trees in which every branch is finite, coinductive types may be infinite or cyclic. Coinductive relations are especially useful for expressing safety properties (absence of errors) without enforcing liveness properties (e.g., termination) at the same time. This makes coinductive relations ideal for formulating partial correctness statements. Coinductive techniques are also relevant for representing program equivalence and for compiler verification [76].

Unlike inductive definitions, there is at best marginal support for coinductive reasoning in type theory. While there are theoretically attractive approaches to coinduction, such as the copatterns of Agda and Beluga [5], none of these approaches are implemented in Coq. Coq provides limited support for coinductive types, but while there is a coinductive analogue to the recursion principle for inductive types, there is no direct analogue to the induction principle. This asymmetry makes reasoning about coinductive relations more difficult than reasoning about their inductive counterparts.

We present a construction of coinductive relations which reduces coinduction to

induction. We formalize the standard construction of coinductive relations in an impredicative universe using order theory. Using the tower construction [112], we obtain both the companion of Pous [99], as well as dual induction and coinduction principles in type theory. The companion allows us to internalize up-to techniques for coinduction, which are useful tools to modularize coinductive proofs [100]. The tower induction principle allows us to perform coinductive proofs with minimal overhead in a proof assistant. This restores the duality between inductive and coinductive relations in type theory.

We demonstrate the power of this approach with a simple development of the theory of strong bisimilarity in CCS with recursive processes and a compiler verification taking both inductive total correctness and coinductive partial correctness into account.

1.1 Structure of the Thesis

The thesis is organized as a series of case studies. Each case study is accompanied by a mechanization in Coq. Our paper presentation is intentionally close to the mechanized development. In particular, we use a nameless term representation in the thesis.

The mechanization is meant to be self-contained. In particular, we do not use the AUTOSUBST library, so as to make the substitution lemmas as explicit as possible. Still, our proofs follow the same structure that we would employ when using AUTOSUBST.

The complete Coq development, as well as a PDF version of the thesis are available online at

https://www.ps.uni-saarland.de/~schaefer/thesis

Definitions and lemmas in the PDF version of the thesis are hyperlinked with the corresponding definitions and lemmas in the formalization. We point out discrepancies between the presentation in the thesis and the formalization in the discussion section of each chapter.

The contents of the thesis are organized as follows.

- **Chapter 2:** We summarize the background type theory, notations, and preliminary definitions.

- **Chapter 3:** We illustrate our approach to working with de Bruijn terms and parallel substitutions on the example of the λ-calculus. We show how the λ-calculus with parallel substitutions forms a model of the σ_{SP}-calculus [1].

Using this model we then show substitutivity of $\beta\eta$-reduction. In particular, we show how substitution lemmas are discharged through rewriting.

- **Chapter 4:** We present a proof of type preservation for CC_ω [78]. We show that the full typing judgment of CC_ω can be extended to a relational (pre-)typing judgment. In this form, we can separate structural properties, such as the context morphism lemma [54], from type soundness and preservation. This allows us to omit well-formedness side conditions when showing structural properties and leads to short proofs.

- **Chapter 5:** We present proofs of weak and strong normalization for System F. Parallel substitutions are already well-adapted to these proofs and we obtain a very concise development. Well-scoped terms of System F are presented using a vector instantiation operation, which generalizes parallel substitutions from single-sorted syntax to multi-sorted syntax.

- **Chapter 6:** We present a general construction of coinductive relations and proof techniques for coinduction. Our construction is based on an inductive tower construction for the greatest fixed-point of a monotone function on a complete lattice. This yields a novel definition of the companion of Pous [99], which itself was introduced as a refinement of the parameterized coinduction of Hur et al. [63].

  In addition, we adapt several results about the tower construction which were previously presented in the context of classical type theory [112] to a constructive setting.

- **Chapter 7:** We develop the metatheory of CCS with recursive processes and strong bisimilarity. In particular, we show that bisimilarity is a congruence, that up-to context reasoning is sound in the presence of recursive processes, and that systems of weakly guarded equations have unique solutions.

- **Chapter 8:** We verify a compiler from a non-deterministic language of guarded commands to a deterministic low-level language of imperative continuations using axiomatic semantics. This development mixes inductive and coinductive reasoning and makes extensive use of the tower construction.

- **Chapter 9:** We explain the design and implementation of AUTOSUBST.

Finally, we give an overview of representations for syntax with binders in **Appendix A**.

1.2 Contributions

- We demonstrate that the combination of (well-scoped) de Bruijn representation with the substitution operations of the σ-calculus provide a practical foundation for formalizing syntax with binders.
- We extend our approach to syntax with binders to type systems through relational typing.
- We present a new proof technique for strong normalization of System F. Our technique scales to handle sums and other positive inductive types, as demonstrated in [47].
- We present a new construction of the companion [99] using an inductive tower construction [112]. The tower construction allows us to construct new proof rules for the companion as well as for coinduction and induction in general, with the (co-)directed (co-)induction principle in Chapter 8.

  The tower construction originates in Zermelo's work on the well-ordering theorem in classical set theory. We show how the central constructions from this setting can be developed in constructive type theory.
- We apply the companion to CCS with recursive processes and show the admissibility of up-to context reasoning using open relative bisimilarity. This technique yields several classic results about strong bisimilarity in CCS as a corollary and allows us to extend these results to the setting of CCS with recursive processes.
- We show how weakest preconditions and the new proof rules for induction and coinduction can be used for compiler verification for a non-deterministic language.
- We automate our approach to formalizing syntax with binders with the AUTOSUBST library for Coq.

1.3 Published Results

[1] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. „Call-by-push-value in Coq: Operational, Equational, and Denotational Theory". In: **Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs**. CPP 2019. Cascais, Portugal: ACM, 2019, pp. 118–131.

[2] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. „Binder Aware Recursion Over Well-Scoped De Bruijn Syntax". In: **Certified Programs and Proofs - 7th International Conference, CPP 2018, Los Angeles, USA, January 8-9, 2018** (Jan. 2018).

[3] Steven Schäfer and Gert Smolka. „Tower Induction and Up-to Techniques for CCS with Fixed Points". In: **International Conference on Relational and Algebraic Methods in Computer Science**. Springer. 2017, pp. 274–289.

[4] Steven Schäfer, Tobias Tebbi, and Gert Smolka. „Autosubst: Reasoning With De Bruijn Terms and Parallel Substitutions". In: **Lecture Notes in Computer Science** (2015), pp. 359–374.

[5] Steven Schäfer, Sigurd Schneider, and Gert Smolka. „Axiomatic Semantics for Compiler Verification". In: **Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs**. CPP 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 188–196.

[6] Steven Schäfer, Gert Smolka, and Tobias Tebbi. „Completeness and Decidability of De Bruijn Substitution Algebra in Coq". In: **Proceedings of the 2015 Conference on Certified Programs and Proofs - CPP '15** (2015).

[7] Gert Smolka, Steven Schäfer, and Christian Doczkal. „Transfinite constructions in classical type theory". In: **International Conference on Interactive Theorem Proving**. Springer. 2015, pp. 391–404.

[8] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. „Autosubst 2: Reasoning with Multi-sorted De Bruijn Terms and Vector Substitutions". In: **Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs**. CPP 2019. Cascais, Portugal: ACM, 2019, pp. 166–180.

The theoretical justification for using the σ-calculus for reasoning about languages with binders was presented in [6]. We first presented the AUTOSUBST library in [4].

The generalization to vector substitutions for multi-sorted systems was presented in [8]. Relational typing first appeared in [2]

Our approach to strong normalization was presented as part of [1].

The type theoretic tower construction first appeared in [7] and its application to coinductive proofs and CCS was presented in [3].

We demonstrated the use of axiomatic semantics for compiler verification in [5].

2 Preliminaries

Every technical result in this thesis is formalized in the Coq proof assistant [121] under the assumption of propositional and functional extensionality. On paper we will occasionally deviate from the actual formalization for notational convenience, so as not to overburden the presentation with details of its type theoretic encoding.

2.1 Type Theory

While the type theory underlying Coq is complex, our proofs use only a small fragment of this theory. Informally, we use a version of Martin-Löf dependent type theory [80] with the following features.

- A predicative and cumulative hierarchy of universes $\mathcal{U}_i$. We will usually omit the universe index and simply write $A : \mathcal{U}$ to say that A is a type in some universe $\mathcal{U}_i$.
- An impredicative universe of propositions $\mathcal{P}$ at the bottom of the universe hierarchy.
- The existence of strictly positive inductive types in all universes.
- The principle of **propositional extensionality** (PE) which states that logically equivalent propositions are equal. That is, for two propositions $P, Q : \mathcal{P}$ such that $P \leftrightarrow Q$ we have $P = Q$.
- The principle of **functional extensionality** (FE) which states that pointwise equal functions are equal. That is, for two functions $f, g : \forall x : A.\, B\, x$ such that $f\, x = g\, x$ for all x we have $f = g$.

These assumptions are by no means minimal. In particular, extensionality assumptions are merely a tool for simplifying proofs.

While functional extensionality is generally accepted as harmless, propositional extensionality is a priori incompatible with Homotopy Type Theory [126]. In our case, we only consider set-level constructions and so our proofs could be translated into Homotopy Type Theory extended with the propositional resizing principle, by using homotopy propositions in place of $\mathcal{P}$.

Another justification for extensionality assumptions in type theory is given by Hofmann [59], who proves that extensionality is a conservative extension of intensional type theory. Formally, Hofmann constructs a Setoid model of type theory, in which a type is interpreted as a type together with a distinguished $\mathcal{P}$-valued partial equivalence relation. This construction can be seen as a proof-term translation from a type theory with extensionality assumptions (FE, PE, quotients, etc.) to a type theory without these features.

Note that PE implies proof irrelevance.

Lemma 2.1 (Proof Irrelevance) For all $P : \mathcal{P}$ and proofs $p, q : P$ we have $p = q$.

Proof Since we have strictly positive inductive types, we can construct a proposition $\top$ with a unique witness $I : \top$. We have $P \leftrightarrow \top$ since P is inhabited. By propositional extensionality it suffices to show the statement for $P = \top$ where we have $p = q = I$. ■

2.2 Notations

We write $\mathbb{N}$ and $\mathbb{B}$ for the type of natural numbers and booleans as usual. We identify booleans with the propositions $\top, \bot$.

We use set notations whenever they simplify the presentation. As is standard in type theory, we identify functions $P : A \to \mathcal{P}$ with sets and use the notations $P\,x$ and $x \in P$ interchangeably. We sometimes define predicates with comprehensions as follows.

$$\begin{aligned} \{\, x \mid P \,\} &:= \lambda x.\ P \\ \{\, f\,x \mid P \,\} &:= \lambda y.\ \exists x.\ f\,x = y \wedge P \end{aligned}$$

Furthermore, we also use the element notation for lists. Lists over a type X are the inductive type with the constructors $[\,]$ for the empty list and $x \cdot s$ for the extension of a list with a new element. When we write $x \in s$ where s is a list we mean that x appears in s.

The one notation we use throughout the thesis which is non-standard is $f \circ g$ for *forward* composition of functions.

$$f \circ g := \lambda x.\ g(f\,x)$$

This is the convention used in the σ-calculus [1].

2.3 Abstract Reduction Systems

We will frequently deal with languages whose semantics are given by a small-step reduction relation. Many definitions and lemmas about such systems are independent of the actual language under consideration and can be formulated for an arbitrary notion of "reduction". For the purpose of this chapter, an "abstract reduction system" is simply a relation $R : A \to A \to \mathcal{P}$.

The study of abstract reduction at this level of generality was introduced by Huét [62]. The definitions in this chapter are carefully chosen to simplify the formalization.

As usual we write $R \subseteq S$ for the order on relations.

$$R \subseteq S \;:=\; \forall xy.\, R\,x\,y \;\to\; S\,x\,y$$

Definition 2.2 The reflexive, transitive closure of a relation $R : A \to A \to \mathcal{P}$ is the relation $R^* : A \to A \to \mathcal{P}$ defined inductively by the following rules.

$$\frac{}{R^*\,s\,s}\;\textsc{$*$-Refl} \qquad\qquad \frac{R^*\,s\,t \qquad R\,t\,u}{R^*\,s\,u}\;\textsc{$*$-Step}$$

The relation R^* is always reflexive by $*$-REFL, but transitivity remains to be shown.

Fact 2.3 R^* is transitive.

Proof Let $R^*\,s\,t$ and $R^*\,t\,u$ be given. We show $R^*\,s\,u$ by induction on the derivation of $R^*\,t\,u$. In the case of $*$-REFL we have $t = u$ and therefore $R^*\,s\,u$ holds by assumption. In the case of $*$-STEP we have $R^*\,s\,u_0$ and $R\,u_0\,u_1$ and hence $R^*\,s\,u_1$ follows by $*$-STEP. ■

It is also clear by the induction principle for R^* that any reflexive and transitive relation containing R necessarily contains R^*. Thus R^* is a closure operator, and in fact has the following "homomorphism" property which is frequently useful for lifting operations into the reflexive, transitive closure.

Fact 2.4 Let $R : X \to X \to \mathcal{P}, S : Y \to Y \to \mathcal{P}$ be relations, $f : X \to Y$ a function such that $R\,x\,y$ implies $S^*\,(f\,x)\,(f\,y)$. Then $R^*\,x\,y$ implies $S^*\,(f\,x)\,(f\,y)$.

Proof By induction on the derivation of $R^*\,x\,y$. The case for $*$-REFL is trivial. In the case of $*$-STEP we have $R^*\,x\,y$ and $R\,y\,z$. We have to show that $S^*\,(f\,x)\,(f\,z)$ holds. This follows from Fact 2.3, since we have $S^*\,(f\,x)\,(f\,y)$ by induction and $S^*\,(f\,y)\,(f\,z)$ by assumption. ■

Fact 2.5 We have $R \subseteq R^*$ for all R and $R^* \subseteq S^*$ whenever $R \subseteq S^*$ for all R, S.

Proof The statement $R \subseteq R^*$ follows from $*$-STEP together with $*$-REFL. Assuming $R \subseteq S^*$ and $R^*\, s\, t$ we have $S^*\, s\, t$ by Fact 2.4 with the identity function for f. ■

In addition to the reflexive, transitive closure of a relation we need the least equivalence relation containing a given relation.

Definition 2.6 The reflexive, transitive, symmetric closure of a relation $R : A \to A \to \mathcal{P}$ is the relation $R^{\equiv} : A \to A \to \mathcal{P}$ defined inductively by the following rules.

$$\frac{}{R^{\equiv}\, s\, s}\ \equiv\text{-REFL} \qquad \frac{R^{\equiv}\, s\, t \qquad R\, t\, u}{R^{\equiv}\, s\, u}\ \equiv\text{-STEP} \qquad \frac{R^{\equiv}\, s\, t \qquad R\, u\, t}{R^{\equiv}\, s\, u}\ \equiv\text{-RSTEP}$$

Fact 2.7 For all relations R we have $R^* \subseteq R^{\equiv}$.

Proof Assume $R^*\, x\, y$, we show $R^{\equiv}\, x\, y$ by induction on the derivation of $R^*\, x\, y$. Both cases follow immediately, since all constructors of R^* are also constructors of $R^{\equiv}$. ■

Just as in the case of the reflexive, transitive closure of a relation, reflexivity of $R^{\equiv}$ is immediate by $\equiv$-REFL, while transitivity and symmetry remain to be shown. The proof of transitivity of $R^{\equiv}$ is completely analogous to the proof of transitivity for R^*, since our definition of $R^{\equiv}$ can be read as the reflexive, transitive closure of the symmetric closure of R. Symmetry then follows, since the transitive closure of a a symmetric relation is symmetric.

Fact 2.8 $R^{\equiv}$ is transitive.

Proof By induction analogous to the proof of Fact 2.3. ■

Fact 2.9 $R^{\equiv}$ is symmetric.

Proof Assuming $R^{\equiv}\, s\, t$ we show $R^{\equiv}\, t\, s$ by induction on the derivation of $R^{\equiv}\, s\, t$. In the case of $\equiv$-REFL we have $s = t$ and $R^{\equiv}\, t\, s$ follows by $\equiv$-REFL. In the case of $\equiv$-STEP we have $R^{\equiv}\, t\, s$ and $R\, t\, u$ and have to show that $R^{\equiv}\, u\, s$ holds. By $\equiv$-RSTEP and $\equiv$-REFL we have $R^{\equiv}\, u\, t$ and $R^{\equiv}\, u\, s$ follows by transitivity. The case of $\equiv$-RSTEP is analogous. ■

Finally, $R^{\equiv}$ is a closure operator and in particular, computes the reflexive, transitive, symmetric closure as desired. As for R^* we note the stronger homomorphism property.

Fact 2.10 Let $R : X \to X \to \mathcal{P}, S : Y \to Y \to \mathcal{P}$ be relations, $f : X \to Y$ a function such that $R\,x\,y$ implies $S^{\equiv}\,(f\,x)\,(f\,y)$. Then $R^{\equiv}\,x\,y$ implies $S^{\equiv}\,(f\,x)\,(f\,y)$.

Proof Analogous to the proof of Fact 2.4 using Fact 2.8, Fact 2.9. ∎

Fact 2.11 We have $R \subseteq R^{\equiv}$ for all R and $R^{\equiv} \subseteq S^{\equiv}$ whenever $R \subseteq S^{\equiv}$ for all R, S.

Proof Analogous to the proof of Fact 2.5. ∎

Definition 2.12 (Properties of Abstract Reduction Systems)

- Two relations R, S **commute** if for all x, y, z such that $R\,x\,y$ and $S\,x\,z$ there exists some w such that $R\,z\,w$ and $S\,y\,w$.
- A relation has the **diamond property** if it commutes with itself.
- A relation R is **confluent** if R^* has the diamond property.
- A relation R has the **Church-Rosser property** if $R^{\equiv}\,x\,z$ implies that there is some y such that $R^*\,x\,y$ and $R^*\,z\,y$.

Lemma 2.13 If R, S commute then so do S^* and R.

Proof Assume that $S^*\,x\,y$ and $R\,x\,z$ hold. We have to find some w such that $R\,y\,w$ and $S^*\,z\,w$ hold. The proof proceeds by induction on the derivation of $S^*\,x\,y$. In the case of REFL we have $x = y$ and we choose $w = z$. In the case of $*$-STEP we have $S^*\,x\,y'$ and $S\,y'\,y$. By induction there is some w' such that $R\,y'\,w'$ and $S^*\,z\,w'$ hold. But since R, S commute there is also some w such that $R\,y\,w$ and $S\,w'\,w$ hold and thus also $S^*\,z\,w$. ∎

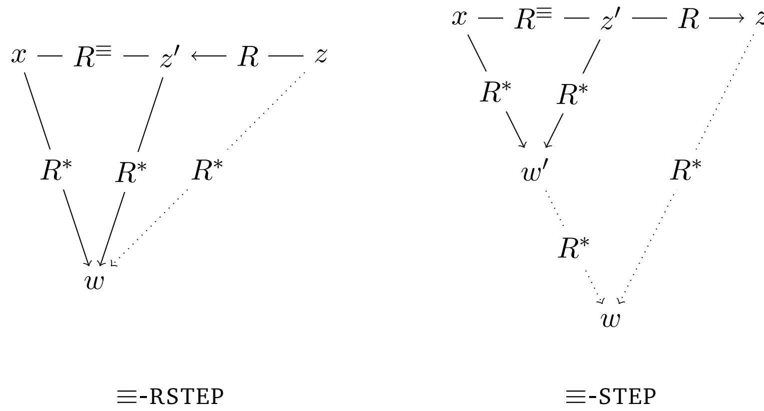
Corollary 2.14 If R has the diamond property then R is confluent.

Lemma 2.15 A relation R has the Church-Rosser property iff it is confluent.

Proof Assume that R has the Church-Rosser property and that we have $R^*\,x\,y$ and $R^*\,x\,z$. By Fact 2.7 we have $R^{\equiv}\,x\,y$ and $R^{\equiv}\,x\,z$ and since $R^{\equiv}$ is symmetric and transitive we have $R^{\equiv}\,y\,z$. By the Church-Rosser property it follows that there is some w such that $R^*\,y\,w$ and $R^*\,z\,w$, which implies that R is confluent.

In the other direction, assume that R is confluent and that we have $R^{\equiv}\,x\,z$. We show that there is some w such that $R^*\,x\,w$ and $R^*\,z\,w$ by induction on the derivation of $R^{\equiv}\,x\,z$.

- In the case of $\equiv$-REFL we have $x = y$ and choose $w = x$.



- In the case of $\equiv$-RSTEP we have $R^{\equiv}\,x\,z'$ and $R\,z\,z'$ and thus in particular $R^*\,z\,z'$. By induction there is some w such that $R^*\,x\,w$ and $R^*\,z'\,w$ and thus by transitivity $R^*\,z\,w$.
- In the case of $\equiv$-STEP we have $R^{\equiv}\,x\,z'$ and $R\,z'\,z$ and thus also $R^*\,z'\,z$. By induction there is some w' such that $R^*\,x\,w'$ and $R^*\,z'\,w'$. At this point we use the fact that R has the Church-Rosser property, which implies that there is some w such that $R^*\,w'\,w$ and $R^*\,z\,w$ and hence by transitivity also $R^*\,x\,w$. ■

2.4 Discussion

In the presence of functional and propositional extensionality, the existence of all strictly positive inductive types follows from the existence of a number of simpler inductive types [2, 11]. For example, it would be sufficient to assume the existence of a propositional equality type in $\mathcal{P}$, together with booleans, W-types, and Σ types in $\mathcal{U}_i$ for all i and Π types in all universes. In Chapter 6 we will explicitly construct coinductive predicates using the impredicativity of $\mathcal{P}$. The same construction could be used for inductive predicates, further reducing the type theoretic preliminaries we rely on.

In fact, the use of extensionality in these constructions is only a matter of convenience. Using Hofmann's model of extensional type theory the constructions would simply yield setoids rather than types with the right equality.

Propositional extensionality only implies proof irrelevance when we already have a proposition with a unique witness [23]. In the proof of Lemma 2.1 we use the proposition $\top$ with the unique witness I for this purpose.

Quotients in Type Theory. The type theory we consider lacks quotient types, which would be required to develop named terms modulo α-equivalence. None of the

results in the thesis require quotient types, apart from those quotients which we build implicitly using FE and PE.

For completeness, the Coq development accompanying the thesis includes a construction of FE and quotient types from the existence of a "propositional truncation" in the sense of Homotopy Type Theory [126]. The type theory we use in our formalization is effectively a subset of observational type theory [12].

3 De Bruijn Representation of Syntax with Binders

In this chapter we focus on the de Bruijn representation on the example of the λ-calculus [32]. In order to reason effectively about de Bruijn terms we introduce parallel substitution and instantiation operations on terms, and investigate their equational theory as in [106]. We show that terms and parallel substitutions yield a model of the σ_{SP}-calculus [38], which is a slight extension of the σ-calculus of explicit substitutitions [1]. This allows us to automate all “substitution lemmas” which appear in proofs about the λ-calculus and related systems.

We illustrate this approach in a proof that $\beta\eta$-reduction respects instantiation of variables.

The content of this chapter is based on [106] and [108].

3.1 De Bruijn Representation of λ-Terms

The λ-calculus is the archetype of a language with local definitions. A **term** of the λ-calculus is either an application, a binder, or a variable. A variable is a reference — either to a binder within the same term or to an outside context. In order to make this context explicit we represent terms as a family of types $\mathbb{E}_k$ for $k \in \mathbb{N}$. An element of the type $\mathbb{E}_k$ is a term in a context of size k. When we want to be explicit about the size of the context we write s^k, t^k for elements of $\mathbb{E}_k$. In most cases the size of the context is clear and we simply write s, t for terms.

A variable in a context of size k is represented as an **index** $x, y, z \in \mathsf{I}_k$, where I_k is the finite type with exactly k elements. For notational convenience, we identify I_k with the corresponding initial segment of the natural numbers. Under this identification, we then use the **de Bruijn index convention** where the index n refers to the n-th enclosing binder, counting from 0.

De Bruijn terms for the λ-calculus are given by the following grammar.

$$\mathbb{E}_k \ni s^k, t^k ::= x^k \mid s^k\, t^k \mid \lambda\, s^{k+1} \qquad \text{where } x^k \in \mathsf{I}_k$$

We avoid confusion between object-level binders ($\lambda\, s$) and meta-level binders ($\lambda x.x$) by omitting the trailing dot in object-level binders. The position of bound variables in object-level terms is explicit in the grammar. We give some examples of informal

named terms and their corresponding de Bruijn representations.

$$\lambda x.\, x \qquad \lambda\, 0 \qquad : \mathbb{E}_0$$

$$\lambda x\, y.\, x \qquad \lambda\, \lambda\, 1 \qquad : \mathbb{E}_0$$

$$\lambda y\, x.\, y \qquad \lambda\, \lambda\, 1 \qquad : \mathbb{E}_0$$

$$\lambda x.\, x\, a\, (\lambda y\, z.\, z\, y) \qquad \circ \vdash \lambda\, 0\, 1\, (\lambda\, \lambda\, 0\, 1) \qquad : \mathbb{E}_1$$

There are two sources of ambiguity in the named representation. The first is the usual issue of α-equivalence. Since variables are references to a binder, the names of bound variables are immaterial and we would have to formally quotient named terms modulo renaming of bound variables. This ambiguity is simply not present in the de Bruijn representation, as seen in the second and third example above.

Under the de Bruijn index convention the meaning of index i depends on the context in which it occurs. For example, in the fourth term above, the index 0 occurs twice, but refers to different binders in each case.

With the well-scoped representation of de Bruijn terms, there is a second source of ambiguity, where every named open term corresponds to multiple de Bruijn terms. A free variable in a named term is a dangling reference to an (implicit) outside context. In the de Bruijn representation we make the context explicit which eliminates this ambiguity.

We can manipulate terms by changing the interpretation of the context. The most straightforward way of doing this is by renaming variables, that is, replacing one context by another. More generally, we can instantiate variables by terms living in a different context using a substitution.

A **substitution** is a function mapping indices to terms. A **renaming** is a substitution that replaces indices by indices. For convenience, we identify functions between finite types and renamings. Note that we do not assume that renamings are bijective. The letters σ, τ, θ denote substitutions, while ξ, ζ stand for renamings.

Formally, a substitution maps from a specific finite type (e.g., I_m) into a specific term type (e.g., $\mathbb{E}_n$). However, the domains of substitutions will always be clear from the context and hence we do not make this explicit in the notation for substitutions.

A substitution can be seen as a list $(s_0, s_1, s_2, \ldots, s_m)$ of terms. This view motivates the definition of a **cons** operation $s \cdot \sigma$.

$$_\cdot_ \;:\; \mathbb{E}_n \to (\mathsf{I}_m \to \mathbb{E}_n) \to (\mathsf{I}_{m+1} \to \mathbb{E}_n)$$
$$s \cdot \sigma := \lambda x.\begin{cases} s & \text{if } x = 0 \\ \sigma(x-1) & \text{otherwise} \end{cases}$$

Intuitively, cons allows us to extend a list $\sigma = (t_0, \ldots, t_m)$ by a new head element $s \cdot \sigma = (s, t_0, \ldots, t_m)$.

We also introduce notations for the **identity** and **shift** renamings.

$$\mathsf{id} \;:\; \mathsf{I}_n \to \mathbb{E}_n \qquad\qquad \uparrow \;:\; \mathsf{I}_n \to \mathbb{E}_{n+1}$$
$$\mathsf{id} \;:=\; \lambda x.\, x \qquad\qquad \uparrow \;:=\; \lambda x.\, x + 1$$

Next, we define the **instantiation** of a term under a renaming $s\langle\xi\rangle$.

$$_\langle_\rangle \;:\; \mathbb{E}_m \to (\mathsf{I}_m \to \mathsf{I}_n) \to \mathbb{E}_n$$
$$x\langle\xi\rangle := \xi(x)$$
$$(s\,t)\langle\xi\rangle := s\langle\xi\rangle\, t\langle\xi\rangle$$
$$(\lambda\, s)\langle\xi\rangle := \lambda\, s\langle 0 \cdot \xi \circ \uparrow\rangle$$

Where the notation $\xi \circ \uparrow$ in the third equation refers to *forward* composition of functions (ξ then $\uparrow$).

In the third equation, we change the renaming ξ to reflect the extended context in the body of a binder. While $\lambda\, s$ is a term in a context of size m, its body is a term in a context of size $m + 1$. It follows that we also need to extend the renaming $\xi : \mathsf{I}_m \to \mathsf{I}_n$ to a renaming $\Uparrow\xi = 0 \cdot \xi \circ \uparrow \;:\; \mathsf{I}_{m+1} \to \mathsf{I}_{n+1}$. As we want to implement capture-avoiding renaming, we have to preserve the bound index 0 and shift all other indices to skip over the additional binder. Pleasantly, this is the only natural definition of $\Uparrow\xi$ in the well-scoped setting for this case, so the types guide us towards a correct implementation.

We write $\Uparrow\sigma$ (pronounced **up**) for this operation and extend it to substitutions as

follows.

$$\Uparrow_ \;:\; (\mathsf{I}_m \to \mathbb{E}_n) \to (\mathsf{I}_{m+1} \to \mathbb{E}_{n+1})$$
$$\Uparrow\sigma := \lambda x. \begin{cases} 0 & \text{if } x = 0 \\ \sigma(x-1)\langle\uparrow\rangle & \text{otherwise} \end{cases}$$

Using this definition of up, we extend the **instantiation** and **composition** operations $s[\sigma]$ (read "s under σ") and $\sigma \circ \tau$ to substitutions. Instantiation implements capture-avoiding substitution and composition of substitutions, respectively.

$$\begin{aligned} _[_] &: \mathbb{E}_m \to (\mathsf{I}_m \to \mathbb{E}_n) \to \mathbb{E}_n \\ x[\sigma] &= \sigma(x) \\ (s\,t)[\sigma] &= s[\sigma]\,t[\sigma] \\ (\lambda\,s)[\sigma] &= \lambda\,s[\Uparrow\sigma] \end{aligned}$$

$$\begin{aligned} _\circ_ &: (\mathsf{I}_m \to \mathbb{E}_n) \to (\mathsf{I}_n \to \mathbb{E}_k) \to (\mathsf{I}_m \to \mathbb{E}_k) \\ (\sigma \circ \tau)(x) &= \sigma(x)[\tau] \end{aligned}$$

There is a lot of redundancy between the definitions of instantiation for renamings and substitutions. This is not surprising, since instantiation for renamings is a special case of instantiation for substitutions.

Lemma 3.1 $s\langle\xi\rangle = s[\xi]$

Proof By induction on s. The cases for variables and application are immediate from the definition. For $s = \lambda\,t$ we use the definition of $\Uparrow\xi$ along with the inductive hypothesis to show

$$(\lambda\,s)\langle\xi\rangle = \lambda\,s\langle 0 \cdot \xi \circ \uparrow\rangle = \lambda\,s\langle\Uparrow\xi\rangle = \lambda\,s[\Uparrow\xi] = (\lambda\,s)[\xi].$$ ■

In particular, this shows that the up operation can be defined in terms of cons and substitution composition.

Corollary 3.2 $\Uparrow\sigma \;=\; 0 \cdot (\sigma \circ \uparrow)$

3.2 Equational Theory of Substitutions

De Bruijn terms and parallel substitutions, together with the substitution operations (instantiation, composition, cons, shift, and id) form a model of the σ-calculus by Abadi et al. [1]. The σ-calculus is a calculus of explicit substitutions. Explicit

$$
\begin{aligned}
(s\,t)[\sigma] &= s[\sigma]\,t[\sigma] & \mathsf{id} \circ \sigma &= \sigma \\
(\lambda\, s)[\sigma] &= \lambda\, s[0 \cdot \sigma \circ \uparrow] & \sigma \circ \mathsf{id} &= \sigma \\
0[s \cdot \sigma] &= s & (\sigma \circ \tau) \circ \theta &= \sigma \circ (\tau \circ \theta) \\
\uparrow \circ\, (s \cdot \sigma) &= \sigma & (s \cdot \sigma) \circ \tau &= s[\tau] \cdot \sigma \circ \tau \\
s[\mathsf{id}] &= s & s[\sigma][\tau] &= s[\sigma \circ \tau] \\
0[\sigma] \cdot (\uparrow \circ\, \sigma) &= \sigma & 0 \cdot \uparrow &= \mathsf{id}
\end{aligned}
$$

Fig. 3.1.: The convergent rewriting system of the σ_{SP}-calculus

substitutions were introduced to analyze reduction and its implementation in a more fine-grained way. For us, it is interesting for two reasons.

First, the σ-calculus can express all substitutions necessary to describe reductions in the λ-calculus. For instance, β-reduction can be expressed using cons and id. In de Bruijn representation, a term $(\lambda\, s)\, t$ reduces to the term s where the index 0 is replaced by t and every other index is decremented, since we move to a smaller context. Using the substitution operations of the σ-calculus, this is expressed as $(\lambda\, s)\, t \rhd s[t \cdot \mathsf{id}]$. We can similarly express η-reduction as $(\lambda\, s[\uparrow]\, 0) \rhd s$. The bound index 0 cannot appear in the image of s under the shift substitution, which replaces an explicit side condition on the rule which would be necessary in a named presentation of the calculus [18].

Second, the σ-calculus yields a useful decision procedure for equational substitution lemmas. The decision procedure is based on the rewriting system shown in Figure 3.1. As a rewriting system, the rules of Figure 3.1 are confluent [38] and strongly normalizing [36]. As an equational theory, they are complete for our definition of the substitution operations [108]. Thus, we obtain a rewriting-based decision procedure for equations containing the operations defined in this section, the term constructors of the λ-calculus, and universally quantified meta-variables for terms and substitutions.

We will not repeat the proof of completeness here, which we have previously presented in [108], since no further developments depend on this result. The fact that the substitution operations as defined form a model of the σ-calculus is however crucial for the following developments.

The equations of the σ-calculus fall into two categories. Definitional rules, such as $(s\,t)[\sigma] = s[\sigma]\,t[\sigma]$, specify how instantiation interacts with the terms of a particular language. General rules, such as $(s \cdot \sigma) \circ \tau = s[\tau] \cdot \sigma \circ \tau$ are basic properties of our definitions and not bound to any particular term language.

Fact 3.3 For all s, σ we have

(1) $(s\,t)[\sigma] = s[\sigma]\,t[\sigma]$

(2) $(\lambda\, s)[\sigma] = \lambda\, s[0 \cdot \sigma \circ \uparrow]$

(3) $0[s \cdot \sigma] = s$

(4) $\uparrow \circ\, (s \cdot \sigma) = \sigma$

(5) $\mathsf{id} \circ \sigma = \sigma$

(6) $0[\sigma] \cdot (\uparrow \circ\, \sigma) = \sigma$

(7) $0 \cdot \uparrow = \mathsf{id}$

(8) $(s \cdot \sigma) \circ \tau = s[\tau] \cdot \sigma \circ \tau$

Proof By definition with functional extensionality and Corollary 3.2. ∎

Next, we consider the compatibility of instantiation and composition with the identity substitution.

Lemma 3.4 For all terms s and substitutions σ we have

(1) $s\langle \mathsf{id} \rangle = s$

(2) $s[\mathsf{id}] = s$

(3) $\sigma \circ \mathsf{id} = \sigma$

Proof (1) By induction on s. We use functional extensionality to show that $\Uparrow\!\mathsf{id} = \mathsf{id}$.

(2) By (1), Fact 3.3, and Lemma 3.1.

(3) By (2) with functional extensionality. ∎

Finally, we consider the compatibility of instantiation and composition.

Lemma 3.5 For all $s, \xi, \zeta, \sigma, \tau, \theta$ we have

(1) $s\langle \xi \rangle[\sigma] = s[\xi \circ \sigma]$

(2) $s\langle \xi \rangle\langle \zeta \rangle = s\langle \xi \circ \zeta \rangle$

(3) $s[\sigma]\langle \xi \rangle = s[\sigma \circ \xi]$

(4) $\Uparrow\!\sigma \circ \Uparrow\!\tau = \Uparrow\!(\sigma \circ \tau)$

(5) $s[\sigma][\tau] = s[\sigma \circ \tau]$

(6) $(\sigma \circ \tau) \circ \theta = \sigma \circ (\tau \circ \theta)$

Proof (1) By induction on s. For $s = \lambda\, t$ we have

$$
\begin{aligned}
(\lambda\, t)\langle \xi \rangle[\sigma] &= \lambda\, t\langle 0 \cdot \xi \circ \uparrow \rangle[\Uparrow\!\sigma] \\
&= \lambda\, t[(0 \cdot \xi \circ \uparrow) \circ \Uparrow\!\sigma] \\
&= \lambda\, t[0 \cdot (\xi \circ \sigma \circ \uparrow)] \\
&= \lambda\, t[\Uparrow\!(\xi \circ \sigma)] \\
&= (\lambda\, t)[\xi \circ \sigma]
\end{aligned}
$$

(2) By (1) and Lemma 3.1.

(3) By induction on s similar to (1). For $s = \lambda\, t$ we have to show $\Uparrow\sigma \circ (0 \cdot \xi \circ \uparrow) = \Uparrow(\sigma \circ \xi)$ which follows from (1) and (2) with functional extensionality.

(4) By (1) and (3) with functional extensionality.

(5) By induction on s similar to (1) using (4) in the case where $s = \lambda\, t$.

(6) By (5) using functional extensionality. ∎

This concludes the proof of all equations in Figure 3.1.

3.3 Case Study: Substitutivity of $\beta\eta$-Reduction

We now illustrate how the equations in Figure 3.1 help with solving substitution lemmas. We show that $\beta\eta$-reduction on λ-terms is *substitutive*, i.e., compatible with instantiation. Besides being an obviously desirable property of reduction, substitutivity is required to show confluence of reduction and also plays a role in some proofs of strong normalization.

Definition 3.6 The $\beta\eta$-reduction relation $s \rhd t$ is inductively defined by the following rules.

β-REDUCTION
$$\frac{}{(\lambda\, s)\, t \rhd s[t \cdot \mathsf{id}]}$$

η-REDUCTION
$$\frac{}{\lambda\, s[\uparrow]\, 0 \rhd s}$$

APPL
$$\frac{s \rhd s'}{s\, t \rhd s'\, t}$$

APPR
$$\frac{t \rhd t'}{s\, t \rhd s\, t'}$$

LAM
$$\frac{s \rhd s'}{\lambda\, s \rhd \lambda\, s'}$$

Lemma 3.7 (Substitutivity) $s \rhd t$ implies $s[\sigma] \rhd t[\sigma]$.

Proof By induction on the derivation of $s \rhd t$. The congruence rules APPL, APPR, and LAM follow immediately from the inductive hypothesis.

For β-REDUCTION we have

$$\begin{aligned}
((\lambda\, s)\, t)[\sigma] &= (\lambda\, s[0 \cdot \sigma \circ {\uparrow}])\, t[\sigma] && \text{Fact 3.3 (1,2)}\\
&\rhd s[0 \cdot \sigma \circ {\uparrow}][t[\sigma] \cdot \mathsf{id}] && \beta\text{-REDUCTION}\\
&= s[(0 \cdot \sigma \circ {\uparrow}) \circ (t[\sigma] \cdot \mathsf{id})] && \text{Lemma 3.5 (5)}\\
&= s[0[t[\sigma] \cdot \mathsf{id}] \cdot (\sigma \circ {\uparrow}) \circ (t[\sigma] \cdot \mathsf{id})] && \text{Fact 3.3 (8)}\\
&= s[t[\sigma] \cdot \sigma \circ ({\uparrow} \circ (t[\sigma] \cdot \mathsf{id}))] && \text{Fact 3.3 (3), Lemma 3.5 (6)}\\
&= s[t[\sigma] \cdot \sigma \circ \mathsf{id}] && \text{Fact 3.3 (4)}\\
&= s[t[\sigma] \cdot \sigma] && \text{Lemma 3.4 (3)}\\
&= s[t[\sigma] \cdot \mathsf{id} \circ \sigma] && \text{Fact 3.3 (5)}\\
&= s[(t \cdot \mathsf{id}) \circ \sigma] && \text{Fact 3.3 (8)}\\
&= s[t \cdot \mathsf{id}][\sigma] && \text{Lemma 3.5 (5)}
\end{aligned}$$

For η-REDUCTION we have

$$\begin{aligned}
(\lambda\, s[{\uparrow}]\, 0)[\sigma] &= \lambda\, (s[{\uparrow}][0 \cdot \sigma \circ {\uparrow}]\; 0[0 \cdot \sigma \circ {\uparrow}]) && \text{Fact 3.3 (1,2)}\\
&= \lambda\, (s[{\uparrow} \circ (0 \cdot \sigma \circ {\uparrow})]\; 0) && \text{Lemma 3.5 (5), Fact 3.3 (3)}\\
&= \lambda\, (s[\sigma \circ {\uparrow}]\; 0) && \text{Fact 3.3 (4)}\\
&= \lambda\, (s[\sigma][{\uparrow}]\; 0) && \text{Lemma 3.5 (5)}\\
&\rhd s[\sigma] && \eta\text{-REDUCTION}
\end{aligned}$$

■

The proof of Lemma 3.7 is, of course, unnecessarily verbose for illustration purposes. Since the rules in Figure 3.1 are confluent we can always replace long chains of equational reasoning by reducing both sides of an equation to normal form. For example, in the case of β-REDUCTION in the proof of Lemma 3.7, we have $s[0 \cdot \sigma \circ {\uparrow}][t[\sigma] \cdot \mathsf{id}] = s[t \cdot \mathsf{id}][\sigma]$ since both sides reduce to the same normal form, namely $s[t[\sigma] \cdot \sigma]$.

3.4 Discussion

Throughout this thesis we consider formal systems with binders quantifying over the terms of the system itself. This is in contrast to systems such as first-order logic, which contain binders quantifying over a different type — in that case the complications we deal with can be avoided by only considering closed terms [50].

When we reason about syntax with binders in type theory we need to choose a concrete representation of binders. We explain our choice of a well-scoped de Bruijn

representation in Appendix A.

The use of a nameless representation of λ-terms along with the use of parallel substitutions goes back to de Bruijn [30]. Nevertheless, the use of single-index substitutions is widespread in the programming language community. Compared to parallel instantiation, single-index instantiation is equally complicated to define, significantly less expressive, and the definition involves ad-hoc recursive “shift” functions.

We believe that single-index substitutions are responsible for much of the dissatisfaction with de Bruijn terms. In contrast, the careful choice of substitution operations by Abadi et al. [1] makes this approach to formalizing syntax elegant.

Using unscoped de Bruijn terms as in [30] is fairly error-prone. The use of well-scoped term representations goes back to Bird and Paterson [25], and represents a particularly sweet spot in the design space. On the one hand, well-scoped term representations avoid many of the pitfalls of programming with unscoped de Bruijn terms [6]. On the other hand, well-scoped terms are not bound to any particular type system unlike intrinsically typed terms. Well-scoped terms also easily scale to dependently typed languages, which is otherwise problematic with intrinsically typed terms.

There is also some theoretical support behind using well-scoped de Bruijn terms. Altenkirch and Reus [10] first introduced monadic term representations, which can be seen as another refinement of de Bruijn's term representation. Monads are used extensively in the categorical approach to abstract algebra [64] and so their appearance in term representations for higher-order languages makes sense. However, in a monadic term representation variables are represented as elements of an arbitrary type, which might not even have decidable equality. In order to get a useful model for language in which variables must have identity — such as System $F_{<:}$ which has been the subject of the POPLmark challenge [17] — we need more restrictions on the types interpreting variables. Following this thread to its logical conclusion leads to the theory of relative monads [13]. Indeed, Lemma 3.4, Lemma 3.5 exactly state that well-scoped de Bruijn terms with instantiation form a monad relative to $\mathsf{I}_$, regarded as a functor from the skeleton of the category of finite sets into the category Set. An equivalent construction (in terms of presheafs over a category of renamings) has also appeared in Fiore et al.'s categorical approach to abstract syntax with variable binding [46].

There is an apparent ambiguity remaining in our definition of de Bruijn terms. An open term such as $\lambda\, 1 : \mathbb{E}_1$ exists in all contexts with at least one free variable. There is some indication that removing this ambiguity actually leads to a superior term

representation. In particular, McBride [81] presents the first structurally recursive implementation for instantiation of parallel hereditary substitutions. The definition makes essential use of a precise term representation which keeps track of exact contexts. For the topics we consider in this thesis, however, well-scoped de Bruijn terms are perfectly adequate.

There are several differences between the presentation in this chapter and the actual Coq formalization of de Bruijn terms. While we could define finite types as initial segments of the natural numbers[1], doing so would not lead to very nice reduction behavior. Instead, the easiest definition to use seems to be to define the finite type I_k by recursion on k as follows.

$$\begin{aligned} \mathsf{I}_0 &:= \mathsf{void} \\ \mathsf{I}_{n+1} &:= \mathsf{option}\,\mathsf{I}_n \end{aligned}$$

This leads to equally simple definition for $\uparrow$ as Some and 0 as None and $_ \cdot _$ as the recursor for option. Most of the rules in Fact 3.3 then hold definitionally which at the very least leads to compact proof terms.

Additionally, while on paper we do not make a distinction between the types of renamings and substitutions, this distinction has to be explicit in the Coq formalization. This is essentially an artifact of Coq's termination checker and for the most part we consider it an implementation detail that is best ignored. With better support for well-founded recursion, e.g., as in F^* we could also work with a direct definition of instantiation and avoid introducing renamings at all [117].

[1] This is how finite types are implemented in the mathematical components library for Coq [55].

Subject Reduction in CC_ω 4

Subject reduction, or type preservation states that typing is preserved under reduction. This is a classic metatheoretic statement about typed languages. We use subject reduction in CC_ω as a case study for formalized metatheory using de Bruijn representation.

The calculus CC_ω is the extended calculus of constructions (ECC) [78] without dependent sum types. Explicitly, CC_ω is a Martin-Löf dependent type theory with a hierarchy of predicative universes and a single impredicative universe of propositions. Types are identified under conversion and subtyping, both of which complicate the proof of subject reduction. Nevertheless, using the techniques of Chapter 3 all proofs go through with minimal overhead compared to a traditional paper presentation of this proof [78].

Our formalized setting suggests several improvements over the usual formulation of the typing judgment in ECC. We present a two level approach to typing in CC_ω, where we first present a relational "pretyping" judgment whose context is an arbitrary — not necessarily well-formed — hypothetical typing judgment in the sense of higher-order abstract syntax [89]. We then identify the well-formed types and contexts in a second step and derive the full typing judgment of CC_ω. This approach avoids most of the complicated well-formedness side conditions on ECC typing and leads to simpler and more uniform proofs.

We also present a more direct formulation of the subtyping relation, compared to Luo [78]. Our subtyping relation is not obviously transitive, but much closer to an actual implementation of subtyping in a proof assistant.

4.1 Syntax and Semantics of CC_ω

We define a family of types $\mathbb{E}_k$ of terms in a context of size k inductively according to the following grammar.

$$\mathbb{E}_k \ni s^k, t^k, A^k, B^k ::= x^k \mid s^k\, t^k \mid \lambda\, s^{k+1} \mid \Pi A^k\; B^{k+1} \mid U_\alpha \qquad \text{where } x^k \in \mathsf{I}_k$$
$$\alpha \in \{z \in \mathbb{Z} \mid z \geq -1\}$$

Here U_α refers to a universe, which is either the impredicative universe of propositions U_{-1} or one of infinitely many predicative universes $U_0, U_1, \ldots$ In the remainder we usually write A, B for types and s, t for terms, but there is no syntactic difference

between the two in CC_ω.

We define instantiation on terms as in Chapter 3. Instantiation satisfies the following definitional equations, along with the general equations of the σ-calculus.

$$\begin{aligned} x[\sigma] &= \sigma\, x \\ (s\, t)[\sigma] &= s[\sigma]\, t[\sigma] \\ (\lambda\, s)[\sigma] &= \lambda\, s[\Uparrow\sigma] \\ (\Pi s\ t)[\sigma] &= \Pi s[\sigma]\ t[\Uparrow\sigma] \\ (U_\alpha)[\sigma] &= U_\alpha \end{aligned} \qquad \begin{aligned} s[\mathsf{id}] &= s \\ s[\sigma][\tau] &= s[\sigma \circ \tau] \end{aligned}$$

We define the reduction relation of CC_ω inductively by the following rules.

$$\beta\text{-REDUCTION}\ \frac{}{(\lambda\, s)\, t \rhd s[t \cdot \mathsf{id}]} \qquad \text{APP}_L\ \frac{s \rhd s'}{s\, t \rhd s'\, t} \qquad \text{APP}_R\ \frac{t \rhd t'}{s\, t \rhd s\, t'}$$

$$\text{LAM}\ \frac{s \rhd s'}{\lambda\, s \rhd \lambda\, s'} \qquad \text{PI}_L\ \frac{s \rhd s'}{\Pi s\ t \rhd \Pi s'\ t} \qquad \text{PI}_R\ \frac{t \rhd t'}{\Pi s\ t \rhd \Pi s\ t'}$$

We write $s \rhd^* t$ for the reflexive, transitive closure of reduction (Definition 2.2) and $s \equiv t$ for **conversion**, the reflexive, transitive, symmetric closure of reduction (Definition 2.6).

The universes and types of CC_ω are connected with a subtyping relation. Subtyping includes both conversion, identifying types which are definitionally equal, as well as the cumulativity of universes, ensuring that types in U_i are included in U_j for $i < j$. We choose to separate these two concerns in the definition below, since this simplifies subsequent inversion lemmas. We first introduce a type inclusion judgment $A <:_1 B$, which implements cumulativity and then combine this judgment with conversion to obtain the full subtyping judgment $A <: B$.

Definition 4.1 Let A, B be two terms of CC_ω. We define the type inclusion relation $A <:_1 B$ inductively by the following rules.

$$\frac{}{A <:_1 A} \qquad \frac{m \leq n}{U_m <:_1 U_n} \qquad \frac{B_1 <:_1 B_2}{\Pi A\ B_1 <:_1 \Pi A\ B_2}$$

The full subtyping relation is defined as the relational composition of type inclusion

with conversion.

$$A <: B \quad := \quad \exists A' \, B'. \; A \equiv A' <:_1 B' \equiv B$$

Using subtyping we define the typing judgment of CC_ω. We first present a pre-typing judgment as a function between term relations. This presentation is inspired by typing judgments in higher-order abstract syntax and differs from the more traditional presentation where typing judgments are defined in terms of a finite context mapping variables to their types.

Definition 4.2 We define the (pre-)typing judgment of CC_ω

$$_ \vdash _ : _ \quad : \quad (\mathbb{E}_n \to \mathbb{E}_n \to \mathcal{P}) \to \mathbb{E}_n \to \mathbb{E}_n \to \mathcal{P}$$

inductively for all n by the following rules.

$$\text{VAR}\ \frac{\mathcal{G}\, s\, A}{\mathcal{G} \vdash s : A} \qquad \text{SORT}\ \frac{}{\mathcal{G} \vdash U_n : U_{n+1}} \qquad \text{APP}\ \frac{\mathcal{G} \vdash s : \Pi A\, B \qquad \mathcal{G} \vdash t : A}{\mathcal{G} \vdash s\, t : B[t \cdot \mathsf{id}]}$$

$$\text{LAM}\ \frac{\mathcal{G} \vdash A : U_n \qquad \mathcal{G}, A \vdash s : B}{\mathcal{G} \vdash \lambda\, s : \Pi A\; B} \qquad \text{PI}\ \frac{\mathcal{G} \vdash A : U_m \qquad \mathcal{G}, A \vdash B : U_n}{\mathcal{G} \vdash \Pi A\; B : U_{m \oplus n}}$$

$$\text{SUB}\ \frac{\mathcal{G} \vdash s : A \qquad A <: B \qquad \mathcal{G} \vdash B : U_n}{\mathcal{G} \vdash s : B}$$

where the relation $\mathcal{G}, A$ is inductively defined by the following rules

$$\frac{\mathcal{G}\, s\, B}{(\mathcal{G}, A)\; s[\uparrow]\; B[\uparrow]} \qquad\qquad \frac{}{(\mathcal{G}, A)\; 0\; A[\uparrow]}$$

and the operation $m \oplus n$ on universe levels is defined by

$$m \oplus n \;:=\; \begin{cases} -1, & \text{if } n = -1 \\ \max\, m\, n, & \text{otherwise} \end{cases}$$

Furthermore, we say that a term A **is a type in context** $\mathcal{G}$ (written $\mathcal{G} \vdash A$) whenever there exists some α such that $\mathcal{G} \vdash A : U_\alpha$.

Intuitively, the operation "$\mathcal{G}, A$" stands for the extension of the context $\mathcal{G}$ by a new

bound variable with a specified type A. The operation $m \oplus n$ on universe levels expresses the fact that the universe of propositions U_{-1} is impredicative, that is, $\Pi A\ B$ is in U_{-1} whenever B is. We will occasionally need to refer to the empty context which we write as "$\cdot$".

The pretyping judgment is deficient in that it admits an arbitrary hypothetical judgments $\mathcal{G}$ to appear as a context. With the right context, this allows us to derive arbitrary types. In particular, pretyping does not have the subject reduction property in general, since the context $\mathcal{G}$ might not have this property.

Nevertheless, pretyping has excellent structural properties, and if we restrict ourselves to well-formed contexts we can recover the full typing judgment of CC_ω. While not strictly necessary, we also restrict attention to finite contexts, since this matches the usual presentations of type theory.

Definition 4.3 A context $\mathcal{G}$ is **well-formed** (written $\mathcal{G} \vdash$) if it can be derived inductively using the following rules.

$$\frac{}{\cdot \vdash} \qquad\qquad \frac{\mathcal{G} \vdash \qquad \mathcal{G} \vdash A}{\mathcal{G}, A \vdash}$$

Note that this matches the usual definition of context well-formedness [78].

Our main goal in this chapter is to show that typing in well-formed contexts has the subject reduction property. For all terms s, A and well-formed contexts $\mathcal{G} \vdash$ such that $\mathcal{G} \vdash s : A$ we have $\mathcal{G} \vdash t : A$ whenever $s \rhd t$.

For this result it is crucial that the subtyping relation and therefore also conversion and reduction are well-behaved. Thus we begin out investigation by showing the Church-Rosser theorem for reduction (Chapter 4.2). The next ingredient of the proof of subject reduction is the compatibility of typing with instantiation, which holds already for pretyping and takes on a particularly simple form in this context (Chapter 4.4). From these lemmas we can then conclude both soundness of typing (Lemma 4.39) as well as subject reduction (Theorem 4.40).

4.2 The Church-Rosser Theorem for CC_ω

The Church-Rosser theorem [33] for CC_ω states that reduction has the Church-Rosser property, or equivalently, that reduction is confluent. In particular, this implies that two terms are convertible $A \equiv B$ if and only if they have a common reduct C, i.e., a term C such that $A \rhd^* C$ and $B \rhd^* C$.

Since confluence is a property of the reflexive, transitive closure of reduction, we begin by giving a different characterization of $_ \rhd^* _$ as the reflexive, transitive

closure of a better behaved parallel reduction relation.

Definition 4.4 The **parallel reduction relation** $s \Rrightarrow t$ is defined inductively by the following rules.

$$\beta\text{-REDUCTION}\quad \frac{s \Rrightarrow s' \qquad t \Rrightarrow t'}{(\lambda\, s)\, t \Rrightarrow s'[t' \cdot \mathsf{id}]} \qquad \text{VAR}\quad \frac{}{x \Rrightarrow x} \qquad \text{APP}\quad \frac{s \Rrightarrow s' \qquad t \Rrightarrow t'}{s\, t \Rrightarrow s'\, t'} \qquad \text{LAM}\quad \frac{s \Rrightarrow s'}{\lambda\, s \Rrightarrow \lambda\, s'}$$

$$\text{PI}\quad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\Pi A\ B \Rrightarrow \Pi A'\ B'} \qquad \text{U}\quad \frac{}{U_\alpha \Rrightarrow U_\alpha}$$

We extend parallel reduction pointwise to substitutions and set

$$\sigma \Rrightarrow \tau := \forall x.\ \sigma\, x \Rrightarrow \tau\, x.$$

The **maximal parallel reduction** of a term s is the term $\rho\, s$ defined by recursion as follows.

$$\begin{aligned}
\rho\, x &:= x \\
\rho((\lambda\ s)\, t) &:= (\rho\, s)[\rho\, t \cdot \mathsf{id}] \\
\rho(s\, t) &:= \rho\, s\, (\rho\, t) && \text{for } s \neq \lambda\, b \text{ for all } b \\
\rho(\lambda\, s) &:= \lambda\, \rho\, s \\
\rho(\Pi A\ B) &:= \Pi \rho\, A\ \rho\, B \\
\rho(U_\alpha) &:= U_\alpha
\end{aligned}$$

Fact 4.5 Parallel reduction is reflexive.

Proof We show $s \Rrightarrow s$ by induction on s, using the congruence rules VAR, APP, LAM, PI, U. ∎

Intuitively, parallel reduction is an extended reduction relation which permits the contraction of several visible redexes at the same time. Maximal parallel reduction recursively contracts all visible redexes.

It is relatively easy to see that parallel reduction is confluent, and in fact has the stronger diamond property. The technical reason why parallel reduction is so much better behaved than ordinary reduction is that it is substitutive in a stronger sense than what is true for ordinary reduction. First recall the usual notion of substitutivity.

Lemma 4.6 For all substitutions σ and terms s, t such that $s \Rrightarrow t$ we have $s[\sigma] \Rrightarrow t[\sigma]$.

Proof Analogous to the proof of Lemma 3.7, using Fact 4.5. ■

Corollary 4.7 $\sigma \Rrightarrow \tau$ implies $\Uparrow\sigma \Rrightarrow \Uparrow\tau$.

The new feature of parallel substitution is that it also allows reduction inside of substitutions. This fails for ordinary reduction, since a substitution might instantiate the same variable several times, while we can only take a single step at a time. For example we have $\sigma := ((\lambda\, 0) s \cdot \mathsf{id}) \rhd (s \cdot \mathsf{id}) =: \tau$, but $(0\,0)[\sigma] = ((\lambda\, 0) s)\, ((\lambda\, 0) s) \not\rhd s\, s = (0\,0)[\tau]$. Meanwhile, we do have $(0\,0)[\sigma] \Rrightarrow (0\,0)[\tau]$ for parallel reduction.

Lemma 4.8 For all substitutions σ, τ such that $\sigma \Rrightarrow \tau$ and terms s, t such that $s \Rrightarrow t$ we have $s[\sigma] \Rrightarrow t[\tau]$.

Proof By induction on the derivation of $s \Rrightarrow t$.

- U: Trivial.
- VAR: Follows by assumption from $\sigma \Rrightarrow \tau$.
- APP: Follows from the inductive hypotheses.
- LAM, PI: Follows from the inductive hypotheses and the fact that $\sigma \Rrightarrow \tau$ implies $\Uparrow\sigma \Rrightarrow \Uparrow\tau$ which in turn follows from Lemma 4.6.
- β-REDUCTION: We have

  $$((\lambda\, s)\, t)[\sigma] = (\lambda\, s[\Uparrow\sigma])\, t[\sigma] \Rrightarrow s'[\Uparrow\tau][t'[\tau] \cdot \mathsf{id}]$$

  by the inductive hypothesis since $\Uparrow\sigma \Rrightarrow \Uparrow\tau$ and therefore $s[\Uparrow\sigma] \Rrightarrow s'[\Uparrow\tau]$ and similarly $t[\sigma] \Rrightarrow t'[\tau]$. The statement then follows from the rules of the σ_{SP}-calculus, since we have $s'[\Uparrow\tau][t'[\tau] \cdot \mathsf{id}] = s'[t'[\tau] \cdot \tau] = s'[t' \cdot \mathsf{id}][\tau]$. ■

Lemma 4.9 For all terms s, t such that $s \Rrightarrow t$ we have $t \Rrightarrow \rho\, s$.

Proof By induction on the derivation of $s \Rrightarrow t$. The cases for VAR, APP, LAM, PI follow immediately from the inductive hypotheses. In the case of β-REDUCTION we have $s = (\lambda\, b)\, a$ and $t = b'[a' \cdot \mathsf{id}]$ where $a \Rrightarrow a'$ and $b \Rrightarrow b'$. By induction we know that $a' \Rrightarrow \rho\, a$ and $b' \Rrightarrow \rho\, b$ which together with Lemma 4.8 implies that $t = b'[a' \cdot \mathsf{id}] \Rrightarrow (\rho\, b')[\rho\, a' \cdot \mathsf{id}] = \rho\, s$. ■

Lemma 4.10 Parallel reduction is confluent.

Proof From Lemma 4.9 we conclude that parallel reduction has the diamond property and hence by Corollary 2.14 that parallel reduction is confluent. ∎

The confluence of parallel reduction in turn implies the confluence of reduction, since $\triangleright^*$ and $\mathrel{\triangleright\!\!\triangleright}^*$ are the same relation. Showing this formally is easy, if somewhat tedious. We begin by noting some congruence properties of $_ \triangleright^* _$.

Fact 4.11 Multi-step reduction is a congruence.

(1) $s \triangleright^* s'$, $t \triangleright^* t'$ imply $s\,t \triangleright^* s'\,t'$ and $\Pi s\; t \triangleright^* \Pi s'\; t'$

(2) $s \triangleright^* s'$ implies $\lambda\, s \triangleright^* \lambda\, s'$

Proof By Fact 2.4 and Fact 2.3. ∎

Lemma 4.12 (Substitutivity of $\triangleright$) If $s \triangleright t$ then $s[\sigma] \triangleright t[\sigma]$.

Proof Analogous to Lemma 3.7. ∎

Lemma 4.13 (Strong Substitutivity of $\triangleright^*$) If $s \triangleright^* t$ and $\sigma\, x \triangleright^* \tau\, x$ for all x then $s[\sigma] \triangleright^* t[\tau]$.

Proof By Fact 2.4 and Lemma 4.12 it suffices to show that $t[\sigma] \triangleright^* t[\tau]$. This in turn follows by induction on t using Fact 4.11. ∎

Next we show that $_ \mathrel{\triangleright\!\!\triangleright} _$ lies between $_ \triangleright _$ and $_ \triangleright^* _$ as a relation.

Lemma 4.14 $s \triangleright t$ implies $s \mathrel{\triangleright\!\!\triangleright} t$

Proof By induction on the derivation of $s \triangleright t$. Each case follows immediately from the inductive hypothesis together with Fact 4.5. ∎

Lemma 4.15 $s \mathrel{\triangleright\!\!\triangleright} t$ implies $s \triangleright^* t$

Proof By induction on the derivation of $s \mathrel{\triangleright\!\!\triangleright} t$ using Fact 4.11 and Lemma 4.13. ∎

This implies that we can lift confluence of $_ \mathrel{\triangleright\!\!\triangleright} _$ to the confluence of $_ \triangleright _$.

Theorem 4.16 Reduction has the Church-Rosser property.

Proof We first show that $\triangleright^*$ and $\mathrel{\triangleright\!\!\triangleright}^*$ are equivalent relations. By Lemma 4.14 we have $\triangleright \subseteq \mathrel{\triangleright\!\!\triangleright} \subseteq \mathrel{\triangleright\!\!\triangleright}^*$ and hence by Fact 2.5 we have $\triangleright^* \subseteq \mathrel{\triangleright\!\!\triangleright}^*$. In the reverse direction, by Lemma 4.15 we have $\mathrel{\triangleright\!\!\triangleright} \subseteq \triangleright^*$ and by Fact 2.5 we have $\mathrel{\triangleright\!\!\triangleright}^* \subseteq \triangleright^*$. Together these imply that $\mathrel{\triangleright\!\!\triangleright}^* = \triangleright^*$.

Now since parallel reduction is confluent by Lemma 4.10 and confluence is a property of $\mathrel{\triangleright\!\!\triangleright}^*$ it follows that reduction is confluent. But confluence and the Church-Rosser property are equivalent by Lemma 2.15 and hence reduction has the Church-Rosser property. ∎

4.3 Properties of Subtyping

The Church-Rosser property implies that conversion and subtyping are well-behaved.

Fact 4.17 (Type Inversions)

(1) For all A, B, C such that $\Pi A\ B \rhd C$ there are A', B' such that $C = \Pi A'\ B'$, $A \rhd^* A'$, and $B \rhd^* B'$.

(2) For all A, B, C such that $\Pi A\ B \rhd^* C$ there are A', B' such that $C = \Pi A'\ B'$, $A \rhd^* A'$, and $B \rhd^* B'$.

(3) There is no A such that $U_\alpha \rhd A$.

(4) For all A such that $U_\alpha \rhd^* A$ we have $A = U_\alpha$.

Lemma 4.18 (Injectivity of Products) $\Pi A\ B \equiv \Pi A'\ B'$ implies $A \equiv A'$ and $B \equiv B'$.

Proof By Theorem 4.16 there is some term C such that $\Pi A\ B \rhd^* C$ and $\Pi A'\ B' \rhd^* C$. By Fact 4.17 and injectivity of constructors there are A'', B'' such that $C = \Pi A''\ B''$, $A \rhd^* A''$, $A' \rhd^* A''$, $B \rhd^* B''$, $B' \rhd^* B''$. Using Fact 2.7 together with the symmetry and transitivity of $\equiv$ we have $A \equiv A'$ and $B \equiv B'$ as required. ∎

Lemma 4.19 (Injectivity of Universes) $U_\alpha \equiv U_\beta$ implies that $\alpha = \beta$.

Proof By Theorem 4.16 there is some term A such that $U_\alpha \rhd^* A$ and $U_\beta \rhd^* A$, and by Fact 4.17 this implies that $U_\alpha = A = U_\beta$. ∎

Lemma 4.20 (Disjointness of Products and Universes) $\Pi A\ B \not\equiv U_\alpha$

Proof By Theorem 4.16 there is some term C such that $\Pi A\ B \rhd^* C$ and $U_\alpha \rhd^* C$, but by Fact 4.17 this implies that $U_\alpha = C = \Pi A'\ B'$ for some A', B' which is impossible. ∎

These lemmas in turn imply that subtyping is well-behaved, and in particular that subtyping is transitive.

Lemma 4.21 Subtyping is transitive, $A <: B$ and $B <: C$ imply $A <: C$.

Proof By the definition of subtyping there are A', B', B'', C' such that

$$A \equiv A' <:_1 B' \equiv B \equiv B'' <:_1 C' \equiv C.$$

The statement follows if we can show that type inclusion is transitive in the sense that whenever $A <:_1 B \equiv C <:_1 D$ then $A <: D$. We show this by induction on the derivation of $A <:_1 B$.

- In the case of reflexivity we have $A = B$ and hence $A <: D$ by definition of subtyping.
- In the case of universes we have $A = U_m$, $B = U_n$ with $m \leq n$. Now consider the derivation of $C <:_1 D$. If this derivation is by reflexivity, then $C = D$ and the statement again follows by the definition of subtyping. It is impossible to encounter the case for products at this point since then we would have $U_n \equiv \Pi A'\ B'$ which is impossible by Lemma 4.20. Finally, if we have the case of universes then $C = U_n$ by Lemma 4.19 and $D = U_k$ with $n \leq k$. Since we have $m \leq k$ then in particular $U_m <:_1 U_k$ holds and the statement follows.
- In the case of products we have $A = \Pi A'\ B'$, $B = \Pi A'\ B''$ with $B' <:_1 B''$. Consider the derivation of $C <:_1 D$. In the case of reflexivity we have $C = D$ and the statement follows by the definition of subtyping. The case for universes is again impossible by Lemma 4.20. It follows that we must have derived $C <:_1 D$ using the product rule which implies that $C = \Pi A''\ C'$, $D = \Pi A''\ C''$, and $C' <:_1 C''$. Since we have $\Pi A'\ B'' = B \equiv C = \Pi A''\ C'$ we have $A' \equiv A''$ and $B'' \equiv C'$ by Lemma 4.18. In particular we have $B' <:_1 B'' \equiv C' <:_1 C''$ and hence $B' <: C''$ by the inductive hypothesis. Thus there are some D', D'' such that $B' \equiv D' <:_1 D'' \equiv C''$ and by the product rule for $<:_1$ and the congruence property of $\equiv$ we have $A = \Pi A'\ B' \equiv \Pi A'\ D' <:_1 \Pi A'\ D'' \equiv \Pi A''\ C'' = D$ and hence $A <: D$. ∎

Furthermore, there are inversion lemmas for subtyping which follow from the corresponding inversion lemmas for conversion. We only need the case for products.

Lemma 4.22 If $\Pi A\ B <: \Pi A'\ B'$ then $A \equiv A'$ and $B <: B'$.

Proof By the definition of subtyping and Fact 4.17 (2), there are C, C', D, D' such that $A \equiv C$, $B \equiv D$, $A' \equiv C'$, $B' \equiv D'$ and $\Pi C\ D <:_1 \Pi C'\ D'$. By the definition of type inclusion this implies $C = C'$ and $D <:_1 D'$. In particular, we have $A \equiv C = C' \equiv A'$ and so $A \equiv A'$, as well as $B \equiv D <:_1 D' \equiv B'$ and hence $B <: B'$. ∎

Finally, conversion and subtyping are substitutive.

Lemma 4.23 We have $A[\sigma] \equiv B[\sigma]$ whenever $A \equiv B$.

Proof By Fact 2.10 and Lemma 4.12. ∎

Lemma 4.24 We have $A[\sigma] <: B[\sigma]$ whenever $A <: B$.

Proof Conversion is substitutive by Lemma 4.23. It suffices to show that type inclusion is substitutive. We show that $A <:_1 B$ implies $A[\sigma] <:_1 B[\sigma]$ by induction on the derivation of $A <:_1 B$. The case of reflexivity and universes are trivial, in the case of products we have $A = \Pi C\ D$, $B = \Pi C\ D'$, and $D <:_1 D'$. By the inductive hypothesis we have $D[\Uparrow\sigma] <:_1 D'[\Uparrow\sigma]$ and hence

$$A[\sigma] = \Pi C[\sigma]\ D[\Uparrow\sigma] <:_1 \Pi C[\sigma]\ D'[\Uparrow\sigma] = B[\sigma].$$

■

4.4 Properties of Pretyping Judgments

In this section we show that the pretyping judgment is compatible with instantiation. To this end we extend the pretyping judgment to substitutions.

Definition 4.25 A substitution σ maps a context $\mathcal{G}$ into $\mathcal{G}'$ (written $\sigma : \mathcal{G} \to \mathcal{G}'$) if

$$\forall s A.\ \ \mathcal{G}\ s\ A\ \ \to\ \ \mathcal{G}'\ s[\sigma]\ A[\sigma]$$

In this way substitutions can be seen as maps of contexts.

Fact 4.26 For all contexts $\mathcal{G}, \mathcal{G}', \mathcal{G}''$ and substitutions σ, τ we have

(1) $\mathsf{id}\ :\ \mathcal{G} \to \mathcal{G}$

(2) $\sigma \circ \tau\ :\ \mathcal{G} \to \mathcal{G}''$ whenever $\sigma\ :\ \mathcal{G} \to \mathcal{G}'$ and $\tau\ :\ \mathcal{G}' \to \mathcal{G}''$

(3) $\uparrow\ :\ \mathcal{G} \to \mathcal{G}, A$

(4) $s \cdot \sigma\ :\ \mathcal{G}, A \to \mathcal{G}$ whenever $\sigma\ :\ \mathcal{G} \to \mathcal{G}'$ and $\mathcal{G}'\ s\ A[\sigma]$

Proof (1) By definition.

(2) By Lemma 3.5.

(3) By the definition of $\mathcal{G}, A$.

(4) For all $(\mathcal{G}, A)\ t\ B$ we have to show $\mathcal{G}\ t[s \cdot \sigma]\ B[s \cdot \sigma]$. We proceed by case analysis on the derivation of $(\mathcal{G}, A)\ t\ B$. There are two cases to consider.
In the first case we have $t = 0$ and $B = A[\uparrow]$. We have

$$\mathcal{G}\ 0[s \cdot \sigma]\ A[\uparrow][s \cdot \sigma]\ \ =\ \ \mathcal{G}\ s\ A[\sigma]$$

which holds by assumption.

In the second case we have $\mathcal{G}\ t'\ B'$ with $t = t'[\uparrow]$ and $B = B'[\uparrow]$. We have

$$\mathcal{G}\ t'[\uparrow][s \cdot \sigma]\ B'[\uparrow][s \cdot \sigma] \;=\; \mathcal{G}\ t'[\sigma]\ B'[\sigma]$$

which holds by assumption on σ. ■

We begin by showing that the pretyping judgment respects context maps. The crux of the proof is that context maps are compatible with context extension.

Lemma 4.27 If $\sigma : \mathcal{G} \to \mathcal{G}'$ then $\Uparrow\sigma : (\mathcal{G}, A) \to (\mathcal{G}', A[\sigma])$.

Proof By Fact 4.26 (4) it suffices to show that $(\mathcal{G}', A[\sigma])\ 0\ A[\sigma \circ \uparrow]$ and $\sigma \circ \uparrow\ :\ \mathcal{G} \to (\mathcal{G}', A[\sigma])$. The former follows from the definition of $(\mathcal{G}', A[\sigma])$, the latter follows from Fact 4.26 (2) and (3). ■

Lemma 4.28 For $\mathcal{G} \vdash s : A$ and $\sigma : \mathcal{G} \to \mathcal{G}'$ we have $\mathcal{G}' \vdash s[\sigma] : A[\sigma]$.

Proof By induction on the derivation of $\mathcal{G} \vdash s : A$.

- VAR: This case is equivalent to the assumption on σ.
- SORT: Trivial.
- APP: By the inductive hypothesis we have $\mathcal{G}' \vdash s[\sigma] : \Pi A[\sigma]\ B[\Uparrow\sigma]$ and $\mathcal{G}' \vdash t[\sigma] : A[\sigma]$. It follows that we have $\mathcal{G}' \vdash (s\,t)[\sigma] : (B[\Uparrow\sigma])[t[\sigma] \cdot \mathsf{id}] = B[(t[\sigma]) \cdot \sigma]$.
- LAM: By the inductive hypothesis and Lemma 4.27 we have $\mathcal{G}' \vdash A[\sigma] : U_n$ and $\mathcal{G}', A[\sigma] \vdash s[\Uparrow\sigma] : B[\Uparrow\sigma]$ and hence $\mathcal{G}' \vdash (\lambda\, s)[\sigma] : (\Pi A\ B)[\sigma]$.
- PI: By the inductive hypothesis and Lemma 4.27 we have $\mathcal{G}' \vdash A[\sigma] : U_m$ and $\mathcal{G}', A[\sigma] \vdash B[\Uparrow\sigma] : U_n$ and hence $\mathcal{G}' \vdash \Pi A\, B : U_{m \oplus n}$.
- SUB: By the inductive hypothesis we have $\mathcal{G}' \vdash s[\sigma] : A[\sigma]$ and $\mathcal{G}' \vdash B[\sigma] : U_n$. Furthermore, by Lemma 4.24 we have $A[\sigma] <: B[\sigma]$ and hence $\mathcal{G}' \vdash s[\sigma] : B[\sigma]$.

■

With some additional definitions we can see Lemma 4.28 as another property of context maps.

Definition 4.29 For a context $\mathcal{G}$, we denote by $\mathrm{T}\,\mathcal{G}$ the context of terms typable under $\mathcal{G}$.

$$\mathrm{T}\,\mathcal{G}\ s\ A \;:=\; \mathcal{G} \vdash s : A$$

We say that a substitution σ is a **context morphism** from $\mathcal{G}$ to $\mathcal{G}'$ if $\sigma : \mathcal{G} \to \mathrm{T}\,\mathcal{G}'$.

Using this notation we can summarize Lemma 4.28 as saying that $\sigma : \mathrm{T}\,\mathcal{G} \to \mathrm{T}\,\mathcal{G}'$ whenever $\sigma : \mathcal{G} \to \mathcal{G}'$. Extending the shift renaming in this way yields the classical weakening statement for typing.

Corollary 4.30 (Weakening) If $\mathcal{G} \vdash s : B$ then $\mathcal{G}, A \vdash s[\uparrow] : B[\uparrow]$.

Proof By Lemma 4.28 and Fact 4.26 (3). ■

Furthermore, weakening implies that context morphisms are compatible with context extension.

Lemma 4.31 If $\sigma : \mathcal{G} \to \mathrm{T}\,\mathcal{G}'$ then $\Uparrow\sigma : (\mathcal{G}, A) \to \mathrm{T}(\mathcal{G}', A[\sigma])$.

Proof By Lemma 4.27 we have $\Uparrow\sigma : (\mathcal{G}, A) \to (\mathrm{T}\,\mathcal{G}', A[\sigma])$ and it suffices to show that $\mathsf{id} = 0 \cdot \uparrow : (\mathrm{T}\,\mathcal{G}', A[\sigma]) \to \mathrm{T}(\mathcal{G}', A[\sigma])$.

By Fact 4.26 (4) this follows from $\uparrow : \mathrm{T}\,\mathcal{G}' \to \mathrm{T}(\mathcal{G}', A[\sigma])$ and $\mathrm{T}(\mathcal{G}', A[\sigma])\, 0\, A[\sigma][\uparrow]$. The former is precisely Corollary 4.30, while the later follows by VAR and the definition of $(\mathcal{G}', A[\sigma])$. ■

In turn,Lemma 4.31 allows us to adapt the proof of Lemma 4.28 for T.

Lemma 4.32 For $\mathcal{G} \vdash s : A$ and $\sigma : \mathcal{G} \to \mathrm{T}\,\mathcal{G}'$ we have $\mathcal{G}' \vdash s[\sigma] : A[\sigma]$.

Proof Exactly like the proof of Lemma 4.28, but using Lemma 4.31 in place of Lemma 4.27. ■

Another way to read Lemma 4.32 is that $\sigma : \mathcal{G} \to \mathrm{T}\,\mathcal{G}'$ implies $\sigma : \mathrm{T}\,\mathcal{G} \to \mathrm{T}\,\mathcal{G}'$.

There are two important special cases of Lemma 4.32. The first concerns single variable substitutions, the second conversion in the context.

Corollary 4.33 If $\mathcal{G}, A \vdash s : B$ and $\mathcal{G} \vdash t : A$ then $\mathcal{G} \vdash s[t \cdot \mathsf{id}] : B[t \cdot \mathsf{id}]$.

Proof The assumption $\mathcal{G} \vdash t : A$ implies that $t \cdot \mathsf{id} : (\mathcal{G}, A) \to \mathrm{T}\,\mathcal{G}$ by Fact 4.26 and the result follows by Lemma 4.32. ■

Corollary 4.34 If $\mathcal{G}, A \vdash s : C$, $A \equiv B$, and $\mathcal{G} \vdash A$ then $\mathcal{G}, B \vdash s : C$.

Proof By SUB and the assumptions we have $\mathsf{id} = 0 \cdot \uparrow : (\mathcal{G}, A) \to (\mathcal{G}, B)$ by Fact 4.26 and the result follows from Lemma 4.32. ■

4.5 Type Soundness and Subject Reduction

Previously we developed the theory of CC_ω with respect to an abstract context $\mathcal{G}$ representing an arbitrary hypothetical judgment. In order to show type soundness or preservation we need to restrict attention to well-typed contexts. Hence all contexts in this section will be well-formed. We write Γ, Δ for well-formed contexts.

Lemma 4.35 Let Γ be a well-formed context. Then $\Gamma\ s\ A$ implies that s is a variable.

Proof By induction on the derivation of $\Gamma \vdash$. ∎

Typing in well-formed contexts is generally better behaved than under arbitrary contexts. For example, we obtain inversion lemmas that state that whenever a term has a type then so do all of its subterms. This is not the case for typing in arbitrary contexts, since a context $\mathcal{G}$ itself might not have this property. In the following we need two inversion lemmas, one for products and one for abstractions.

Lemma 4.36 Let Γ be a well-formed context. If $\Gamma \vdash \Pi A\ B$ then $\Gamma \vdash A$ and $\Gamma, A \vdash B$.

Proof We show that whenever $\Gamma \vdash \Pi A\ B : C$ then $\Gamma \vdash A$ and $\Gamma, A \vdash B$ by induction on the derivation of $\Gamma \vdash \Pi A\ B : C$.

Since Γ is well-formed there are only two cases to consider by Lemma 4.35. In case Pi, the statement follows immediately by assumption. In case Sub, the statement follows by the inductive hypothesis. ∎

Lemma 4.37 Let Γ be a well-formed context. If $\Gamma \vdash \lambda s : \Pi A\ B$ then $\Gamma, A \vdash s : B$.

Proof The proof is complicated by the presence of the subtyping rule. There could be an unbounded number of application of the subtyping rule in the derivation of $\Gamma \vdash \lambda s : \Pi A\ B$ before an application of the Lam rule. Each application of the subtyping rule can in turn change the type away from $\Pi A\ B$, although we must ultimately end up with some type $\Pi C\ D$ in order to apply the Lam rule.

In this case we need to apply the subtyping rule again at the end, which requires that B is a type in context Γ, A. However, we only obtain this assumption from an application of the subtyping rule. Thus there are two cases to consider. Either we encounter the Lam rule directly, in which case we can close the proof without further assumptions, or we have encountered an application of the Sub rule, in which case we must remember that B is well-formed.

Formally, we show the following statement. If $\Gamma \vdash \lambda s : C$ where either $C = \Pi A\ B$ or $C <: \Pi A\ B$ and $\Gamma, A \vdash B$ then $\Gamma, A \vdash s : B$. The proof proceeds by induction on the derivation of $\Gamma \vdash \lambda s : C$. There are two cases to consider, since the variable case is contradictory by Lemma 4.35.

- In the case of the rule LAM, if $C = \Pi A\ B$ then we have $\Gamma, A \vdash s : B$ by assumption. Otherwise we have $\Gamma, A' \vdash s : B'$ and $\Gamma \vdash A'$ for A', B' such that $\Pi A'\ B' <: \Pi A\ B$ and $\Gamma, A \vdash B$. By Lemma 4.22 this implies that $A' \equiv A$ and $B' <: B$. By Corollary 4.34 we have $\Gamma, A \vdash s : B'$ and finally using the subtyping rule we have $\Gamma, A \vdash s : B$ as claimed.

- In the case of the subtyping rule, if $C = \Pi A\ B$ then we have some D such that $\Gamma \vdash \lambda s : D$, $D <: \Pi A\ B$, and $\Gamma \vdash \Pi A\ B$. By Lemma 4.36 this implies $\Gamma, A \vdash B$ and the statement follows by induction. Otherwise we have $C <: \Pi A\ B$, $\Gamma, A \vdash B$ and some D such that $\Gamma \vdash \lambda s : D$, and $D <: C$. By transitivity of subtyping (Lemma 4.21) this implies that $D <: \Pi A\ B$ and the statement follows by induction. ∎

Type soundness states that whenever $\Gamma \vdash s : A$, then A is a type in context Γ. We first show that well-formed contexts are sound in this sense and then obtain the full type soundness result.

Lemma 4.38 $\Gamma\ s\ A$ implies $\Gamma \vdash A$ for well-formed contexts Γ.

Proof By induction on the derivation of $\Gamma \vdash$. Since there is no s, A such that $\Gamma\ s\ A$ holds in the empty context the only case to consider is for contexts of the form Γ, B. Now we have $(\Gamma, B)\ s\ A$ either because $A = B[\uparrow]$, or because $A = A'[\uparrow]$ with $s = s'[\uparrow]$ and $\Gamma\ s'\ A'$. In the former case we have $\Gamma \vdash B$ by assumption, in the latter case we have $\Gamma \vdash A'$ by the inductive hypothesis. Either way the statement follows by Corollary 4.30. ∎

Lemma 4.39 (Type Soundness) $\Gamma \vdash s : A$ implies $\Gamma \vdash A$ for well-formed contexts Γ.

Proof By induction on the derivation of $\Gamma \vdash s : A$.

- VAR: By Lemma 4.38.

- SORT,PI: We have $\Gamma \vdash U_\alpha$ for all α.

- APP: We have $\Gamma \vdash s : \Pi A\ B$ and $\Gamma \vdash t : A$ by assumption and $\Gamma \vdash \Pi A\ B$ by the inductive hypothesis. We need to show $\Gamma \vdash B[t \cdot \mathsf{id}]$. By Lemma 4.36 we have $\Gamma, A \vdash B$, and the statement follows with Corollary 4.33.

- LAM: We have $\Gamma \vdash A$ and $\Gamma, A \vdash s : B$ by assumption and $\Gamma, A \vdash B$ by the inductive hypothesis. The statement follows by the typing rule for products.

- SUB: The statement follows by the assumption of the subtyping rule. ∎

Finally putting all the pieces together we show that typing is preserved under reduction.

Theorem 4.40 (Type Preservation) Let Γ be a well-formed context. If $\Gamma \vdash s : A$ and $s \rhd s'$, then $\Gamma \vdash s' : A$.

Proof By induction on the derivation of $\Gamma \vdash s : A$.

- VAR: By Lemma 4.35 s is a variable and hence normal.
- SORT: There is no t such that $U_\alpha \rhd t$.
- APP: We have $\Gamma \vdash s : \Pi A\ B$ and $\Gamma \vdash t : A$ by assumption. For all u such that $s\,t \rhd u$ we have to show that $\Gamma \vdash u : B[t \cdot \mathsf{id}]$. There are three cases to consider for the derivation of $s\,t \rhd u$.
  - If $u = s'\,t$ where $s \rhd s'$ we have $\Gamma \vdash s' : \Pi A\ B$ by the inductive hypothesis and thus $\Gamma \vdash s'\,t : B[t \cdot \mathsf{id}]$ by APP.
  - If $u = s\,t'$ where $t \rhd t'$ we have $\Gamma \vdash t' : A$ by the inductive hypothesis and $\Gamma \vdash s\,t' : B[t' \cdot \mathsf{id}]$ by APP. The statement follows by the subtyping rule since we have $B[t' \cdot \mathsf{id}] \equiv B[t \cdot \mathsf{id}]$ and $\Gamma \vdash B[t \cdot \mathsf{id}]$ by Lemma 4.39 applied to the derivation of $\Gamma \vdash s\,t : B[t \cdot \mathsf{id}]$.
  - If $s = \lambda\, b$, $u = b[t \cdot \mathsf{id}]$ we have $\Gamma, A \vdash b : B$ by Lemma 4.37 and the statement follows from Corollary 4.33.
- LAM: We have $\Gamma \vdash A$ and $\Gamma, A \vdash s : B$, and need to show that for all u such that $\lambda\, s \rhd u$ we have $\Gamma \vdash u : \Pi A\ B$. When $\lambda\, s \rhd u$ there is some u' such that $s \rhd u'$ and $u = \lambda\, u'$. By LAM it suffices to show that $\Gamma, A \vdash u' : B$, but this follows from the inductive hypothesis.
- PI: We have $\Gamma \vdash A$ and $\Gamma, A \vdash B$ and need to show that whenever $\Pi A\ B \rhd u$ we have $\Gamma \vdash u$. There are two cases to consider.
  - If $A \rhd A'$ and $u = \Pi A'\ B$ we have $\Gamma \vdash A'$ by the inductive hypothesis and $\Gamma, A' \vdash B$ by Corollary 4.34. The statement follows by PI.
  - If $B \rhd B'$ and $u = \Pi A\ B'$ we have $\Gamma, A \vdash B'$ by the inductive hypothesis and the statement follows by PI.
- SUB: Since the term under consideration does not change this follows immediately by induction. ∎

4.6 Discussion

Definitions Following Luo [78], we consider a "Russel-style" presentation for CC_ω, which does not distinguish between terms and types. This makes subject reduction a very delicate affair. It is possible to add dependent sum types with the following rules and the evident reduction rules to the system while retaining subject reduction.

$$\frac{A <:_1 A' \qquad B <:_1 B'}{\Sigma A\, B <:_1 \Sigma A'\, B'} \qquad \frac{\mathcal{G} \vdash A : U_\alpha \qquad \mathcal{G}, A \vdash B : U_\beta \qquad \alpha, \beta \geq 0}{\mathcal{G} \vdash \Sigma A\, B : U_{\alpha \oplus \beta}}$$

$$\frac{\mathcal{G} \vdash s : A \qquad \mathcal{G} \vdash t : B[s \cdot \mathsf{id}]}{\mathcal{G} \vdash (s, t) : \Sigma A\, B}$$

$$\frac{\mathcal{G} \vdash s : \Sigma A\, B}{\mathcal{G} \vdash \pi_1\, s : A} \qquad \frac{\mathcal{G} \vdash s : \Sigma A\, B}{\mathcal{G} \vdash \pi_2\, s : B[\pi_1\, s \cdot \mathsf{id}]}$$

This leads to the extended calculus of construction considered by Luo [78]. It is, however, equally possible to add dependent sum types and destroy the subject reduction property by replacing the last two rules with the induction principle for dependent sums.

$$\frac{\mathcal{G}, \Sigma A\, B \vdash P : U_\alpha \qquad \mathcal{G} \vdash s : \Sigma A\, B \qquad \mathcal{G}, A, B \vdash p : P[(1, 0) \cdot \mathsf{id}]}{\mathcal{G} \vdash \Sigma\mathsf{rec}_P\; p\; s : P[s \cdot \mathsf{id}]}$$

This phenomenon is not specific to dependent sum types. Rather, the same problem occurs with any inductive container in a Russel-style type theory in the presence of cumulativity [79]. As noted by Luo, a "Tarski-style" type theory which distinguishes between terms and types does not suffer from this deficiency.

Additionally, a Tarski-style type theory would permit us to work with intrinsically typed terms, assuming the existence of quotient inductive inductive types [9]. A Russel-style type theory like the one we consider in this chapter forces us to use untyped terms with a separate typing judgment.

Finally, our two-level definition of subtyping is different from Luo's definition of subtyping by transfinite iteration. A faithful translation of Luo's definition is possible with an inductive definition of subtyping which builds in transitivity and conversion. However, we find it more direct to start with a two-level definition of subtyping and only showing the correctness of this definition (in particular transitivity) after the fact. After all results are established (compare from Chapter 4.3, in particular

reflexivity, transitivity, and compatibility with conversion) it is clear that the two definitions are equivalent.

Relational Typing and Context Morphisms. We first considered a relational typing judgment in [66] for a syntax directed version of System F typing. Since this syntax directed type system forms a model of the syntax of System F it has excellent structural properties. The pretyping judgment of CC_ω is not syntax directed and even more permissive since it is not restricted to finite contexts, yet it retains the excellent structural properties that we had previously observed.

When restricted to well-formed contexts Lemma 4.32 is exactly the "context morphism lemma" of Goguen and McKinna [54]. Our proof of the context morphism lemma is unusual in that we never have to restrict context morphisms to renamings. The more primitive notion of context maps, which is only visible with a relational typing judgment, replaces the usual consideration of renamings.

Context maps also turn out to have very pleasant structural properties (Fact 4.26, Lemma 4.28, Lemma 4.32, . . .). We can summarize the situation very concisely in the language of category theory.

Consider a category whose objects are hypothetical judgments, i.e., relations $\mathbb{E}_n \to \mathbb{E}_n \to \mathcal{P}$, and whose morphisms are context maps. This is a category by Fact 4.26. In this setting, Lemma 4.28 and Lemma 4.32 show that pretyping is a monad on contexts, where T is the action on objects and which is the identity on morphisms. The opposite of the Kleisli category of this monad, restricted to the subcategory generated by the well-formed contexts, is exactly the usual contextual category of our type theory [61]. At least some properties — such as the existence of products and the pullback characterization of $\Uparrow$ — could be inferred from the same properties in the category of hypothetical judgments.

Beyond the Kleisli category of this monad, it may be interesting to consider its Eilenberg-Moore category. Concretely, an object of the Eilenberg-Moore category of T is a hypothetical judgment $\mathcal{G}$ such that $\mathcal{G} \vdash s : A$ implies $\mathcal{G}\ s\ A$ for all s, A. In other words, an object is a model of the CC_ω type system, and a morphism is similarly a morphism of models.

We could have taken this point of view in the current chapter and shown type soundness in a broader context, but ultimately the result would have required additional assumptions.

The Church-Rosser theorem. Our proof of the Church-Rosser theorem for CC_ω essentially follows the proof of Takahashi for confluence of the λ-calculus [119]. The method of using parallel reduction to show confluence goes back to Tait and Martin Löf, but was never published in this form. Takahashi introduced the notion of a

maximal parallel reduction on top of this, which further simplifies the proof of the diamond property for parallel reduction.

This particular proof method implies more general results in the theory of rewrite systems [86, 124], so it is not at all surprising that essentially the same proof can be used to show confluence of CC_ω.

Normalization in System F

5

In this chapter we present proofs of weak and strong normalization of the polymorphically typed λ-calculus (System F) [51, 102]. Weak normalization is formulated for a fine-grained call-by-value variant of System F (called System F_{cbv}) and states that every closed, well-typed term has a normal form reachable via call-by-value reduction. Strong normalization for System F states that every well-typed term terminates under an arbitrary reduction strategy.

While these results are standard and go back to Girard [52], we present simpler and more systematic proofs.

It seems to be folklore that normalization proofs can be seen as a form of syntactic model construction. The model in question is usually called the (unary) "logical relation" interpretation and is useful beyond normalization proofs. We make this connection explicit by introducing a notion of set-model for the types of System F [66, 7]. We then construct our logical relations as a particular model. This kind of abstraction is standard in more expressive calculi, such as System F^{ω} or CC, but is usually left implicit in the simpler case of System F.

Following Dreyer et al. [42] we split the definition of our logical relation into two parts. A value relation classifies well-behaved values of System F_{cbv}, while a term relation lifts the value relation to all terms of System F_{cbv}. With this definition of the logical relation there is a direct correspondence to the call-by-value reduction relation and the proof of weak normalization becomes straightforward.

For strong normalization, we use the same proof idea, but refine the notion of value and the lifting into an expression relation. In particular, the proof of strong normalization does not use Girard's notion of "reducibility candidates" [52].

All constructions work directly with a de Bruijn representation of System F. This requires a slightly more involved definition of instantiation on terms and changes to the underlying calculus of explicit substitutions. Our proofs make heavy use of parallel substitutions, as is standard in normalization proofs, while all substitution lemmas follow mechanically through rewriting in our extended calculus of explicit substitutions.

5.1 Polymorphic Types and Models

The types of System F are given by the following grammar.

$$\mathbb{T}_k \ni A^k, B^k ::= X^k \mid A^k \to B^k \mid \forall\, A^{k+1} \qquad \text{where } X^k \in \mathsf{I}_k$$

We denote types with the letters A, B, C. A type is either a variable X, Y, Z, a function type $A \to B$, or a type quantification $\forall\, A$.

The types of System F are isomorphic to λ-terms and the treatment of substitution and instantiation proceeds as in Chapter 3. We write σ for type substitutions. Type instantiation satisfies the following equations, along with the general laws of the σ-calculus.

$$\begin{aligned} X[\sigma] &= \sigma\, X \\ (A \to B)[\sigma] &= A[\sigma] \to B[\sigma] \\ (\forall\, A)[\sigma] &= \forall\, A[0 \cdot \sigma \circ \uparrow] \end{aligned}$$

In this chapter we build interpretations of polymorphic types. The notion of model we need for this is particularly simple.

Definition 5.1 A **model** of System F types is given by a type D, the domain, along with functions

$$\begin{aligned} \mathbb{A} &: D \to D \to D \\ \mathbb{Q} &: (D \to D) \to D \end{aligned}$$

interpreting function types and type quantification. We **evaluate** a type A in a model $[\![A]\!]_\rho$ by recursion on A.

$$\begin{aligned} [\![_]\!]_{_} &: \mathbb{T}_k \to (\mathsf{I}_k \to D) \to D \\ [\![X]\!]_\rho &:= \rho\, X \\ [\![A \to B]\!]_\rho &:= \mathbb{A}\ [\![A]\!]_\rho\ [\![B]\!]_\rho \\ [\![\forall\, A]\!]_\rho &:= \mathbb{Q}\ (\lambda d.\ [\![A]\!]_{d \cdot \rho}) \end{aligned}$$

The underlying model will always be clear from the context and so we do not make it explicit in the notation for evaluation.

Evaluation is well-behaved with respect to instantiation. This is easy to see in the case of renamings.

Lemma 5.2 We have $[\![A\langle\xi\rangle]\!]_\rho = [\![A]\!]_{\xi\circ\rho}$, where $\xi \circ \rho$ refers to forward composition of functions.

Proof By induction on the type A. The cases for type variables and function types are trivial. In the case of quantification we have

$$\begin{aligned}
[\![(\forall\, A)\langle\xi\rangle]\!]_\rho &= [\![\forall\, A\langle 0\cdot\xi\circ\uparrow\rangle]\!]_\rho \\
&= \mathbb{Q}\ (\lambda d.\ [\![A\langle 0\cdot\xi\circ\uparrow\rangle]\!]_{d\cdot\rho}) \\
&= \mathbb{Q}\ (\lambda d.\ [\![A]\!]_{(0\cdot\xi\circ\uparrow)\circ(d\cdot\rho)}) \\
&= \mathbb{Q}\ (\lambda d.\ [\![A]\!]_{d\cdot\xi\circ\rho}) \\
&= [\![\forall\, A]\!]_{\xi\circ\rho}
\end{aligned}$$

■

Corollary 5.3 (Weakening) $[\![A\langle\uparrow\rangle]\!]_{d\cdot\rho} = [\![A]\!]_\rho$.

Using weakening, and essentially the same proof as for Lemma 5.2 we show that evaluation respects instantiation.

Lemma 5.4 $[\![A[\sigma]]\!]_\rho = [\![A]\!]_{[\![\sigma]\!]_\rho}$, where $[\![\sigma]\!]_\rho := \lambda i.\ [\![\sigma\ i]\!]_\rho$.

Proof By induction on A. The cases for type instantiation and function types are trivial. In the case of quantification we use Corollary 5.3 to show $[\![\sigma\circ\uparrow]\!]_{d\cdot\rho} = [\![\sigma]\!]_\rho$.

$$\begin{aligned}
[\![(\forall\, A)[\sigma]]\!]_\rho &= [\![\forall\, A[0\cdot\sigma\circ\uparrow]]\!]_\rho \\
&= \mathbb{Q}\ (\lambda d.\ [\![A[0\cdot\sigma\circ\uparrow]]\!]_{d\cdot\rho}) \\
&= \mathbb{Q}\ (\lambda d.\ [\![A]\!]_{[\![0\,\cdot\,\sigma\circ\uparrow]\!]_{d\cdot\rho}}) \\
&= \mathbb{Q}\ (\lambda d.\ [\![A]\!]_{d\,\cdot\,[\![\sigma\circ\uparrow]\!]_{d\cdot\rho}}) \\
&= \mathbb{Q}\ (\lambda d.\ [\![A]\!]_{d\,\cdot\,[\![\sigma]\!]_\rho}) \\
&= [\![\forall\, A]\!]_{[\![\sigma]\!]_\rho}
\end{aligned}$$

■

Corollary 5.5 $[\![A[B\cdot\mathsf{id}]]\!]_\rho = [\![A]\!]_{[\![B]\!]_\rho\cdot\rho}$

5.2 Syntax and Semantics of System $\mathsf{F}_{\mathsf{cbv}}$

The terms of System $\mathsf{F}_{\mathsf{cbv}}$ contain both value binders and variables, as well as types and through them type variables. Additionally, it is useful to distinguish between terms and values in the syntax. We represent terms and values of System $\mathsf{F}_{\mathsf{cbv}}$ by

two families $\mathbb{E}, \mathbb{V} : \mathbb{N} \to \mathbb{N} \to \mathcal{U}$, indexed with the sizes of the contexts of types and values. The terms and values of System $\mathsf{F}_{\mathsf{cbv}}$ are given by the following grammar.

$$\begin{aligned}
\mathbb{E}_{m,n} &\ni s^{m,n},\ t^{m,n} ::= s^{m,n}\ t^{m,n} \mid s^{m,n}\ A^m \mid v^{m,n} \\
\mathbb{V}_{m,n} &\ni u^{m,n},\ v^{m,n} ::= x^n \mid \lambda A^m\ s^{m,n+1} \mid \Lambda\, s^{m+1,n} \qquad \text{where } x^n \in \mathsf{I}_n
\end{aligned}$$

We write s, t, b for terms, and u, v for values. A term is either an application $s\,t$, a type instantiation $s\,A$, or a value. A value is either a variable x, y, z, a term abstraction $\lambda A\, s$, or a type abstraction $\Lambda\, s$. Note that both type instantiation and term abstractions contain types, contrary to our treatment of CC_ω in Chapter 4.

We define a simultaneous type and value renaming operation for System $\mathsf{F}_{\mathsf{cbv}}$. As usual, we write ξ, ζ for renamings, which we identify with functions between finite types. The instantiation of a type renaming ξ and a term renaming ζ to a term s is written $s\langle\xi, \zeta\rangle$ and defined as follows.

$$\begin{aligned}
\langle,_\rangle &: \mathbb{E}_{m,n} \to (\mathsf{I}_m \to \mathsf{I}'_m) \to (\mathsf{I}_n \to \mathsf{I}'_n) \to \mathbb{E}_{m',n'} \\
(s\,t)\langle\xi,\zeta\rangle &:= s\langle\xi,\zeta\rangle\ t\langle\xi,\zeta\rangle \\
(s\,A)\langle\xi,\zeta\rangle &:= s\langle\xi,\zeta\rangle\ A\langle\xi\rangle \\
x\langle\xi,\zeta\rangle &:= \zeta\, x \\
(\lambda A\, s)\langle\xi,\zeta\rangle &:= \lambda A\langle\xi\rangle\ s\langle\xi, \Uparrow\zeta\rangle \\
(\Lambda\, s)\langle\xi,\zeta\rangle &:= \Lambda\, s\langle\Uparrow\xi, \zeta\rangle
\end{aligned}$$

The definition of instantiation for renamings is largely unsurprising and follows the same pattern as the definition of instantiation in Chapter 3. Formally, we also define instantiation of renamings for values.

In order to define simultaneous instantiation operations for System $\mathsf{F}_{\mathsf{cbv}}$ we need to be able to extend the domain of a substitution when going under a term ($\Uparrow\tau$) or type binder ($\uparrow\tau$).

$$\begin{aligned}
\Uparrow_ &: (\mathsf{I}_n \to \mathbb{V}_{m,n'}) \to (\mathsf{I}_{n+1} \to \mathbb{V}_{m,n'+1}) \\
\Uparrow\tau &:= 0 \cdot \tau \circ \langle \mathsf{id}, \uparrow\rangle \\
\uparrow_ &: (\mathsf{I}_n \to \mathbb{V}_{m,n'}) \to (\mathsf{I}_n \to \mathbb{V}_{m+1,n'}) \\
\uparrow\tau &:= \tau \circ \langle\uparrow, \mathsf{id}\rangle
\end{aligned}$$

Where we write $\tau \circ \langle\xi,\zeta\rangle$ for the substitution mapping x to $(\tau\, x)\langle\xi,\zeta\rangle$.

Note that when we descend under a term binder with $\Uparrow\tau$ we add an interpretation for the new value binding and skip the additional term binder in the range of τ.

When we descend under a type binder, we do not introduce a new value binding, but we still have to skip the additional type binder in the range of τ. The definitions essentially follow from the scoping discipline by inspecting the underlying types.

With these definitions we define the simultaneous instantiation operations on the terms and values of System $\mathsf{F}_{\mathsf{cbv}}$. Given both a type substitution σ and a value substitution τ we define the operation $s[\sigma,\tau]$ instantiating both type and value variables at the same time. Given a type substitution σ, and value substitutions τ,τ' we also define the composition of value substitutions $\tau\circ(\sigma,\tau)$.

$$
\begin{aligned}
(s\,t)[\sigma,\tau] &= s[\sigma,\tau]\,t[\sigma,\tau] & (\lambda\,A s)[\sigma,\tau] &= \lambda\,A[\sigma]s[\sigma,\Uparrow\tau]\\
(s\,A)[\sigma,\tau] &= s[\sigma,\tau]\,A[\sigma] & (\Lambda\,s)[\sigma,\tau] &= \Lambda\,s[\Uparrow\sigma,\uparrow\tau]\\
x[\sigma,\tau] &= \tau\,x & (\tau\circ(\sigma,\tau'))\,x &:= (\tau\,x)[\sigma,\tau']
\end{aligned}
$$

With these definitions, the expected theorems analogous to the development in Chapter 3 hold.

Fact 5.6 The following equations hold in System $\mathsf{F}_{\mathsf{cbv}}$.

- $s\langle\xi,\zeta\rangle = s[\xi,\zeta]$
- $\Uparrow\tau = 0\cdot\tau\circ(\mathsf{id},\uparrow)$
- $\uparrow\tau = \tau\circ(\uparrow,\mathsf{id})$
- $s[\mathsf{id},\mathsf{id}] = s$
- $s[\sigma,\tau][\sigma',\tau'] = s[\sigma\circ\sigma',\tau\circ(\sigma',\tau')]$
- $(s\cdot\tau)\circ(\sigma,\tau') = s[\sigma,\tau']\cdot\tau\circ(\sigma,\tau')$
- $\mathsf{id}\circ(\sigma,\tau) = \tau$
- $\tau\circ(\mathsf{id},\mathsf{id}) = \tau$
- $\tau\circ(\sigma,\tau')\circ(\sigma',\tau'') = \tau\circ(\sigma\circ\sigma',\tau'\circ(\sigma',\tau''))$

Together with the general equations of the σ_{SP}-calculus, Fact 5.6 provides an effective way of solving substitution lemmas.

We define reduction in System $\mathsf{F}_{\mathsf{cbv}}$ with a big-step evaluation relation $s \Downarrow v$ between terms and values.

Definition 5.7 **Evaluation** of a (closed) term to a value is the relation

$$_\Downarrow_\ :\ \mathbb{E}_{0,0}\to\mathbb{V}_{0,0}\to\mathcal{P}$$

defined inductively by the following rules.

$$
\frac{}{v\Downarrow v}
\qquad
\frac{s\Downarrow\lambda A\,b \qquad t\Downarrow u \qquad b[\mathsf{id},u\cdot\mathsf{id}]\Downarrow v}{s\,t\Downarrow v}
\qquad
\frac{s\Downarrow\Lambda\,b \qquad b[A\cdot\mathsf{id},\mathsf{id}]\Downarrow v}{s\,A\Downarrow v}
$$

Note that we use the scoping discipline to restrict evaluation to closed terms and values. In particular, this excludes the case that a term evaluates to a variable.

We now introduce the typing judgments for System F_{cbv}.

Definition 5.8 The type system of System F_{cbv} consists of a term typing relation and a value typing relation

$$\begin{aligned} _ \vdash _ : _ &\quad : \quad (\mathsf{I}_n \to \mathbb{T}_m) \to \mathbb{E}_{m,n} \to \mathbb{T}_m \to \mathcal{P} \\ _ \vdash^v _ : _ &\quad : \quad (\mathsf{I}_n \to \mathbb{T}_m) \to \mathbb{V}_{m,n} \to \mathbb{T}_m \to \mathcal{P} \end{aligned}$$

The corresponding typing judgments are inductively defined by the following rules.

$$\textsc{App}\ \frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\,t : B} \qquad \textsc{Inst}\ \frac{\Gamma \vdash s : \forall\, A}{\Gamma \vdash s\,B : A[B \cdot \mathsf{id}]} \qquad \textsc{Val}\ \frac{\Gamma \vdash^v v : A}{\Gamma \vdash v : A}$$

$$\textsc{Var}\ \frac{}{\Gamma \vdash^v x : \Gamma\, x} \qquad \textsc{Lam}\ \frac{\Gamma, A \vdash b : B}{\Gamma \vdash^v \lambda A\, b : A \to B} \qquad \textsc{All}\ \frac{\Gamma \circ {\uparrow} \vdash b : A}{\Gamma \vdash^v \Lambda\, b : \forall\, A}$$

In our well-scoped setting, a context is a type substitution. By convention, we write Γ for contexts. Most of the rules in Definition 5.8 are standard, but the rules for type instantiation and abstraction require more explanation. There is no need for a side condition in the rule for type instantiation, since capturing of bound variables is impossible in the de Bruijn representation. In the rule for type abstraction, we descend under a type binder and so we must skip this binder in the range of Γ.

The situation is somewhat reminiscent of the definition of instantiation in System F_{cbv}. Just as in the definition of instantiation, the explicit scoping discipline prevents errors in the formulations of the typing judgement.

Most presentations of System F would also keep track of an additional type kinding context (usually denoted by Δ). However, in System F, this context only ensures that type variables are well-scoped, which is already enforced by our scoping discipline. In effect, the kinding context is implicit in the type variable bound.

5.3 Weak Normalization of System F_{cbv}

Weak normalization states that a closed, well-typed term evaluates to a value.

We show weak normalization by building a model of System F types in which a type is interpreted as a set of closed, weakly normalizing terms. The fundamental property of the model is that any term which is well-typed in the empty context is in

the interpretation of its type.

Definition 5.9 The $\mathcal{WN}$-model of System $\mathrm{F_{cbv}}$ is the model $(D, \mathbb{A}, \mathbb{Q})$ defined as follows.

$$\begin{aligned} D &:= \mathbb{V}_{0,0} \to \mathcal{P} \\ \mathbb{A}\, U\, V &:= \{\lambda A\, s \mid \forall v.\ v \in U \to s[\mathsf{id}, v \cdot \mathsf{id}] \in \mathcal{L}\, V\} \\ \mathbb{Q}\, F &:= \{\Lambda\, s \mid \forall U A.\ s[A \cdot \mathsf{id}, \mathsf{id}] \in \mathcal{L}(F\, U)\} \end{aligned}$$

$$\begin{aligned} \mathcal{L} &: (\mathbb{V}_{0,0} \to \mathcal{P}) \to \mathbb{E}_{0,0} \to \mathcal{P} \\ \mathcal{L}\, U &:= \{s \mid \exists v.\ s \Downarrow v \land v \in U\} \end{aligned}$$

In other words, a type is interpreted as a set of closed values. The function $\mathcal{L}$ lifts a set of closed values into a set of closed terms by evaluation.

We write $\mathcal{V}[A]_\rho$ for the **value relation**, the evaluation of A in the $\mathcal{WN}$-model, and $\mathcal{E}[A]_\rho$ for the **expression relation**, the lifting of the value relation $\mathcal{L}(\mathcal{V}[A]_\rho)$.

We extend the $\mathcal{WN}$-model to contexts, which we interpret as sets of *closing* substitutions.

$$\begin{aligned} \mathcal{C}[_]_ &: (\mathsf{I}_n \to \mathbb{T}_m) \to (\mathsf{I}_m \to D) \to (\mathsf{I}_n \to \mathbb{V}_{0,0}) \to \mathcal{P} \\ \mathcal{C}[\Gamma]_\rho &:= \{\tau \mid \forall i.\ \tau\, i \in \mathcal{V}[\Gamma\, i]_\rho\} \end{aligned}$$

We have the following equations for evaluation in the $\mathcal{WN}$-model.

$$\begin{aligned} s \in \mathcal{E}[A]_\rho &= \exists v.\ s \Downarrow v \land v \in \mathcal{V}[A]_\rho \\ v \in \mathcal{V}[X]_\rho &= v \in \rho\, X \\ (\lambda C\, s) \in \mathcal{V}[A \to B]_\rho &= \forall v.\ v \in \mathcal{V}[A]_\rho \to s[\mathsf{id}, v \cdot \mathsf{id}] \in \mathcal{E}[B]_\rho \\ (\Lambda\, s) \in \mathcal{V}[\forall\, A]_\rho &= \forall U A.\ s[A \cdot \mathsf{id}, \mathsf{id}] \in \mathcal{E}[A]_{U \cdot \rho} \end{aligned}$$

From Chapter 5.1 we have some generic results which apply to the $\mathcal{WN}$-model.

Lemma 5.10 $\mathcal{E}[A\langle\uparrow\rangle]_{U \cdot \rho} = \mathcal{E}[A]_\rho$

Proof By Corollary 5.3. ■

Lemma 5.11 $\mathcal{E}[A[B \cdot \mathsf{id}]]_\rho = \mathcal{E}[A]_{\mathcal{V}[A]_\rho \cdot \rho}$

Proof By Corollary 5.5. ■

The $\mathcal{WN}$-model interprets types as sets of closed terms and values. We extend this interpretation to open terms in a context Γ as follows.

Definition 5.12 The semantic typing relations $_ \vDash^v _ : _$ and $_ \vDash _ : _$ are defined as

$$\Gamma \vDash^v v : A \;:=\; \forall \sigma \tau \rho.\ \tau \in \mathcal{C}[\Gamma]_\rho \to v[\sigma, \tau] \in \mathcal{V}[A]_\rho$$
$$\Gamma \vDash s : A \;:=\; \forall \sigma \tau \rho.\ \tau \in \mathcal{C}[\Gamma]_\rho \to s[\sigma, \tau] \in \mathcal{E}[A]_\rho$$

With these definitions we proceed directly to the proof of the fundamental theorem.

Theorem 5.13 Syntactic typing implies semantic typing.

$$\Gamma \vdash^v v : A \;\to\; \Gamma \vDash^v v : A$$
$$\Gamma \vdash s : A \;\to\; \Gamma \vDash s : A$$

Proof By (mutual) induction on the typing derivations.

- VAR: By assumption on τ.
- VAL: By induction, since we have $v \in \mathcal{E}[A]_\rho \leftrightarrow v \in \mathcal{V}[A]_\rho$ for values v.
- APP: By induction, it suffices to show that $s\,t \in \mathcal{E}[B]_\rho$ whenever $s \in \mathcal{E}[A \to B]_\rho$ and $t \in \mathcal{E}[A]_\rho$. From the definition of the expression relation we have $s \Downarrow u$ where $u \in \mathcal{V}[A \to B]_\rho$, as well as $t \Downarrow v$ where $v \in \mathcal{V}[A]_\rho$. By the definition of the value relation we have $u = \lambda C\, b$ and $b[\mathsf{id}, v \cdot \mathsf{id}] \in \mathcal{E}[B]_\rho$. In particular, we have $b[\mathsf{id}, v \cdot \mathsf{id}] \Downarrow w$ with $w \in \mathcal{V}[B]_\rho$.

  Everything together amounts to $s\,t \Downarrow w$ with $w \in \mathcal{V}[B]_\rho$, which is the definition of $s\,t \in \mathcal{E}[B]_\rho$.

- LAM: We have to show that $(\lambda C\, b[\sigma, \Uparrow\tau]) \in \mathcal{V}[A \to B]_\rho$ whenever $\tau \in \mathcal{C}[\Gamma]_\rho$. By the definition of the value relation and Fact 5.6 it suffices to show that $b[\sigma, v \cdot \tau] \in \mathcal{E}[B]_\rho$ given $v \in \mathcal{V}[A]_\rho$. Together, the assumptions on v, τ imply that $v \cdot \tau \in \mathcal{C}[A \cdot \Gamma]_\rho$, and so the statement follows by induction.

- INST: By induction, it suffices to show that $s\,A \in \mathcal{E}[B[A \cdot \mathsf{id}]]_\rho$ whenever $s \in \mathcal{E}[\forall\, B]_\rho$. From the definition of the value relation we have $s \Downarrow u$ with $u \in \mathcal{V}[\forall\, B]_\rho$. By the definition of the value relation this means that $u = \Lambda\, b$ with $b[A \cdot \mathsf{id}, \mathsf{id}] \in \mathcal{E}[B]_{U \cdot \rho}$ for all U. This in turn implies that $b[A \cdot \mathsf{id}, \mathsf{id}] \Downarrow v$ where $v \in \mathcal{V}[B]_{U \cdot \rho}$ for all U.

  By Lemma 5.11 the conclusion is equivalent to $s\,A \in \mathcal{E}[B]_{\mathcal{V}[A]_\rho \cdot \rho}$. Putting everything together, we have $s\,A \Downarrow v$ with $v \in \mathcal{V}[B]_{\mathcal{V}[A]_\rho \cdot \rho}$ and the statement follows.

- ALL: We have to show that $(\Lambda\, b[\Uparrow\sigma, \uparrow\tau]) \in \mathcal{V}[\forall A]_\rho$ holds whenever $\tau \in \mathcal{C}[\Gamma]_\rho$. By the definition of the value relation together with Fact 5.6 this amounts to showing that $b[B \cdot \sigma, \tau] \in \mathcal{E}[A]_{U \cdot \rho}$ for all U. This follows from the inductive hypothesis, since $\tau \in \mathcal{C}[\Gamma \circ \uparrow]_{U \cdot \rho}$ follows from Lemma 5.10. ■

Corollary 5.14 If $\vdash s : A$, then there is some v such that $s \Downarrow v$.

Proof By Theorem 5.13 we have $s[\mathsf{id}, \mathsf{id}] \in \mathcal{E}[A]_!$, where ! is the unique function from I_0. The statement follows from Fact 5.6 and the definition of the expression relation. ■

5.4 Syntax and Semantics of System F

The syntax of System $\mathsf{F}_{\mathsf{cbv}}$ is restricted to only allow value substitutions. In order to investigate strong normalization we need the full reduction relation of System F, and therefore also the full substitution operation. For convenience, we introduce a new term type for System F. As before, we represent terms of System F by a family $\mathbb{E} : \mathbb{N} \to \mathbb{N} \to \mathcal{U}$, indexed with the size of the contexts of types and terms. The terms of System F are given by the following grammar.

$$\mathbb{E}_{m,n} \ni u^{m,n},\ v^{m,n} ::= s^{m,n}\ t^{m,n} \mid s^{m,n}\ A^m \mid x^n \mid \lambda A^m\ s^{m,n+1} \mid \Lambda\, s^{m+1,n} \qquad \text{where } x^n \in \mathsf{I}_n$$

We write s, t, u, v for terms, x, y, z for term variables.

Instantiation is defined as in Chapter 5.2, and satisfies the same equations (Fact 5.6).

For full System F, we define evaluation with a small-step reduction relation $s \rhd t$, which represents a single step of computation.

Definition 5.15 The **reduction** relation for System F ($s \rhd t$, read "s reduces to t") is inductively defined by the following rules.

$$\frac{}{(\lambda A\, s)\, t \rhd s[\mathsf{id}, t \cdot \mathsf{id}]} \qquad \frac{}{(\Lambda\, s)\, A \rhd s[A \cdot \mathsf{id}, \mathsf{id}]} \qquad \frac{s \rhd s'}{s\, t \rhd s'\, t} \qquad \frac{t \rhd t'}{s\, t \rhd s\, t'}$$

$$\frac{s \rhd s'}{s\, A \rhd s'\, A} \qquad \frac{s \rhd s'}{\lambda A\, s \rhd \lambda A\, s'} \qquad \frac{s \rhd s'}{\Lambda\, s \rhd \Lambda\, s'}$$

As usual, reduction is compatible with instantiation.

Lemma 5.16 (Substitutivity) $s \rhd t$ implies $s[\sigma, \tau] \rhd t[\sigma, \tau]$.

Proof By induction on the derivation of $s \rhd t$, analogous to the proof of Lemma 3.7. ■

In addition, the names of free variables are irrelevant for the reduction relation. We make this precise with a partial converse to Lemma 5.16.

Lemma 5.17 (Anti-Renaming Lemma) If $s\langle\xi,\zeta\rangle \rhd t$ then there exists some s' such that $s \rhd s'$ and $t = s'\langle\xi,\zeta\rangle$.

Proof By induction on the derivation of $s\langle\xi,\zeta\rangle \rhd t$ analogous to the proof of Lemma 5.16. For the congruence rules the proof proceeds by induction, with a substitution lemma in cases where we traverse a binder. In the case of β-reduction we have

$$\begin{aligned}((\lambda A\, s)\, t)\langle\xi,\zeta\rangle &= (\lambda A\langle\xi\rangle\, s\langle\xi,\Uparrow\zeta\rangle)\, t\langle\xi,\zeta\rangle \\ &\rhd s\langle\xi,\Uparrow\zeta\rangle[\mathsf{id}, t\langle\xi,\zeta\rangle\cdot\mathsf{id}] \\ &= s[\xi, t[\xi,\zeta]\cdot\zeta] \\ &= s[\mathsf{id}, t\cdot\mathsf{id}]\langle\xi,\zeta\rangle\end{aligned}$$

The case of type β-reduction is similar. ■

In particular, Lemma 5.16 and Lemma 5.17 imply that $s \rhd t \leftrightarrow s\langle\xi,\zeta\rangle \rhd t\langle\xi,\zeta\rangle$.

Technically, type substitutions do not matter at all for System F reduction and so the anti-renaming lemma could be generalized to reductions of the form $s[\sigma,\zeta] \rhd t$, but this additional generality is not needed in the remaining development.

The type system of System F is entirely analogous to the type system of System $\mathsf{F}_{\mathsf{cbv}}$.

Definition 5.18 The typing system of System F is given by the relation $_ \vdash _ : _$ defined inductively by the following rules.

$$\textsc{App}\ \frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\, t : B} \qquad \textsc{Inst}\ \frac{\Gamma \vdash s : \forall A}{\Gamma \vdash s\, B : A[B\cdot\mathsf{id}]} \qquad \textsc{Var}\ \frac{}{\Gamma \vdash x : \Gamma\, x}$$

$$\textsc{Lam}\ \frac{\Gamma, A \vdash b : B}{\Gamma \vdash \lambda A\, b : A \to B} \qquad \textsc{All}\ \frac{\Gamma \circ \uparrow\, \vdash b : A}{\Gamma \vdash \Lambda\, b : \forall A}$$

Finally, we define strong normalization of terms. Classically, a term s is strongly normalizing, or terminating, if there are no infinite reduction paths starting at s. Constructively, we take the contrapositive statement as the definition. Informally, a term s is strongly normalizing if all reduction paths starting at s are finite. There

is a simple inductive definition of strong normalization in this sense, originally due to Altenkirch [8].

Definition 5.19 The set of strongly normalizing terms $\mathcal{SN}$ is inductively defined by the following rule.

$$\frac{\forall t.\ s \rhd t \to t \in \mathcal{SN}}{s \in \mathcal{SN}}$$

A term s is strongly normalizing if all of its redices, i.e., all terms t such that s reduces to t, are strongly normalizing. Since reduction in System F is finitely branching, a proof of $\mathcal{SN}\ s$ can be visualized as a finite, cycle-free reduction graph connecting s to all of its redices.

5.5 Strong Normalization of System F

The strong normalization property for System F states that every well-typed term is strongly normalizing.

The proof of strong normalization proceeds exactly as in the case of weak normalization. We first build a model for System F types, where a type is interpreted as a set of terms. Unlike in the case of weak normalization, we consider terms in arbitrary contexts, since we ultimately want to show a statement about open terms.

As in the $\mathcal{WN}$-model, we split the model into two parts, a value relation and a term relation. A value, for our purposes, is a weakly head-normal term, excluding variables.

Definition 5.20 A term s is a value if it is of the form $\lambda C\, b$ or $\Lambda\, b$. We write $\mathsf{value} : \mathbb{E} \to \mathcal{P}$ for the corresponding predicate which checks that a term is a value.

The only difference between the $\mathcal{WN}$-model and the $\mathcal{SN}$-model is the way in which we lift the value relation into the term relation. First, since we no longer deal exclusively with closed values, we need to ensure that the expression relation is stable under context extension. In fact, we demand that the expression relation is stable under arbitrary renamings, since this is equally easy to establish. Second, since we have to handle all reduction paths in order to obtain strong normalization, membership in the expression relation requires terms to be well-behaved along arbitrary reduction paths.

Definition 5.21 The $\mathcal{SN}$-model of System F is the model $(D, \mathbb{A}, \mathbb{Q})$ where

$$\begin{aligned} D &:= \forall m\, n.\ \mathbb{V}_{m,n} \to \mathcal{P} \\ \mathbb{A}\, U\, V &:= \{\lambda A\, s \mid \forall v.\ v \in \mathcal{L}\, U \to s[\mathsf{id}, v \cdot \mathsf{id}] \in \mathcal{L}\, V\} \\ \mathbb{Q}\, F &:= \{\Lambda\, s \mid \forall U A.\ s[A \cdot \mathsf{id}, \mathsf{id}] \in \mathcal{L}(F\, U)\} \end{aligned}$$

Where $\mathcal{L}\, U$ is defined inductively by the following rule.

$$\frac{\begin{array}{c}\mathsf{value}\, s \to s \in \overline{U} \\ \forall t.\ s \rhd t \to t \in \mathcal{L}\, U\end{array}}{s \in \mathcal{L}\, U} \qquad\qquad s \in \overline{U} := \forall \xi\, \zeta.\ s\langle \xi, \zeta\rangle \in U$$

We write $\mathcal{V}[A]_\rho$ for the **value relation**, the evaluation of A in the $\mathcal{SN}$-model, and $\mathcal{E}[A]_\rho$ for the **expression relation**, the lifting of the value relation $\mathcal{L}(\mathcal{V}[A]_\rho)$.

We extend the $\mathcal{SN}$-model to contexts, which we interpret as sets of substitutions.

$$\begin{aligned} \mathcal{C}[_]_ &: (\mathsf{I}_n \to \mathbb{T}_m) \to (\mathsf{I}_m \to D) \to (\mathsf{I}_n \to \mathbb{E}_{m',n'}) \to \mathcal{P} \\ \mathcal{C}[\Gamma]_\rho &:= \{\tau \mid \forall i.\ \tau\, i \in \mathcal{E}[\Gamma\, i]_\rho\} \end{aligned}$$

The definition of the $\mathcal{SN}$-model is very similar to the definition of the $\mathcal{WN}$-model (Definition 5.9), except for three changes.

- The first change is in the definition of the semantic function type $\mathbb{A}$, which quantifies over all elements in the expression relation, instead of the value relation. This is required, since we will substitute arbitrary terms for the bound variables during reduction, instead of only substituting values.

- The second change is the definition of the lifting operation $\mathcal{L}$, which is now reminiscent of the inductive definition of strong normalization.

- The third change is in the context relation $\tau \in \mathcal{C}[\Gamma]_\rho$, which now demands that the image of τ is in the expression relation.

Fact 5.22 We have the following properties of $\mathcal{L}$.

(1) $s \in \mathcal{L}\, U$ implies $s \in \mathcal{SN}$.

(2) $s \in \mathcal{L}\, U$ and $s \rhd t$ imply $t \in \mathcal{L}\, U$

(3) $x \in \mathcal{L}\, U$

(4) $s \in \mathcal{L}\, U$ implies $s\langle \xi, \zeta\rangle \in \mathcal{L}\, U$

Proof (1) By induction on the derivation of $s \in \mathcal{L}\, U$.

(2) By case analysis on the derivation of $s \in \mathcal{L}\,U$.

(3) We have $x \not\succ t$ for all t, and by definition a variable is not a value.

(4) By induction on the derivation of $s \in \mathcal{L}\,U$ using Lemma 5.17. ∎

We have the following equations for evaluation in the $\mathcal{SN}$-model, which follow from Fact 5.6 and Fact 5.22 (4).

$$
\begin{aligned}
s \in \overline{\mathcal{V}[X]_\rho} &= \forall \xi\zeta.\ v\langle \xi, \zeta\rangle \in \rho\, X \\
(\lambda C\, s) \in \overline{\mathcal{V}[A \to B]_\rho} &= \forall \xi\zeta v.\ v \in \mathcal{E}[A]_\rho \to s[\xi, v \cdot \zeta] \in \mathcal{E}[B]_\rho \\
(\Lambda\, s) \in \overline{\mathcal{V}[\forall\, A]_\rho} &= \forall \xi U A.\ s[A \cdot \xi, \mathsf{id}] \in \mathcal{E}[A]_{U \cdot \rho}
\end{aligned}
$$

The generic results of Chapter 5.1 apply unchanged to the $\mathcal{SN}$-model.

Lemma 5.23 $\mathcal{E}[A\langle\uparrow\rangle]_{U\cdot\rho} = \mathcal{E}[A]_\rho$

Proof By Corollary 5.3. ∎

Lemma 5.24 $\mathcal{E}[A[B \cdot \mathsf{id}]]_\rho = \mathcal{E}[A]_{\mathcal{V}[A]_\rho \cdot \rho}$

Proof By Corollary 5.5. ∎

As before, we define a semantic typing judgement using the context and expression relations.

Definition 5.25 The semantic typing relation $_ \vDash _ : _$ is defined as follows.

$$
\Gamma \vDash s : A \ := \ \forall \sigma\tau\rho.\ \tau \in \mathcal{C}[\Gamma]_\rho \to s[\sigma, \tau] \in \mathcal{E}[A]_\rho
$$

We could now proceed directly to the proof of the fundamental theorem, but since the individual cases are more involved we organize them into separate lemmas.

Lemma 5.26 We have the following compatibility lemmas for the context relation.

(1) We have $s \cdot \tau \in \mathcal{C}[A \cdot \Gamma]_\rho$ whenever $s \in \mathcal{E}[A]_\rho$ and $\tau \in \mathcal{C}[\Gamma]_\rho$.

(2) We have $\tau \circ (\xi, \zeta) \in \mathcal{C}[\Gamma]_\rho$ whenever $\tau \in \mathcal{C}[\Gamma]_\rho$.

(3) We have $\Uparrow\tau \in \mathcal{C}[A \cdot \Gamma]_\rho$ whenever $\tau \in \mathcal{C}[\Gamma]_\rho$.

Proof (1) By case analysis.

(2) By Fact 5.22 (4).

(3) By (1,2) with Fact 5.22 (3). ∎

Lemma 5.27 We have the following compatibility lemmas for the expression relation.

(1) $s \in \mathcal{E}[A \to B]_\rho$ and $t \in \mathcal{E}[A]_\rho$ imply $s\,t \in \mathcal{E}[B]_\rho$

(2) $s \in \mathcal{SN}$ and $(\lambda C\, s) \in \overline{\mathcal{V}[A \to B]_\rho}$ imply $(\lambda C\, s) \in \mathcal{E}[A \to B]_\rho$

(3) $s \in \mathcal{E}[\forall\, A]_\rho$ implies $s\, C \in \mathcal{E}[A[B \cdot \mathsf{id}]]_\rho$

(4) $s \in \mathcal{SN}$ and $(\Lambda\ s) \in \overline{\mathcal{V}[\forall\, A]_\rho}$ imply $(\Lambda\, s) \in \mathcal{E}[\forall\, A]_\rho$

Proof (1) By simultaneous induction on the derivations of $s \in \mathcal{E}[A \to B]_\rho$ and $t \in \mathcal{E}[A]_\rho$. Since $s\,t$ is not a value it suffices to show that $u \in \mathcal{E}[B]_\rho$ for all u such that $s\,t \rhd u$. There are three cases to consider, either s or t makes a step, in which case the statement follows by the inductive hypothesis, or we perform a β-reduction. In this case we have $s = \lambda C\, b$ and $u = b[\mathsf{id}, t \cdot \mathsf{id}]$.
In particular, since s is a value, we have $s \in \overline{\mathcal{V}[A \to B]_\rho}$ and the statement follows from the definition of $\mathcal{V}[A \to B]_\rho$.

(2) By induction on the derivation of $s \in \mathcal{SN}$. Since the term $(\lambda C\, s)$ is a value, we need to show $(\lambda C\, s) \in \overline{\mathcal{V}[A \to B]_\rho}$, which holds by assumption, as well as $t \in \mathcal{E}[A \to B]_\rho$ for all redices t of $\lambda C\, s$. This follows by induction, since $(\lambda C\, s) \rhd t$ implies that t is of the form $(\lambda C\, s')$ with $s \rhd s'$. The assumption on s is preserved for s' by Fact 5.22 (2).

(3) By induction on the derivation of $s \in \mathcal{E}[\forall\, A]_\rho$. Since $s\, C$ is not a value, it suffices to show $t \in \mathcal{E}[A[B \cdot \mathsf{id}]]_\rho$ for all redices t of $s\, C$.
There are two cases to consider. Either $t = s'\, C$ where $s \rhd s'$ and the statement follows by induction, or $s = \Lambda\, b$ and $t = b[C \cdot \mathsf{id}, \mathsf{id}]$. In the latter case, s is a value, and thus we have $s[C \cdot \mathsf{id}, \mathsf{id}] \in \mathcal{E}[A]_{U \cdot \rho}$ for all U by assumption. With $U = \mathcal{V}[A]_\rho$ the statement then follows from Lemma 5.24.

(4) By induction on the derivation of $s \in \mathcal{SN}$. Since the term $(\Lambda\, s)$ is a value, we have to show $(\Lambda\, s) \in \overline{\mathcal{V}[\forall\, A]_\rho}$, which holds by assumption, as well as $t \in \mathcal{E}[\forall\, A]_\rho$ for all redices t of $(\Lambda\, s)$. However, all redices of $(\Lambda\, s)$ are of the form $(\Lambda\, s')$ where $s \rhd s'$ and the statements follows by induction. The assumption on $s' \in \overline{\mathcal{V}[\forall\, A]_\rho}$ is preserved by Fact 5.22 (2). ∎

With Lemma 5.26 and Lemma 5.27 we obtain the fundamental theorem of logical relations.

Theorem 5.28 Syntactic typing implies semantic typing: $\Gamma \vdash s : A$ implies $\Gamma \vDash s : A$.

Proof By induction on the derivation of $\Gamma \vdash s : A$. The case for VAR follows by assumption, while the cases for APP, INST follow directly from Lemma 5.27 (1) and (3) respectively.

For the remaining cases we reason very similarly to the proof of Theorem 5.13

- Consider the case for LAM. We have to show that $\lambda C[\sigma]\, s[\sigma, \Uparrow\tau] \in \mathcal{E}[A \to B]_\rho$ where $\tau \in \mathcal{C}[\Gamma]_\rho$. By Lemma 5.27 (2) it suffices to show that $s[\sigma, \Uparrow\tau] \in \mathcal{SN}$ and $(\lambda C[\sigma]\ s[\sigma, \Uparrow\tau]) \in \overline{\mathcal{V}[A \to B]_\rho}$.

  For the former, note that we have $\Uparrow\tau \in \mathcal{C}[A \cdot \Gamma]_\rho$ by Lemma 5.26 and thus $s[\sigma, \Uparrow\tau] \in \mathcal{E}[A \to B]_\rho$ by induction. Strong normalization follows from Fact 5.22 (1).

  For the latter, we have to show

  $$s[\sigma, \Uparrow\tau][\xi, t \cdot \zeta] = s[\sigma \circ \xi, t \cdot \tau \circ \zeta] \in \mathcal{E}[B]_\rho$$

  for all $t \in \mathcal{E}[A]_\rho$, which follows by induction using Lemma 5.26 (1) and (2).

- Consider the case for ALL. We have to show that $\Lambda\, s[\Uparrow\sigma, \uparrow\tau] \in \mathcal{E}[\forall\, A]_\rho$ where $\tau \in \mathcal{C}[\Gamma]_\rho$. By Lemma 5.27 (4) it suffices to show that $s[\Uparrow\sigma, \uparrow\tau] \in \mathcal{SN}$ and $\Lambda\, s[\Uparrow\sigma, \uparrow\tau] \in \overline{\mathcal{V}[\forall\, A]_\rho}$.

  For the former, we have $\uparrow\tau = \tau \circ (\uparrow, \mathsf{id}) \in \mathcal{C}[\Gamma \circ \uparrow]_{\perp \cdot \rho} = \mathcal{C}[\Gamma]_\rho$ by Lemma 5.23 and Lemma 5.26 (2). By induction we then have $s[\Uparrow\sigma, \uparrow\tau] \in \mathcal{E}[A]_{\perp \cdot \rho}$ and hence $s[\Uparrow\sigma, \uparrow\tau] \in \mathcal{SN}$ by Fact 5.22 (1).

  For the latter, we have to show that

  $$s[\Uparrow\sigma, \uparrow\tau][C \cdot \xi, \mathsf{id}] = s[C \cdot \sigma \circ \xi, \tau \circ (\xi, \mathsf{id})] \in \mathcal{E}[A]_{U \cdot \rho}$$

  for all C, U, which follows by induction using Lemma 5.26 (2) and Lemma 5.23 to show $\tau \circ (\xi, \mathsf{id}) \in \mathcal{C}[\Gamma \circ \uparrow]_{U \cdot \rho}$. ∎

Corollary 5.29 $\Gamma \vdash s : A$ implies $s \in \mathcal{SN}$.

Proof By Theorem 5.28 we have $s[\mathsf{id}, \mathsf{id}] = s \in \mathcal{E}[A]_\rho$ for an arbitrary choice of ρ, since $\mathsf{id} \in \mathcal{C}[\Gamma]_\rho$ holds vacuously by Fact 5.22 (3). The statement then follows from Fact 5.22 (1). ∎

5.6 Discussion

System F and proofs of strong normalization originate in the literature on proof theory. In proof theory, strong normalization corresponds to the soundness of cut elimination, which can be used to show consistency. The term “logical relation” was coined by Plotkin [95], but the underlying idea goes back to Tait [118]. From a

modern point of view, Tait showed strong normalization of Gödel's System T, which implies the consistency of finite-type Peano arithmetic in a roundabout fashion [53].

In their contemporary form, without the trappings of proof theory, logical relations first appeared in Girard's work [51]. Girard defines only a single expression relation.

To the best of our knowledge, the idea of using several mutually defined relations comes from Pitts and Stark [93], who work with a value, expression, and context relation to handle interactions with an environment. The idea to use the same kind of stratification to show normalization is due to Dreyer et al. [42].

In this chapter we have presented a simple, self-contained normalization proof of System F. Normalization proofs of this form are particularly well-suited to formalization. After the definitions are in place, all of the proofs proceed by applying induction and constructors as needed, with the occasional substitution lemma being solved automatically from the rules of the extended σ_{SP}-calculus. The main contribution of this chapter is in the details of the definitions.

It may not be obvious from the presentation here, but there is a large design space of extensionally equivalent ways of defining the expression relations for System $\mathrm{F}_{\mathsf{cbv}}$ and System F.

For example, Girard [52] uses the following rule in the definition of function types.

$$s \in \mathcal{E}[A \to B]_\rho \;=\; \forall t.\; t \in \mathcal{E}[A]_\rho \to s\,t \in \mathcal{E}[B]_\rho$$

There are several differences to our definition. First, Girard uses a direct recursive definition of the expression relation, rather than a separate value relation. In this case we cannot assume that s is of any specific form, and we have to use the conclusion $s\,t \in \mathcal{E}[B]_\rho$ in the definition above. This in turn requires a "weak-head expansion" lemma to conclude $(\lambda C\, b)\, t \in \mathcal{E}[B]_\rho$ from $b[\mathsf{id}, t \cdot \mathsf{id}] \in \mathcal{E}[B]_\rho$ when verifying the case of the LAM rule in the fundamental theorem.

The same problem occurs in the case of type quantification. Girard uses the following rule for type variables.

$$s \in \mathcal{E}[X]_\rho \;=\; s \in \rho\, X$$

This is problematic, since $\mathcal{E}[A]_\rho$ needs certain closure properties. At the very least, all terms in $\mathcal{E}[A]_\rho$ must be strongly normalizing. Thus, we need to restrict ρ to range over "reducibility candidates", i.e., sets of terms with certain closure properties. This in turn implies that we need to change the definition of $\mathcal{E}[\forall\, A]_\rho$ to quantify over reducibility candidates, and we need to show that $\mathcal{E}[A]_\rho$ is a reducibility candidate for all A.

However, this is *not* the case if we were to apply these definitions to well-scoped (or intrinsically typed) syntax. Part of the definition is that a reducibility candidate contains only strongly normalizing terms. In order to show that $s \in \mathcal{E}[A \to B]_\rho$ is strongly normalizing we need some term $t \in \mathcal{E}[A]_\rho$. Hence we need our reducibility candidates to also be non-empty. As in Fact 5.22, this is usually accomplished by ensuring that all reducibility candidates contain variables — but there are no variables in an empty context!

There are several solutions to this dilemma. We can switch to pure de Bruijn terms and argue that a term terminates whenever the corresponding pure term does, or we can add uninterpreted constants to the language, and argue that doing so does not change the termination behavior of terms. The latter approach is used in [19] and implicitly in [52], where variables also double as typed constants. This is not really problematic in the case of System F, but in more complex systems, such as systems involving dependent types, we also need additional semantic properties of terms, which could be hard to obtain for such an ad-hoc extension.

The real solution to this dilemma is to switch to so called "Kripke-logical relations" [35] as in [4], which are also required to be stable under context extension. We can fix the interpretation of the function type as follows.

$$s \in \mathcal{E}[A \to B]_\rho \;=\; \forall \xi t.\ t \in \mathcal{E}[A]_\rho \to s\langle \mathsf{id}, \xi \rangle\, t \in \mathcal{E}[B]_\rho$$

When we now try to show that s is strongly normalizing, we can obtain $s\langle \mathsf{id}, \uparrow \rangle\, 0 \in \mathcal{E}[B]_\rho$ by switching to a context with an additional free variable. A syntactic inversion lemma, then allows us to conclude $s \in \mathcal{SN}$ from $s\langle \mathsf{id}, \uparrow \rangle\, 0 \in \mathcal{SN}$.

There are three contribution of the construction in this chapter.

- We show that logical relations for System F fit into a very simple notion of model for types. Using this notion of model gives some structural results for free.
- We show that the split into expression and value relations allows us to avoid syntactic inversion lemmas.
- We show that reducibility candidates can be replaced by a closure operator $\mathcal{L}\,U$, which intuitively computes the smallest (Kripke) reducibility candidate containing U.

The lack of inversion lemmas and an explicit notion of reducibility candidates allows us to produce an extremely short formal development.

Moreover, the use of the closure operator $\mathcal{L}$ in the proof of strong normalization generalizes to other settings. For example, it is easy to show that the simply typed λ-calculus with sums [57, Chapter 12] is weakly normalizing, by adapting the proof in Chapter 5.3. We can define the value relation for sums as follows.

$$\mathcal{V}[A + B]_\rho \;=\; \{\, \mathsf{inl}\, s \mid s \in \mathcal{V}[A]_\rho \,\} \cup \{\, \mathsf{inr}\, t \mid s \in \mathcal{V}[B]_\rho \,\}$$

Showing strong normalization is then equally easy by using the closure operator $\mathcal{L}$.

It is much more difficult to extend Girard's proof to sums, since there is no obvious way to define $\mathcal{E}[A + B]_\rho$ directly. The eliminator for sums, and indeed for any positive inductive type, changes the result type and we can no longer define $\mathcal{E}[A + B]_\rho$ by recursion on types.

$$s \in \mathcal{E}[A + B]_\rho \;\stackrel{?}{=}\; \forall C t t'.\, t \in \mathcal{E}[A \to C]_\rho \to t' \in \mathcal{E}[B \to C]_\rho \to \mathsf{case}\, s\, t\, t' \in \mathcal{E}[C]_\rho$$

We can still define $\mathcal{E}[A + B]_\rho$ inductively from $\mathcal{E}[A]_\rho$ and $\mathcal{E}[B]_\rho$ as in [3].

By always using this inductive lifting from a value relation into an expression relation, our approach scales to sums and other type constructors. We demonstrate this in [47], where we present a proof of weak and strong normalization for a Call-by-push-value [77] language with several kinds of sum and product types among other extensions.

Other design variations are possible in the definition of the value relation and $\mathcal{L}$. We have presented $\mathcal{L}$ with a special case for "values", but a notion of value may not exist for all languages. For example, in a "Curry-style" variant of System F without explicit type abstractions and applications we would have to define

$$s \in \mathcal{V}[\forall\, A]_\rho \;=\; \forall U.\; s \in \mathcal{V}[A]_{U \cdot \rho}$$

and this definition needs to apply to all terms, not just to values. The solution is to fold the value condition into the definition of $\mathcal{V}[A]_\rho$ itself and to eliminate the implication from the definition of $\mathcal{L}$.

In the presentation of System F with explicit type application and abstraction it is simpler to use the notion of value, instead of complicating the definition of $\mathcal{V}[A]_\rho$ in every case.

Apart from issues of definition, parallel substitutions are extremely relevant to the study of logical relations. A careful analysis of the substitution lemmas that are needed in the proof of weak and strong normalization reveals that it boils down to

the following two properties of the reduction relation.

$$\begin{aligned} (\lambda C\, s)[\sigma, \tau]\, t &\rhd s[\sigma, t \cdot \tau] \\ (\Lambda\, s)[\sigma, \tau]\, C &\rhd s[C \cdot \sigma, \tau] \end{aligned}$$

These “closure-application” rules are well-known in the literature about implementations of functional programming languages. They are one of the building blocks of virtual machines for lambda terms [72].

6 Tower-based Companions for Coinduction

Up to this point we have only considered inductive definitions, which are well supported in type theory. In the remainder we will also need to use coinductive definitions. Coinduction is formally dual to induction, yet practical support for coinduction in type theory is marginal at best [63].

In this chapter we construct coinductive definitions in type theory and derive powerful proof principles for working with them. We do this at a high level of abstraction, using the language of order theory. In order theory, an inductive definition corresponds to the least fixed point of a monotone function on a complete lattice. Dually, a coinductive definition corresponds to the greatest fixed point of a monotone function on a complete lattice.

The main result in this chapter is an alternative construction of Pous' companion for a monotone function on a complete lattice. Our construction uses type theoretic transfinite recursion — the so called "tower construction" [112]. Using the tower construction we obtain a number of powerful proof principles for the companion, and by extension, for coinductive definitions.

We also take this opportunity to put the tower construction into context. We show that in a classical type theory, the tower construction yields a well-ordered set. This implies a refinement of our proof principles for the companion.

We show that for Scott cocontinuous functions, the companion construction corresponds to the well-known technique of using finite approximations to the greatest fixed point of a function.

The content of this chapter is based on [105] and [112].

6.1 Complete Lattices and Fixed-Points

We recall some standard definitions of order theory, which can be found in any textbook on the subject [26].

Definition 6.1 (Properties of relations) We say that a relation $R : A \to A \to \mathcal{P}$ is

- **Reflexive** if $R\,x\,x$ holds for all x
- **Antisymmetric** if $R\,x\,y$ and $R\,y\,x$ imply $x = y$
- **Transitive** if $R\,x\,y$ and $R\,y\,z$ imply $R\,x\,z$

Definition 6.2 Let A be a type. A **pre-order** on A is a relation $\leq\ : A \to A \to \mathcal{P}$ which is reflexive and transitive. A **partial-order** on A is an antisymmetric pre-order. A **pre-ordered type** is a type A together with a pre-order on A. An **ordered type** is a type A together with a partial-order on A.

Homomorphisms of ordered types are called **monotone functions**. A function $f : A \to B$ between pre-ordered types is monotone if it perserves the order relation, i.e., for all x, y we have $f\,x \leq f\,y$ whenever $x \leq y$.

Ordered types are ubiquitous. We mention just a few examples which are relevant in the remainder.

Example 6.3 Pre- and partially ordered types.

- Implication on propositions is a pre-order. The axiom of propositional extensionality is precisely the statement that implication is antisymmetric.
- Given a family of ordered types F, the dependent function type $\forall (i : I).\ F\,i$ is an ordered type under the pointwise order $f \leq g$ iff $\forall i.\ f\,i \leq g\,i$ by functional extensionality.

Definition 6.4 A **complete lattice** is an ordered type A together with an operation $\bigwedge\ : \forall (I : \mathcal{U}).\ (I \to A) \to A$ (read "meet", or "infimum"), which gives the greatest lower bound of an arbitrary family of elements in A. Formally, given a family $F : I \to A$ of elements in A, and an element $x : A$ we have $x \leq \bigwedge F$ iff x is a lower bound on F.

$$x \leq \bigwedge_{i:I} F\,i \quad \leftrightarrow \quad \forall (i : I).\ x \leq F\,i \tag{6.1}$$

Note that Equation 6.1 specifies the meet operation uniquely, since A is assumed to be an ordered type. In general, given an ordered type A and two elements $x, y : A$ we have $x = y$ iff $x \leq z \Leftrightarrow y \leq z$ for all z.

We will frequently use set notation to denote families over a type. In particular, we make use of the following abbreviations:

$$\{x, y\} := \lambda (b : \mathbb{B}).\ \text{if } b \text{ then } x \text{ else } y$$
$$\{\, f\,x \mid P \,\} := \lambda (a : \Sigma x.P).\ f(\pi\,a)$$

We can use the meet operation in a complete lattice to compute arbitrary least upper bounds as well as greatest lower bounds. The following lemma summarizes some standard constructions in a complete lattice with their associated reasoning principles.

Lemma 6.5 Let A be a complete lattice. We define the following operations on A.

- The least element: $\bot = \bigwedge_{x:A} x$
- Binary meets: $x \wedge y = \bigwedge\{x, y\}$
- Suprema, or joins: $\bigvee F = \bigwedge\{\, x \mid \forall i.\ F\,i \le x \,\}$
- The greatest element $\top = \bigvee_{x:A} x$
- Binary joins $x \vee y = \bigvee\{x, y\}$

The following inference rules are valid in every complete lattice.

$$\frac{}{x \le \top} \qquad \frac{}{\bot \le x}$$

$$\frac{}{x \wedge y \le x} \qquad \frac{}{x \wedge y \le y} \qquad \frac{x \le y \quad x \le z}{x \le y \wedge z}$$

$$\frac{}{x \le x \vee y} \qquad \frac{}{y \le x \vee y} \qquad \frac{x \le z \quad y \le z}{x \vee y \le z}$$

$$\frac{}{\bigwedge F \le F\,i} \qquad \frac{\forall i.\ x \le F\,i}{x \le \bigwedge F} \qquad \frac{}{F\,i \le \bigvee F} \qquad \frac{\forall i.\ F\,i \le x}{\bigvee F \le x}$$

Conversely, the inference rules specify $\top, \bot, \vee, \wedge, \bigvee$ uniquely, since A is in particular an ordered type.

Example 6.6 We give examples of complete lattices.

- $\mathcal{P}$ is a complete lattice. The operation $\bigwedge F$ is universal quantification $\forall i.\ F\,i$. As expected, we can show that $\top$ is True, $\bot$ is False, $\wedge$ is conjunction, $\vee$ is disjunction, and $\bigvee F$ is existential quantification $\exists i.\ F\,i$.
- The product of a family of complete lattices is a complete lattice with the product order. All operations can be lifted pointwise.
- For every complete lattice A there is a complete lattice A^r with the same elements, but for which the order is revered. The operation $\bigwedge F$ in A^r is computed as $\bigvee F$ in A.

A subtype of a complete lattice can itself be made into a complete lattice if it is closed under infima.

Definition 6.7 A predicate $P : A \to \mathcal{P}$ on a complete lattice A is **inf-closed** if for all families F on A, $\bigwedge F \in P$ whenever $F\,i \in P$ for all i.

For inf-closed P, the subtype $\Sigma(x : A).\ P\,x$ is another complete lattice. Infima in $\Sigma(x : A).\ P\,x$ are computed as in A, but suprema are computed according to the formula in Lemma 6.5.

There is another way of defining inf-closed predicates that makes sense in an arbitrary (pre-)ordered type, by representing inf-closed predicates in terms of closure operators.

Definition 6.8 A function $\phi : A \to A$ on an ordered type A is a **closure operator** if

$$x \le \phi\,y \;\leftrightarrow\; \phi\,x \le \phi\,y$$

Separating the two implications, ϕ is a closure operator iff $x \le \phi\,x$ holds and $\phi\,x \le \phi\,y$ holds whenever $x \le \phi\,y$ does. Less concisely, we can say that ϕ is a closure operator iff it is monotone, increasing, and idempotent.

Fact 6.9 Let ϕ be a closure operator on an ordered type. We have

- $x \le \phi\,x$
- ϕ is monotone
- $\phi\,(\phi\,x) = \phi\,x$

There is a direct correspondence between inf-closed predicates and closure operators on a complete lattice.

Lemma 6.10 Let A be a complete lattice, P an inf-closed predicate on A. Then

$$\phi\,x \;:=\; \bigwedge\{\, y \in P \mid x \le y \,\}$$

is a closure operator with image P, i.e., we have

$$\phi\,x = x \;\leftrightarrow\; x \in P$$

Proof We have $x \le \phi\,y$ iff $x \le z$ for all $y \le z \in P$. In particular, we have $x \le \phi\,x$. Conversely, from $x \le \phi\,y$ we can conclude $\phi\,x \le \phi\,y$ since $\phi\,y \in P$ as P is inf-closed.

Since P is inf-closed, we have $\phi\,x \in P$ and hence $\phi\,x = x$ implies $x \in P$. In the reverse direction, if $x \in P$ then $x \le \phi\,x$ by the definition of ϕ and thus $\phi\,x = x$ by Fact 6.9. ∎

Conversely, if ϕ is a closure operator, the set of fixed-points of ϕ is inf-closed. This correspondence extends to an equivalence between the type of inf-closed predicates and the type of closure operators on a complete lattice.

For example, by Example 6.6 the type of relations on a type A is a complete lattice, and it is easy to see that the set of reflexive and transitive relations is inf-closed. The corresponding closure operator is the reflexive, transitive closure of Definition 2.2.

The final result we want to recall concerns the set of fixed points of a monotone function on a complete lattice.

Theorem 6.11 (Knaster-Tarski Theorem [70, 120]) The type of fixed points of a monotone function $f : A \to A$ on a complete lattice A is a complete lattice.

In particular, there is a greatest fixed point νf for which we have

$$\nu f \ = \ \bigvee \{ x \mid x \leq f\, x \}$$

Proof We say that x is a postfixed point of f if $x \leq f\, x$. Since f is monotone, the set of postfixed points of f is closed under suprema. We have $\bigvee F \leq f(\bigvee F)$ whenever $F\, i \leq f\, (F\, i)$ for all i.

Let F be a family of fixed points of f and consider the join of all postfixed points of f below $\bigwedge F$.

$$s \ := \ \bigvee \{ x \mid x \leq f\, x \wedge x \leq \bigwedge F \}$$

We have $s \leq \bigwedge F$ by definition and $s \leq f\, s$ by the discussion above. Since f is monotone, it follows that $f\, s \leq f\, (\bigwedge F) \leq \bigwedge F$ and $f\, s \leq f\, (f\, s)$ and hence $f\, s \leq s$.

Thus s is a fixed point of f, and furthermore greater than any other fixed point of f beneath $\bigwedge F$. This construction turns the set of fixed points of f into a complete lattice. ∎

We give two running examples for the following development.

Example 6.12 Let A be a type and let $\mathcal{S}\, A \ := \ \mathbb{N} \to A$ be the type of sequences (or streams) over A. Consider the function

$$\begin{aligned} \varepsilon \ &: \ (\mathcal{S}\, A \to \mathcal{S}\, A \to \mathcal{P}) \to \mathcal{S}\, A \to \mathcal{S}\, A \to \mathcal{P} \\ \varepsilon \ &:= \ \lambda R\, f\, g.\ f\, 0 = g\, 0 \wedge R\, f'\, g' \end{aligned}$$

where we write f' for the function $f'\, n = f\, (n+1)$.

The function ε is easily seen to be monotone, and by Theorem 6.11 it has a greatest fixed point $\sim := \ \nu\varepsilon$. We have $f \sim g$ whenever there is some relation R such that

$R\,f\,g$ and $R\,h\,i$ implies $h\,0 = i\,0$ and $R\,h'\,i'$ for all h, i. This is the classical definition of bisimilarity for streams.

It is easy to see that in this case $\sim$ coincides with (pointwise) equality of sequences. We have $f \sim f$ for all f using the identity relation and whenever $f \sim g$, we must have $f\,n = g\,n$ for all n by induction.

Example 6.13 A **labeled transition system** (LTS) over an alphabet Σ consists of a type A and a Σ-indexed family of relations $\rhd_\sigma : A \to A \to \mathcal{P}$ over A.

For two labeled transition systems A, B consider the following function.

$$\begin{aligned} \mathrm{s} \;&:\; (A \to B \to \mathcal{P}) \to A \to B \to \mathcal{P} \\ \mathrm{s} \;&:=\; \lambda R\,a\,b.\, \forall \sigma a'.\, a \rhd_\sigma a' \to \exists b'.\, b \rhd_\sigma b' \land R\,a'\,b' \end{aligned}$$

The function s is monotone and thus has a greatest fixed point by Theorem 6.11. We write $a \preceq b$ for $\nu\,\mathrm{s}\;a\,b$ and call the resulting relation **similarity**.

Concretely, we have $a \preceq b$ whenever there is a relation R such that $R\,a\,b$ and for which $R\,a\,b$ implies that for all $a \rhd_\sigma a'$ there is some $b \rhd_\sigma b'$ such that $R\,a'\,b'$. The relation R is called a **simulation**.

6.2 The Tower-Based Companion Construction

Pous [99] gives a characterization of the greatest fixed point as $\nu f = t\,\bot$ using a function t called the companion for f. Parrow and Weber [87] give an ordinal-based construction of the companion in classical set theory. Analogously, it turns out that the companion can be obtained in constructive type theory with an inductive tower construction [112].

Definition 6.14 Let f be a function on a complete lattice. The **tower** for f is the inductive predicate T_f defined by the following rules.

$$\text{T-STEP}\ \frac{x \in T_f}{f\,x \in T_f} \qquad\qquad \text{T-INF}\ \frac{\forall i.\;\; F\,i \in T_f}{\bigwedge F \in T_f}$$

By T-INF, the tower is inf closed and we write t_f for the corresponding closure operator, which we call the **companion** of f. Concretely, t_f is defined as follows.

$$t_f(x) \;:=\; \bigwedge \{\, y \in T_f \mid x \leq y \,\}$$

We will omit the index on T_f and t_f when the function f is clear from the context.

By Fact 6.9 and Lemma 6.10 we have the following properties of t.

Corollary 6.15 t is a closure operator with image T.

(1) t is monotone

(2) $x \leq t\,x$

(3) $t(t\,x) = t\,x$

(4) $x \in T \leftrightarrow t\,x = x$.

These are all consequences of the closure under infima T-INF. Additionally, since the tower is closed under f, we have the following.

Fact 6.16 $f(t\,x) = t(f(t\,x))$

As a consequence of our inductive construction of T, we obtain an induction principle for t.

Theorem 6.17 (Tower Induction) Let P be an inf-closed predicate such that $t\,x \in P$ implies $f(t\,x) \in P$ for all x. Then $t\,x \in P$ holds for all x.

Proof Follows from Corollary 6.15, by induction on the derivation of $t\,x \in T$. ∎

Standard inf-closed predicates include $\lambda x.\ (y \leq x)$ for a fixed y and $\lambda x.\ (g\,x \leq x)$ for monotone g. Both instantiations yield useful facts about t.

Using the predicate $\lambda x.\ (\nu f \leq x)$ in Theorem 6.17, we can reconstruct the greatest fixed point of f in terms of t.

Lemma 6.18 If f is monotone, then $\nu f = t_f\,\bot$.

Proof We have $t\,\bot \leq t(f(t\,\bot)) = f(t\,\bot)$ by monotonicity of t and Fact 6.16. It follows that $t\,\bot \leq \nu f$.

In the reverse direction we show $\nu f \leq t\,x$ for all x using Theorem 6.17. It suffices to show that $\nu f \leq x$ implies that $\nu f \leq f\,x$. This follows from $\nu f = f(\nu f)$ and the monotonicity of f. ∎

More generally, we have $t\,x = \nu f$ for all $x \leq \nu f$, since t is monotone and idempotent.

Using the predicate $\lambda x.\ g\,x \leq x$ in Theorem 6.17, provides a characterization of the up-to functions for t, i.e., the monotone functions below t.

Lemma 6.19 (Up-to Lemma) Let g be monotone. Then the following statements are equivalent.

(1) $g \leq t$

(2) $\forall x.\ g(t\,x) \leq t\,x$

(3) $\forall x.\ g(t\,x) \leq t\,x \rightarrow g(f(t\,x)) \leq f(t\,x)$

Proof The implication from (c) to (b) follows by tower induction. From (b) to (a) we have $g \leq g \circ t \leq t$, by Corollary 6.15 and the monotonicity of g. The implication from (a) to (c) follows from Fact 6.16. ∎

In particular, this shows that f is below t.

Lemma 6.20 Let f be monotone. Then $f(t\,x) \leq t\,x$.

Proof By Lemma 6.19 (b) using the monotonicity of f. ∎

For non-monotone functions g, we can still use Lemma 6.19 to show that $g \leq t$. Since t is monotone, $g \leq t$ is equivalent to $\lceil g \rceil \leq t$, where $\lceil g \rceil$ is the least monotone function above g [63].

$$\lceil g \rceil\, x = \bigvee \{\, g\,y \mid y \leq x \,\}$$

We now relate our companion construction to Pous [99].

Definition 6.21 A function g is **compatible** for f if it is monotone and $g(f\,x) \leq f(g\,x)$ for all x.

Lemma 6.22 For monotone f, we have $t_f = \bigvee \{\, g \mid g \text{ is compatible for } f \,\}$.

Proof Let g be compatible for f. We have $g(f(t\,x)) \leq f(g(t\,x))$ by compatibility, and by Lemma 6.19 this implies $g \leq t$. Additionally, the companion is compatible for f, since it is monotone by Corollary 6.15 and $t(f\,x) \leq t(f(t\,x)) = f(t\,x)$ by Corollary 6.15 and Fact 6.16. ∎

While Pous coined the term "companion" and discovered many of its properties, an equivalent construction had previously been introduced by Hur et al. [63]. Specifically, Hur et al. considered the largest respectful up-to function for a monotone function f.

Definition 6.23 A function g is **respectful** for f if for all $x \leq y$ with $x \leq f\,y$ we have $g\,x \leq f(g\,y)$.

As already noted by Pous, this coincides with the largest compatible up-to function. We give a direct proof of this using Lemma 6.19.

Lemma 6.24 For monotone f, we have $t_f = \bigvee \{\, g \mid g \text{ is respectful for } f \,\}$.

Proof First note that if g is respectful, then so is $\lceil g \rceil$. Assume that $y \leq z$ and $y \leq f\,z$. In order to show that $\lceil g \rceil\, y \leq f(\lceil g \rceil\, z)$ it suffices to show that $g\,x \leq f(\lceil g \rceil\, z)$ for all $x \leq y$. Since g is assumed to be respectful we have $g\,x \leq f(g\,z) \leq f(\lceil g \rceil\, z)$ and thus $\lceil g \rceil$ is respectful.

Thus, without loss of generality, we can assume that g is monotone.

Let g be respectful for f and $g(t\,x) \leq t\,x$. We have $f(t\,x) \leq t\,x$ and $f(t\,x) \leq f(t\,x)$ and so by assumption on g we have $g(f(t\,x)) \leq f(g(t\,x)) \leq f(t\,x)$, hence $g \leq t$ by Lemma 6.19.

In the reverse direction, let $x \leq y$ and $x \leq f\,y$. We have $t\,x \leq t(f\,y) \leq f(t\,y)$ and thus t is compatible. ∎

In light of Lemma 6.22, we can see most results in this section as rederivations of results from [99]. The exceptions are Theorem 6.17 and Lemma 6.19, which are new results for the companion.

Theorem 6.17 and Lemma 6.19 are in some sense equally powerful results. By inspecting the proof, we have already shown that Theorem 6.17 implies Lemma 6.19, but the reverse is also true.

Lemma 6.25 Let t be a function for which $g(t\,x) \leq t\,x$ whenever g is a monotone function such that

$$\forall x.\ g(t\,x) \leq t\,x \rightarrow g(f(t\,x)) \leq f(t\,x).$$

Let P be an inf-closed predicate such that $t\,x \in P$ implies $f\,(t\,x) \in P$. Then we have $t\,x \in P$ for all x.

Proof Let g be the closure operator corresponding to the inf-closed predicate P. By Lemma 6.10 it suffices to show that $g(t\,x) = t\,x$. From Fact 6.9 we have $t(g\,x) \circ t\,x$, thus it suffices to show that $g(t\,x) \leq t\,x$.

By assumption, this is the case whenever $g(f(t\,x)) \leq f(t\,x)$ holds, given $g(t\,x) \leq t\,x$. As before, using Lemma 6.10 and Fact 6.9, this is the case whenever $f(t\,x) \in P$, given $t\,x \in P$, which is exactly our assumption on P. ∎

From a practical point of view, both Theorem 6.17 as well as Lemma 6.19 are useful. In formalizations, Theorem 6.17 is arguably more useful, since there is no syntactic restriction on the shape of the goal. On the other hand, Lemma 6.19 also makes sense in a predicative setting and makes the connections to the classical literature on up-to techniques explicit [103].

In the sequel, we make extensive use of these new results to show soundness of up-to functions.

Example 6.26 (Continued from Example 6.12) We write $f =_R g$ whenever f, g are two streams that are related by the companion for ε at R.

Equality $f = g$ implies $f =_\bot g$ by Lemma 6.18 and thus the relations $f =_R g$ are reflexive for all R.

We could have observed this directly as a consequence of tower induction. In fact, note that the intersection of a family of equivalence relations is an equivalence relation and $\varepsilon\, R$ is an equivalence whenever R is. Hence, by tower induction we know that $=_R$ is an equivalence relation for all R.

The up-to lemma allows us to show that an operation respects $=_R$. For example, consider a function $c : \mathcal{S}\,A \to \mathcal{S}\,A$ on streams. We can use the up-to lemma to show that $c\,f =_R c\,g$ whenever $f =_R g$. To see this, inductively define $C\,R\,(c\,f)\,(c\,g) := R\,f\,g$ and observe that the statement is equivalent to $C(=_R) \subseteq (=_R)$.

By the up-to lemma we can then assume that c respects $=_R$ for a fixed R and we have to show that it does so for a pair of extended streams. Concretely, we can assume that $f =_R g$ and we have to show that $(c\,(a \cdot f))_0 = (c\,(a \cdot g))_0$ and $(c\,(a \cdot f))' =_R (c\,(a \cdot g))'$. In particular, this implies that $(c\,f)_0$ is a function of f_0 and $(c\,(a \cdot f))'$ is a function of a and $c\,f$.

Example 6.27 (Continued from Example 6.13) We write $P \preceq_R Q$ whenever P, Q are related by the companion for s at R.

This example is very similar to the previous one. As before, $P \preceq Q$ implies $P \preceq_R Q$ for all R and $\preceq_R$ is a pre-order for all R by tower induction. The up-to lemma allows us to show that operations respect $\preceq_R$.

For example, to show that an operation c on LTS' respects $\preceq_R$ we can assume that it does so for a fixed R and then we have to show that whenever two nodes x, y are $\preceq_R$ related after one step, $c\,x, c\,y$ are $\preceq_R$ related after one step. We will present several examples of this in Chapter 7.

6.3 Parameterized Tower Induction

We establish a variant of the tower induction principle similar in spirit to Hur et al.'s parameterized coinduction [63].

Lemma 6.28 (Parameterized Tower Induction) Let f be a monotone function on a complete lattice A, u an element of A and P an inf-closed predicate. We have $P(t\,u)$ and $P(f(t\,u))$, whenever

$$\forall x.\ u \leq t\,x \to P(t\,x) \to P(f(t\,x)).$$

Proof The statement $P(f(t\,u))$ follows from $P(t\,u)$ and the assumption together with Fact 6.9.

To show $P(t\,u)$, we generalize the statement to $\forall x.\ Q(t\,x)$ for the inf-closed predicate $Q\,x = u \leq x \to P\,x$. By tower induction, it suffices to show that $P(f(t\,x))$ follows from $u \leq t\,x \to P(t\,x)$ and $u \leq f(t\,x)$. From Lemma 6.20 we know that $u \leq f(t\,x) \leq t\,x$. Thus $P(t\,x)$ holds and $P(f(t\,x))$ follows by assumption. ∎

Hur et al. [63] implement parameterized coinduction with an accumulation rule for parameterized fixed points. Pous shows that the same accumulation rule is applicable to the companion. This also follows immediately from Lemma 6.28 with the predicate $P = \lambda x.\ y \leq x$.

Lemma 6.29 For monotone f we have $x \leq f(t(x \vee y)) \;\leftrightarrow\; x \leq f(t\,y)$.

Proof The right-to-left direction follows from $y \leq x \vee y$ together with the monotonicity of t and f. In the left-to-right direction we use Lemma 6.28. It suffices to show that

$$\forall z.\ y \leq t\,z \to x \leq t\,z \to x \leq f(t\,z).$$

Combining the two assumptions, we have $x \vee y \leq t\,z$. Using Fact 6.9, this is equivalent to $t(x \vee y) \leq t\,z$. The statement follows from $x \leq f(t(x \vee y))$ and the monotonicity of f. ∎

As shown in [63], we can use Lemma 6.29 for both incremental and modular coinductive reasoning. Modularity here is due to the fact that we can read a statement of the form $x \leq f(t\,y)$ as a semantically guarded assumption y. Intuitively, since t is increasing, we can make use of y after at least one unfolding of f.

Together with Lemma 6.18, Lemma 6.29 implies a sound and complete coinduction principle.

Fact 6.30 If f is monotone, then $x \leq f(t\,x) \leftrightarrow x \leq \nu f$.

Proof We have $\nu f = f(\nu f) = f(t\,\bot)$. ∎

Pous observed that every function below the companion is a sound up-to function [103] for f. This is a consequence of Fact 6.30.

Definition 6.31 g is a **sound up-to function** for f, if $x \leq f(g\,x)$ implies $x \leq \nu f$.

Lemma 6.32 If $g \leq t_f$, then g is a sound up-to function for f.

Proof This follows from Fact 6.30: $x \leq f(g\,x) \leq f(t\,x)$. ∎

6.4 Linearity and Well-Foundedness of the Tower Construction

Parrow and Weber [87] give a construction of the companion for bisimilarity in classical set theory. Their construction avoids the quantification over respectful functions by using the theory of ordinals in set theory.

The idea is that if κ is an ordinal larger than the cardinality of the underlying lattice, then $f^\kappa\,\top$ is the greatest fixed point of f. This can be used to construct the companion as follows.

$$t_f\,x \;=\; \bigwedge \{\, f^\alpha\,\top \mid x \le f^\alpha\,\top,\ \alpha \text{ ordinal} \,\}$$

The tower construction [112] may be seen as the analogue of transfinite iteration in type theory. Under this view, we define the set of points reachable from $\top$ by transfinite iteration of f as an inductive predicate.

$$T \;\approx\; \{\, f^\alpha\,\top \mid \alpha \text{ ordinal} \,\}$$

There is one difference between these classical developments and their counterparts in constructive type theory. In the classical setting, T is (co)well-ordered, i.e., linearly ordered and the relation $y \not\le x$ does not have infinite ascending chains on T. This in turn implies some additional properties of the companion.

In order to obtain these classical results in constructive type theory we have to work with stronger (but classically equivalent) assumptions. The road we will take in this thesis is to strengthen the notion of a complete lattice to that of an $\mathcal{L}$-lattice.

Definition 6.33 An $\mathcal{L}$-lattice is a complete lattice A together with a relation $x \sqsubset y$ such that

- $x \sqsubset y \le z$ implies $x \sqsubset z$
- $x \le y \sqsubset z$ implies $x \sqsubset z$
- $\bigwedge F \sqsubset x$ implies $F\,i \sqsubset x$ for some i

A function $f : A \to A$ is $\mathcal{L}$-monotone if it is monotone and we have

$$\forall xy.\ f\,x \sqsubset f\,y \to x \sqsubset y$$

Intuitively, we think of $x \sqsubset y$ as a more informative way of stating that $y \not\le x$. Assuming the axiom of excluded middle, we could define $x \sqsubset y$ as $y \not\le x$ and

conclude that every complete lattice is an $\mathcal{L}$-lattice and that every monotone function is $\mathcal{L}$-monotone. Constructively, the third clause in the definition of an $\mathcal{L}$-lattice does not hold in general, and the two concepts bifurcate.

Both classically and intuitionistically, the basic structure of the linearity proof is best articulated with an induction principle [113].

Lemma 6.34 (Double Induction) For all $R : A \to A \to \mathcal{P}$ such that

- for all $x, y \in T$ with $R\,x\,y$ and $R\,y\,x$ we have $R\,y\,(f\,x)$
- for all $x \in T$, F such that $F\,i \in T$ and $R\,x\,(F\,i)$ for all i we have $R\,x\,(\bigwedge F)$

we have $R\,x\,y$ for all $x, y \in T$.

Proof We show $\forall x \in T.\ R\,x\,y$ by induction on the derivation of $y \in T$.

The case for T-INF holds by assumption.

In the case for T-STEP we have $\forall x \in T.\ R\,x\,y$ and have to show that $R\,x\,(f\,y)$ holds for a fixed $x \in T$. By assumption, this follows from $R\,y\,x$, which we show by a nested induction on the derivation of $x \in T$.

Again, the case for T-INF holds by assumption.

In the case of T-STEP we have $R\,y\,x$ and need to show that $R\,y\,(f\,x)$ holds. By assumption, this follows from $R\,x\,y$, which had been previously established for all $x \in T$. ∎

There is one more general lemma about the tower construction which we need for the following development.

Lemma 6.35 For all $x \in T$ we have $f\,x \le x$.

Proof By induction on the derivation of $x \in T$. In the case of T-STEP we have $f\,x \le x$ and thus $f\,(f\,x) \le f\,x$ by monotonicity of f.

In the case of T-INF we have $f\,(F\,i) \le F\,i$ for all i. By monotonicity we have $f\,(\bigwedge F) \le f\,(F\,i) \le F\,i$ for all i and so $f\,(\bigwedge F) \le \bigwedge F$. ∎

In the remainder of this section we will assume that A is an $\mathcal{L}$-lattice and that f is an $\mathcal{L}$-monotone function.

Using double induction we can show a suitable generalization of linearity.

Lemma 6.36 Let A be an $\mathcal{L}$-lattice and f an $\mathcal{L}$-monotone function. For all $x, y : T$, $y \sqsubset x$ implies $y \leq f\,x$ and $f\,x \sqsubset y$ implies $x \leq y$.

Proof By double induction.

In the case of T-STEP, we can assume the statement for x, y as well as that $x \sqsubset y$ implies $x \leq f\,y$ and $f\,y \sqsubset x$ implies $y \leq x$. Now assuming that $f\,y \sqsubset x$ we have $y \leq x$ and so $f\,y \leq f\,x$ by monotonicity. Assuming $f\,x \sqsubset f\,y$ we have $x \sqsubset y$ by $\mathcal{L}$-monotonicity and thus $x \leq f\,y$.

In the case of T-INF, we assume that the statement holds in the image of a family F. Assuming that $\bigwedge F \sqsubset x$, there is some i such that $F\,i \sqsubset x$ by the definition of an $\mathcal{L}$-lattice. By the inductive hypothesis we have $F\,i \leq f\,x$ and hence $\bigwedge F \leq f\,x$. In the other direction, assuming that $f\,x \sqsubset \bigwedge F$ we have to show that $x \leq F\,i$ for all i. This follows from the inductive hypothesis, since we have $f\,x \sqsubset \bigwedge F \leq F\,i$ and thus $f\,x \sqsubset F\,i$. ∎

Corollary 6.37 For all $x, y \in T$ we have $x \leq y$ whenever $x \sqsubset y$.

Proof We have $x \leq f\,y \leq y$ by Lemma 6.36 and Lemma 6.35. ∎

We now turn to the termination of $\sqsubset$.

Lemma 6.38 The relation $x \sqsubset y$ terminates on T, i.e., the relation

$$x \sqsubset_T y \;:=\; x \sqsubset y \wedge T\,y$$

is strongly normalizing.

Proof By definition, it suffices to show that the relation $\sqsubset_T$ terminates at y for every $x \leq y$ with $x \in T$. We proceed by induction on the derivation of $x \in T$.

In the case of T-STEP we can assume that $f\,x \leq y$ and that the statement holds for all z with $x \leq z$. By the definition of strong normalization it suffices to show the statement for all $y \sqsubset z$. We have $f\,x \leq y \sqsubset z$ and so $x \leq z$ by Lemma 6.36 and the statement follows from the inductive hypothesis.

In the case of T-INF we have $\bigwedge F \leq y$ and can assume that the statement holds for all z such that $F\,i \leq z$. By the definition of strong normalization it suffices to show the statement for all $y \sqsubset z$. We have $\bigwedge F \leq y \sqsubset z$ and so there is some i such that $F\,i \sqsubset z$. By linearity, this implies $F\,i \leq z$ and the statement follows from the inductive hypothesis. ∎

Note that these results can equivalently be phrased in terms of the companion.

Corollary 6.39 We have $t\,x \le t\,y$ whenever $t\,x \sqsubset t\,y$, and the relation

$$x \sqsubset_t y \;:=\; t\,x \sqsubset t\,y$$

is strongly normalizing.

For the classically minded reader we also record the more natural classical rendition of Corollary 6.39.

Corollary 6.40 Assuming the axiom of excluded middle, every non-empty subset of T has a maximal element. Equivalently, for every non-empty P, there is some $y \in P$ such that $t\,x \le t\,y$ for all $x \in P$. In particular, we have $t\,x \le t\,y$ or $t\,y \le t\,x$ for all x, y.

Proof Under excluded middle every complete lattice is an $\mathcal{L}$-lattice by setting

$$x \sqsubset y \;:=\; y \not\le x$$

With this definition Corollary 6.37 reads $y \not\le x \to x \le y$ which is equivalent to $y \le x \vee x \le y$ for all $x, y \in T$, hence in particular for $t\,x, t\,y$.

Now assume that $x \in P \subseteq T$. We show that $\bigvee P \in P$ by induction on the termination of $x \in T$. We have $x \le \bigvee P$ and by linearity either $\bigvee P = x \in P$, or $x < \bigvee P$. By another application of excluded middle this implies that there is some $x < y \in P$ and the statement follows from the inductive hypothesis.

Finally note that $t\,P \subseteq T$ for all sets P and the statement follows. ■

6.5 Companions of Cocontinuous Functions

The Kleene fixed-point theorem [69] allows us to construct the greatest fixed-point of an ω-cocontinuous function by ω-iteration. In this chapter we derive the corresponding result for the companion of a cocontinuous function. Among other things, this allows us to give a more natural description of the companion for stream equality.

Definition 6.41 A family F is **codirected** if for all i, j there is some k such that $F\,i \ge F\,k \le F\,j$.

A monotone function $f : A \to A$ on a complete lattice A is (Scott-) **cocontinuous** if we have

$$\bigwedge_i f(F\,i) \le f(\bigwedge F)$$

for every non-empty, codirected family F.

Note that since f is monotone, we actually have $f(\bigwedge F) = \bigwedge_i f(F\,i)$. This could have been taken as the definition of cocontinuity.

It is well-known that the greatest fixed point of a cocontinuous function is determined by its finite approximations. Formally, this is the dual of Kleene's fixed-point theorem[1].

The same is true for the companion. We first consider the sequence of finite approximations.

Definition 6.42 The finite approximation c_n to νf is defined as the n-fold applications of f to $\top$.

$$
\begin{aligned}
c_n &:= f^n\,\top \\
f^0\,x &:= x \\
f^{n+1}\,x &:= f(f^n\,x)
\end{aligned}
$$

Using this we define the function $k\,x \;:=\; \bigwedge_n\{\, c_n \mid x \le c_n \,\}$.

It is clear that the tower contains all finite approximations.

Lemma 6.43 We have $c_n \in T$ for all n and thus $k\,x \in T$.

Proof By induction on n. We have $\top \in T$ by T-INF and $c_{n+1} = f\,c_n \in T$ by T-STEP and the inductive hypothesis. Finally, we have $k\,x \in T$ by T-INF. ∎

Corollary 6.44 For $m \le n$ we have $c_n \le c_m$.

Proof By Lemma 6.43 and Lemma 6.35. ∎

Cocontinuity ensures that greatest lower bounds of chains of finite approximations are compatible with f.

Lemma 6.45 For f cocontinuous, k is compatible for f. In particular, k is monotone and we have $k(f\,x) \le f(k\,x)$.

Proof Monotonicity follows directly from the definition.

[1] Kleene's theorem is usually formulated with chain-complete posets, but constructively directedness is a more well-behaved notion.

By cocontinuity we have

$$
\begin{aligned}
f(k\,x) &= f\left(\bigwedge_n \{\, c_n \mid x \le c_n \}\right) \\
&= f\left(\bigwedge_{p:\Sigma n.\ x \le c_n} c_{\pi_1 p}\right) \\
&= \bigwedge_{\Sigma n.\ x \le c_n} c_{n+1}
\end{aligned}
$$

since the family $\lambda p.\ c_{\pi_1 p}$ is codirected: Given m, n with $c_m \ge x \le c_n$ we have $x \le c_{\max\, m\, n}$ and $c_m \ge c_{\max\, m\, n} \le c_n$ by Corollary 6.44. Thus it suffices to show that

$$\forall n.\ x \le c_n \to k(f\,x) \le c_{n+1}$$

And this follows since we have $f\,x \le c_{n+1}$ by monotonicity and so $k(f\,x) \le c_{n+1}$ by the definition of k. ■

Lemma 6.46 For f cocontinuous we have $t\,x = k\,x$.

Proof We have $t\,x \le k\,x$ since $k\,x \in T$ by Lemma 6.43 and $x \le k\,x$ by definition. In the reverse direction we have $k\,x \le t\,x$ by Lemma 6.22 since k is compatible by Lemma 6.45. ■

Example 6.47 (Continued from Example 6.26) The function ε is easily seen to be cocontinuous: Let $F : I \to \mathcal{S}\,A \to \mathcal{S}\,A \to \mathcal{P}$ be a non-empty codirected family of relations between streams and let f, g be two streams such that $\varepsilon(F\,i)\,f\,g$ holds for all i. Since F is non-empty, there is some i_0 such that $\varepsilon(F\,i_0)\,f\,g$ holds and thus $f\,0 = g\,0$. Furthermore, we have $\forall i.\ F\,i\,f'\,g'$ by assumption. It follows that

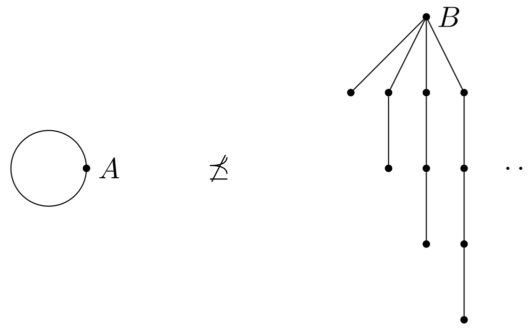
$$\bigwedge_i \varepsilon(F\,i) \le \varepsilon(\bigwedge F)$$

and ε is cocontinuous, since monotonicity was already established.

By Lemma 6.46 we then have $t_\varepsilon\,R\,f\,g = \forall n.\ (R \subseteq c_n) \to c_n\,f\,g$, where c_n is prefix equality $c_n\,f\,g = \forall m.\ m < n \to f\,m = g\,m$.

Thus working with the companion for streams is another way of working with prefix-equality of streams.

Example 6.48 (Continued from Example 6.27) The function s is not cocontinuous, and in fact $\preceq$ cannot be constructed as a limit of finite approximations. Consider the following two LTS over $\Sigma = 1$.



The LTS A is a self loop on the single element type, while B can make a transition to a state n for every natural number n which then admits n further transitions. We have $c_n\, A\, B$ for every n, by relating the single state of A with the state n, but we have $A \not\preceq B$, since if B simulates A by taking a step to a state k we can distinguish the two states after k additional steps.

6.6 The Companion as a Greatest Fixed-Point

In general, we cannot express the companion as a limit of finite approximations. However, the reason for this failure is that the sequence of approximations is not necessarily consistent. In fact, there is a way of expressing $\preceq$ from Example 6.13 as a limit of consistent finite approximations [11]. More generally, we can express all strictly positive coinductive types in this way.

We can apply the same result to the companion, by expressing the companion as a greatest fixed point. This result previously appeared in [99], but we include it to be self-contained.

Lemma 6.49 For a monotone function $f : A \to A$ on a complete lattice A we have

$$t_f \;=\; \nu(\lambda t\, x.\; \bigwedge\{\, f\,(t\, y) \mid x \leq f\, y\,\})$$

where the fixed point is taken over the lattice of functions on A.

Proof Define $c := \nu(\lambda t\, x.\; \bigwedge\{\, f\,(t\, y) \mid x \leq f\, y\,\})$. We show that c is the greatest compatible function for f, which implies that $t_f = c$ by Lemma 6.22.

We first show that c is greater than any comptible function. Let g be monotone with $g(f\, x) \leq f(g\, x)$ for all x. Note that by Tarski's theorem and the definition of c we have

$$(\forall x\, y.\; x \leq f\, y \to g\, x \leq f(g\, y)) \to g \leq c$$

and this follows directly from the compatibility of g, as $g\,x \leq g(f\,y) \leq f(g\,y)$.

Next, we show that c is compatible for f. For all x we have

$$\begin{aligned} c(f\,x) &\leq \bigwedge\{\, f\,(c\,y) \mid f\,x \leq f\,y\,\} \\ &\leq \bigwedge\{\, f\,(c\,y) \mid x \leq y\} \\ &\leq f\,(c\,x) \end{aligned}$$

and so it suffices to show that c is monotone.

But this follows easily from the two properties above. Define the upper monotonization of c as follows.

$$\lceil c \rceil\, x \;:=\; \bigvee\{\, f\,y \mid y \leq x\,\}$$

The function $\lceil c \rceil$ is the greatest monotone function below c. It turns out that $\lceil c \rceil$ is compatible for f and hence $\lceil c \rceil \leq c$ from which we conclude monotonicity of c.

To see this, note that $\lceil c \rceil$ is monotone and we have $\lceil c \rceil(f\,x) \leq f(\lceil c \rceil\, x)$ if $c\,x \leq f(\lceil c \rceil\, y)$ for all $x \leq f y$. By the definition of c it suffices to show that $f(c\,y) \leq f(\lceil c \rceil\, y)$, which follows from the monotonicity of f. ■

The construction in Lemma 6.49 can be seen as defining the companion as the greatest function whose upper monotonization is compatible. Consider a (not necessarily monotone) function $g : A \to A$. The upper monotonization $\lceil g \rceil$ of g is compatible iff for all y:

$$\begin{aligned} &\lceil g \rceil(f\,y) \leq f(\lceil g \rceil\, y) \\ \Longleftrightarrow\; &\forall x.\; x \leq f\,y \to g\,x \leq f(\lceil g \rceil\, y) \end{aligned}$$

In particular since $g \leq \lceil g \rceil$, this holds whenever we have

$$\begin{aligned} &\forall x.\; x \leq f\,y \to g\,x \leq f(g\,y) \\ \Longleftrightarrow\; &\forall x.\; g\,x \leq \bigwedge\{\, f(g\,y) \mid x \leq f\,y\,\} \end{aligned}$$

In a categorical setting it is similarly possible to read Lemma 6.49 as defining the companion $t_f\,x$ as the type of "f-causal functions" from $(\nu\, f)^x$ to νf [101].

Example 6.50 (Continued from Example 6.27) For the function s, we say that R progresses to S [103], written $R \mapsto S$, if $R \subseteq \mathrm{s}\,S$. Concretely, this is the case if whenever $R\,s\,t$ holds and $s \rhd_\sigma s'$ there is some $t \rhd_\sigma t'$ such that $S\,s'\,t'$.

We then have the following concrete description of $\preceq_R$ from Lemma 6.49. The

relation $\preceq_R$ is coinductively defined by the following inference rule.

$$\frac{\forall S \sigma s'.\ (R \mapsto S) \rightarrow s \rhd_\sigma s' \rightarrow \exists t'.\ t \rhd_\sigma t' \wedge s' \preceq_S t'}{s \preceq_R t}$$

It's clear from this that $s \preceq t$ implies $s \preceq_R t$ for all R, by ignoring the extra assumption in the definition of the companion.

In a proof relevant setting it is also possible to replace the condition $R \mapsto S$ directly with the "smallest relation" S such that R progresses to S. This fails for the definition in $\mathcal{P}$, because of the technical elimination restriction on propositions.

6.7 Discussion

Coinductive definitions were introduced by Milner [82] for the analysis of concurrent processes using bisimilarity. The "bisimulation proof method" is exactly using the definition of νf from Tarski's theorem in proofs. This is often tedious in practice and so it is essential to build more refined proof techniques for coinduction.

Proof Techniques for Coinduction Hur et al. [63] introduce parameterized coinduction as an incremental proof technique for coinduction. They cite many prior examples of similar ideas in concrete applications, yet they seem to be the first to put it in the general framework of order theory. Specifically, for a monotone function f on a complete lattice, they construct the function

$$G_f \;=\; \lambda x.\ \nu(\lambda y.\ f(x \vee y)).$$

They show that G_f can be used for modular and incremental coinductive reasoning and demonstrate this with several examples and extensions.

One extension of parameterized coinduction consists of the combination with up-to techniques. Specifically, Hur et al. consider respectful up-to functions. Respectfulness is a sufficient criterion for soundness of up-to functions. They use the fact that the set of respectful up-to functions is closed under union to construct the greatest respectful up-to function t. The parameterized fixed point $G_{t \circ f}$ turns out to obey a nice "unfolding" lemma, which allows them to freely use any respectful up-to technique at arbitrary points in a coinductive proof.

Later on, Pous [99] noticed that the greatest compatible up-to function already admits the parameterized coinduction principle. It turns out that the greatest compatible and the greatest respectful up-to function coincide. Moreover we have $t \circ f = G_{t \circ f}$. This means that the function t is everything we require for incremental and modular

coinductive proofs, which are moreover compatible with up-to techniques.

Pous dubbed the greatest compatible up-to function for f the companion for f.

Our definition of the companion in terms of the tower construction is related to the work of Parrow and Weber [87], who define the companion by transfinite iteration. The main innovation of our work compared to Parrow and Weber is that we work in a constructive setting and in the corollaries we obtain from the tower based companion construction. In particular, (parameterized) tower induction seems to be a genuinely novel proof technique for coinduction.

Beyond this we also provide a characterization of the companion in the case of a cocontinuous function (Chapter 6.5).

Up-To Techniques The study of up-to techniques for bisimilarity, as well as the coinductive definition of bisimilarity, originate with Milner [82]. Milner considers bisimulation up-to bisimilarity to keep proofs of bisimilarity manageable. Practical applications usually require combining several different up-to functions.

One problem with using only sound up-to functions in the sense of Definition 6.31 is that sound up-to functions do not compose in general. This drawback lead Sangiorgi [103] to propose the notion of respectful up-to functions. Respectful up-to functions are sound and closed under composition and union.

Sangiorgi was working with the specific example of bisimilarity, but already noted that the same definition of respectfulness makes sense in the more general context of greatest fixed points in complete lattices.

Pous [100] extends and simplifies the work of Sangiorgi, by abstracting it to the setting of complete lattices and introducing the notion of compatibility. This abstraction immediately yielded concrete gains, as the set of compatible maps forms another complete lattice. In particular, this implies that we can use up-to techniques to show other up-to techniques. Pous refers to this as "second order techniques".

Later on, Pous [99] adapted this development to the companion. For every companion t, there exists a second-order companion, classifying the up-to techniques for t. Pous uses the second-order companion extensively to show soundness of bisimulation up-to context for CCS with replication.

It is not clear if and how the second-order companion is related to tower induction and the up-to Lemma (Lemma 6.19). So far, the two concepts seem to be orthogonal, but with many common applications.

The Tower Construction in Classical Type Theory The classical tower construction in Chapter 6.4 has an interesting history. The tower construction was originally introduced by Zermelo [129], in his second proof of the well-ordering theorem. It is much more common to use the dual construction where the T-INF rule is replaced

by a rule for least upper bounds. In this form, the tower construction sometimes appears in contemporary proofs of Zorn's lemma and related results in algebra and set theory [74].

In classical set theory, the tower construction is usually considered for an increasing function, i.e., a function f with $x \leq f\,x$ for all x, rather than for a monotone function. In this fully general form the tower construction was first identified by Bourbaki [29]. This construction seems to be inherently classical, in that we have not been able to extract any interesting constructive results out of the proofs. Indeed, by a result of Bauer and Lumsdaine [21], the main application of Bourbaki's result fails in the realizability topos and hence does not hold in intuitionistic logic.

The double induction principle (Lemma 6.34) first appeared in the book on set theory by Smullyan and Fitting [113]. In the book, the double induction principle is referred to as the "double superinduction principle" and takes up a prominent position in the development of the theory of ordinals.

The definition of an $\mathcal{L}$-lattice in Chapter 6.4 might seem suspect at first glance, since it is not obvious that such a gadget exists in constructive type theory. Luckily it turns out that there are non-trivial $\mathcal{L}$-lattice's in type theory, for example, the complemented sets of [27], the interval domain, or the extended MacNeille real numbers [125].

In fact, we have extracted the definitions of an $\mathcal{L}$-lattice and $\mathcal{L}$-monotone functions from a development of the well-ordering theorem in classical linear logic according to the methods of [111]. It stands to reason that every complete lattice and monotone function in classical linear logic gives rise to an $\mathcal{L}$-lattice and an $\mathcal{L}$-monotone function under the Chu interpretation.

Strictly speaking, the translation suggests that an $\mathcal{L}$-lattice should satisfy several additional laws, such as $x \sqsubset y \to x \neq y$, but these were not needed in our proofs.

The assumptions in Chapter 6.4 are by no means minimal, and the same construction could be done over a pre-ordered type, rather than a complete lattice. We could have reformulated the tower construction in the form

$$\text{T-INF'}\quad \frac{\forall i.\ F\,i \in T \qquad \forall x.\ (\forall i.\ x \leq F\,i) \leftrightarrow x \leq y}{y \in T}$$

and this would have been equivalent for a complete lattice, while also making sense in an arbitrary pre-ordered type. The additional generality was not useful for us and we would have had to change the definition for Chapter 6.4 to include a similar

condition for $y \sqsubset x$.

While the proof for the linearity of the tower construction is not new, this is the first version I know of that is fully constructive. The proof of Lemma 6.38 is also particularly simple, since we work with the constructively sensible notion of termination, instead of using the existence of a maximal element. It is no accident that the proof of Corollary 6.40 contains three appeals to excluded middle. In a classical proof, these arguments are usually mixed with the proof of well-foundedness of the tower construction, making the structure of the proof more difficult to follow.

The Companion in a Predicative Universe Since we work in a complete lattice, we are implicitly working within an impredicative universe. For cocontinuous functions we can equivalently obtain the companion via ω-iteration (Lemma 6.46). This allows us to define the companion in a predicative, proof relevant setting, but it leaves open the question after the general case.

There are two relevant pieces of related work that we are aware of. First, there is the work by Pous and Rot [101] and Basold et al. [20] who generalize the companion construction from the "proof-irrelevant" setting of order theory to the "proof-relevant" setting of category theory.

The main result is that the categorical analogue of monotonicity, namely functoriality, is no longer sufficient to define the companion. However, in some special cases, there is a "final distributive law of a functor", which clearly corresponds to the greatest compatible function of a monotone function. The work proceeds along deeply classical lines starting from the ordinal based construction of the companion by Parrow and Weber [87]. It is thus not clear how much of this development applies in constructive type theory.

In constructive type theory Danielsson [39] uses size types to construct an analogue of the companion in a predicative universe. In this setting, the companion can be shown to classify "size-preserving" functions, but it appears to be impossible to show that the resulting definition is actually compatible. Nevertheless, the resulting definition allows Danielsson to prove a number of non-trivial results about bisimilarity in CCS, with proofs that are very similar to the proofs given by Pous [99]. So even if the definitions could be separated in some model of type theory, for practical purposes they seem to be equally powerful.

We can only add one preliminary observation to this discussion. In Lemma 6.49 we recall a construction of the companion as a greatest fixed-point. In principle, this definition can be used to define the companion of a fixed function f as a coinductive type. Unfortunately, the definition itself relies on an impredicative quantification and so cannot be applied in a predicative setting. We can, however, eliminate the

impredicative quantification through judicious use of dependent types.

For example, we have shown that the companion for similarity is coinductively defined by the following inference rule.

$$\frac{\forall S \sigma s'.\, (R \mapsto S) \to s \rhd_\sigma s' \to \exists t'.\, t \rhd_\sigma t' \wedge s' \preceq_S t'}{s \preceq_R t}$$

This involves a large quantification over all relations S, which we can replace by reading the relation off from a proof of $R \mapsto S$. For this to work, we formally need to work in a predicative universe $\mathcal{U}$ without an elimination restriction. The resulting definition looks as follows.

$$\frac{\begin{array}{c}\forall \eta : (\forall \sigma s t s'.\, R\, s\, t \to s \rhd_\sigma s' \to \Sigma t'.\, t \rhd_\sigma t').\\ \forall \sigma s'.\, s \rhd_\sigma s' \to \Sigma t'.\, t \rhd_\sigma t' \times s' \preceq_{S\,R\,\eta} t'\end{array}}{s \preceq_R t}$$

$$\text{where } S\, R\, \eta = \lambda s' t'.\, \Sigma \sigma s t (r : R\, s\, t)(p : s \rhd_\sigma s').\, \pi(\eta\, \sigma\, s\, t\, s'\, r\, p) = t'$$

Intuitively, we replace S by the image of the function which corresponds to the proof of $R \mapsto S$. Apart from size issues these two definitions are equivalent, and it might yet be possible to show that this definition corresponds to the final distributive law of a functor as in [101]. A similar construction should apply to any container [2] and might allow us to extend the companion to predicative universes.

Properly investigating any of this is future work.

7 Strong Bisimilarity in CCS

In Chapter 6 we have developed theories for working with coinductive definitions. In this chapter we present a corresponding application.

We consider Milner's calculus of communicating systems (CCS) [82] extended with general recursive processes and show the following results.

- Strong bisimilarity is a congruence (Corollary 7.16).
- Systems of weakly guarded equations have unique solutions (Corollary 7.18).
- Soundness of bisimulation up-to symmetry, transitivity, bisimilarity, and contexts (Corollary 7.19).

These are classic properties of strong bisimilarity in CCS without recursive processes, e.g., the first two appear in [82]. Our contributions lie in the proof strategies and in the extension to recursive processes. Rather than using the bisimulation proof method and setting up three different developments, we will obtain all results from properties of relative bisimilarity — the companion of strong bisimilarity.

Our proof essentially boils down to showing that relative bisimilarity extended to open processes is a congruence. For closed processes, this can be seen as showing the soundness of the bisimulation up-to context proof method [103]. The extension to open processes is not possible with the bisimulation proof method, since there is no reduction relation for open processes. In contrast to this, we can extend relative bisimilarity to open processes by quantifying over closing substitutions. The resulting relation still satisfies the tower induction principle and has all of the pleasant properties of relative bisimilarity on closed terms.

The content of this chapter is based on [105].

7.1 Syntax and Semantics of CCS

We define processes as a family $\mathbb{P} : \mathbb{N} \to \mathcal{U}$ indexed with the size of a process context by the following grammar.

$$P^k, Q^k ::= \bot^k \mid \alpha.P^k \mid P^k \parallel Q^k \mid P^k + Q^k \mid (\nu a)P^k \mid \mu P^{k+1} \mid X^k \quad \text{where } X^k \in \mathsf{I}_k$$
$$\alpha, \beta ::= a \mid \overline{a} \mid \tau \qquad a \in \Sigma$$

$$\frac{}{\alpha.P \overset{\alpha}{\rightsquigarrow} P} \qquad \frac{P \overset{\alpha}{\rightsquigarrow} P'}{P \parallel Q \overset{\alpha}{\rightsquigarrow} P' \parallel Q} \qquad \frac{Q \overset{\alpha}{\rightsquigarrow} Q'}{P \parallel Q \overset{\alpha}{\rightsquigarrow} P \parallel Q'} \qquad \frac{P \overset{a}{\rightsquigarrow} P' \qquad Q \overset{\overline{a}}{\rightsquigarrow} Q'}{P \parallel Q \overset{\tau}{\rightsquigarrow} P' \parallel Q'}$$

$$\frac{P \overset{\overline{a}}{\rightsquigarrow} P' \qquad Q \overset{a}{\rightsquigarrow} Q'}{P \parallel Q \overset{\tau}{\rightsquigarrow} P' \parallel Q'} \qquad \frac{P \overset{\alpha}{\rightsquigarrow} P'}{P + Q \overset{\alpha}{\rightsquigarrow} P'} \qquad \frac{Q \overset{\alpha}{\rightsquigarrow} Q'}{P + Q \overset{\alpha}{\rightsquigarrow} Q'} \qquad \frac{P \overset{\alpha}{\rightsquigarrow} P' \qquad \alpha \neq a, \overline{a}}{(\nu a)P \overset{\alpha}{\rightsquigarrow} (\nu a)P'}$$

$$\frac{P[\mu P \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} Q}{\mu P \overset{\alpha}{\rightsquigarrow} Q}$$

Fig. 7.1.: Labeled transition system for CCS.

A process is either stopped $\bot$, an action prefix $\alpha.P$, a parallel composition $P \parallel Q$, a choice $P + Q$, a restriction $(\nu a)P$, or a recursive process μP. The type Σ is arbitrary and could be kept as a free parameter, but for concreteness we will think of Σ as $\mathbb{N}$. A process P^0 without free variables is called **closed**. We write τ for closing substitutions, i.e., substitutions which replace all variables by closed processes.

The semantics of CCS is given by a labeled transition system (LTS, compare from Example 6.13), i.e., an indexed relation between closed processes $P \overset{\alpha}{\rightsquigarrow} Q$. Intuitively, the relation $P \overset{\alpha}{\rightsquigarrow} Q$ means that process P can reduce to Q and output α in a single step. The labelled transition system for CCS is defined inductively by the rules in Figure 7.1.

We define strong bisimilarity as the greatest fixed-point of a function b on relations between closed processes. First, let s be the function expressing one step of simulation.

$$s\,R := \lambda PQ.\, \forall \alpha P'.\, P \overset{\alpha}{\rightsquigarrow} P' \;\rightarrow\; \exists Q'.\, Q \overset{\alpha}{\rightsquigarrow} Q' \wedge R\,P'\,Q'$$

The function b is simulation in both directions. More precisely, it is the greatest symmetric function below s, where a function between relations is symmetric if it maps symmetric relations to symmetric relations.

$$\begin{aligned} b\,R &:= \lceil s \rceil^s\,R \\ \lceil f \rceil^s &:= \lambda R.\; f\,R \wedge (f\,R^\dagger)^\dagger \\ R^\dagger &:= \lambda PQ.\; R\,Q\,P \end{aligned}$$

Definition 7.1 **Bisimilarity** is the greatest fixed-point of b. **Relative bisimilarity** is

the companion of b. We write

$$P \sim Q := \nu\, b\, P\, Q \qquad \text{bisimilarity}$$
$$P \sim_R Q := t_b\, R\, P\, Q \qquad \text{relative bisimilarity}$$

The definition of bisimilarity is the same as in [82]. To see this, we expand the definitions.

$$\begin{aligned}
\lceil f \rceil^s\, R\, P\, Q &= f\, R\, P\, Q \wedge f\, R^\dagger\, Q\, P \\
b\, R\, P\, Q &= (\forall \alpha P'.\, P \overset{\alpha}{\rightsquigarrow} P' \to \exists Q'.\, Q \overset{\alpha}{\rightsquigarrow} Q' \wedge R\, P'\, Q') \wedge \\
&\quad\ (\forall \alpha Q'.\, Q \overset{\alpha}{\rightsquigarrow} Q' \to \exists P'.\, P \overset{\alpha}{\rightsquigarrow} P' \wedge R\, P'\, Q') \\
P \sim Q &= \exists R.\, R\, P\, Q \wedge R \subseteq b\, R
\end{aligned}$$

where the last equation follows from the characterization of ν in Theorem 6.11.

This characterization of $P \sim Q$ justifies the bisimulation proof method: In order to show that $P \sim Q$ it suffices to find a bisimulation R such that $R\, P\, Q$, where a relation R is a bisimulation if $R \subseteq b\, R$. The problem with this proof technique is that R can be arbitrarily complicated and has to be explicitly constructed.

Consider two processes A, B such that

$$A \sim (\overline{a}.B) \parallel (a.!A)$$
$$B \sim (a.!B) \parallel (\overline{a}.A)$$

where $!P$ refers to the process $\nu(P[\uparrow] \parallel 0)$, describing an infinite parallel composition of P. The processes A and B are obviously bisimilar, as parallel composition is commutative and the only difference beyond this is a renaming. Yet the smallest bisimulation relating A and B is infinite.

Instead of the bisimulation proof method, we use the companion and tower induction. The companion gives us a notion of relative bisimilarity $P \sim_R Q$. Intuitively, processes are bisimilar relative to R if we can show that they are bisimilar, assuming that all R-related processes are bisimilar. In coinductive proofs, we can frequently assume that some processes are bisimilar after we perform a step of reduction. We can express this in terms of relative bisimilarity, by introducing **guarded assumptions** $\circ R$. Given a relation R, we define

$$\circ R := b(t\, R)$$

Tower induction gives us a proof principle for showing bisimilarity in terms of relative bisimilarity.

Lemma 7.2 If $P \sim_R Q$ implies $P \sim_{\circ R} Q$ for all R, then $P \sim Q$.

Proof This is tower induction using the inf-closed predicate $\lambda R.\ R\,P\,Q$. ■

Lemma 7.2 corresponds to the statement that bisimulation up-to the companion is sound. To show that two processes are bisimilar, it suffices to show that they are bisimilar relative to an assumption which states that they are bisimilar after unfolding at least one reduction step.

The main result of this section is that relative bisimilarity is a congruence. This simplifies the proof of the bisimilarity $A \sim B$. By Lemma 7.2, it suffices to show $A \sim_{\circ R} B$, assuming that $A \sim_R B$. We have

$$\begin{aligned} A &\sim (\overline{a}.B) \parallel (a.!A) \sim (a.!A) \parallel (\overline{a}.B) \\ B &\sim (a.!B) \parallel (\overline{a}.A) \end{aligned}$$

Since $\sim \subseteq \sim_{\circ R}$, and $\sim_{\circ R}$ is transitive, it thus suffices to show that

$$(a.!A) \parallel (\overline{a}.B) \sim_{\circ R} (a.!B) \parallel (\overline{a}.A)$$

Since $\sim_{\circ R}$ is a congruence, this follows from $a.!A {\sim_{\circ R}} a.!B$ and $\overline{a}.B {\sim_{\circ R}} \overline{a}.A$. Unfolding the definition of b, we have to show $!A {\sim_R} !B$ and $B \sim_R A$. The former follows by congruence, the latter follows from the symmetry of $\sim_R$. We conclude that $A \sim B$.

In this case, we can further simplify the proof to avoid unfolding the definition of b. We simply strengthen the compatibility with action prefixes to

$$P \sim_R Q \quad \to \quad \alpha.P \sim_{\circ R} \alpha.Q$$

since action prefixes can perform a step of reduction.

Our goal is to show that relative bisimilarity is a congruence. In order to be a congruence under fixed-points we have to extend relative bisimilarity to open terms.

Definition 7.3 Two processes P, Q are in open (relative) bisimilarity $P \mathrel{\dot\sim} Q$ if they are (relatively) bisimilar under all closing substitutions.

$$\begin{aligned} P \mathrel{\dot\sim} Q &:= \forall \tau.\ P[\tau] \sim Q[\tau] && \text{open bisimilarity} \\ P \mathrel{\dot\sim_R} Q &:= \forall \tau.\ P[\tau] \sim_R Q[\tau] && \text{open relative bisimilarity} \end{aligned}$$

To distinguish open bisimilarity from ordinary bisimilarity, we will refer to the latter as closed bisimilarity. Open and closed (relative) bisimilarity coincide for closed processes.

The different notions of bisimilarity are related by the following laws, which instantiate lemmas from Chapter 6.2.

Fact 7.4 $\dot{\sim}_{\perp} = \dot{\sim} \subseteq \dot{\sim}_{\circ R} \subseteq \dot{\sim}_R \supseteq R$

Even though open bisimilarity is not defined coinductively, we obtain a reasoning principle analogous to Lemma 7.2 using tower induction.

Lemma 7.5 If $P \mathrel{\dot{\sim}_R} Q$ implies $P \mathrel{\dot{\sim}_{\circ R}} Q$ for all R, then $P \mathrel{\dot{\sim}} Q$.

Proof This is tower induction using the inf-closed predicate

$$\lambda R.\ \forall \tau.\ R\, P[\tau]\, Q[\tau]$$

■

It is similarly straightforward to show that $\dot{\sim}_R$ is an equivalence relation.

Lemma 7.6 Open relative bisimilarity is an equivalence relation.

Proof By tower induction. The intersection of a family of equivalence relations is an equivalence relation and it is easy to see that $b\,R$ is an equivalence relation if R is. ■

For example, we have the following unfolding law for recursive processes.

Lemma 7.7 $\mu P \mathrel{\dot{\sim}} P[\mu P \cdot \mathsf{id}]$

Proof By Lemma 7.5 it suffices to show that $\mu P \mathrel{\dot{\sim}_{\circ R}} P[\mu P \cdot \mathsf{id}]$. Unfolding the definition using Fact 7.4 and normalizing substitutions this follows from

$$\begin{aligned} & b\,(\sim_R)\,((\mu\,P)[\tau])\,(P[(\mu P)[\tau]\cdot\tau]) \\ = \;& (\forall \alpha Q.\ ((\mu\,P)[\tau] \overset{\alpha}{\rightsquigarrow} Q) \;\rightarrow\; \exists Q'.\ (P[(\mu P)[\tau]\cdot\tau] \overset{\alpha}{\rightsquigarrow} Q') \;\wedge\; Q \sim_R Q') \\ \wedge \;& (\forall \alpha Q.\ (P[(\mu P)[\tau]\cdot\tau] \overset{\alpha}{\rightsquigarrow} Q) \;\rightarrow\; \exists Q'.\ ((\mu\,P)[\tau] \overset{\alpha}{\rightsquigarrow} Q') \;\wedge\; Q \sim_R Q') \end{aligned}$$

There are two directions to consider. When $(\mu P)[\tau] \overset{\alpha}{\rightsquigarrow} Q$ we have $P[(\mu P)[\tau] \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} Q$ by definition and vice-versa. In both cases we have $Q \sim_R Q$ by reflexivity of relative bisimilarity (Lemma 7.6). ■

7.2 Congruence Properties of Relative Bisimilarity

Relative bisimilarity is an equivalence by Lemma 7.6. We show that relative bisimilarity is a congruence by showing that it is compatible with all operations of CCS.

Compatibility can be shown using the up-to lemma (Lemma 6.19), or equivalently using tower induction. In [105] we have presented the development with the up-to lemma. Here we present the same development directly using tower induction.

We prove that $\dot{\sim}_R$ is a congruence for all R. The intersection of a family of congruence relations is again a congruence. By tower induction we can assume that $\dot{\sim}_R$ is a congruence for a fixed R and show that this implies that $\dot{\sim}_{\circ R}$ is a congruence. This in turn boils down to showing that $\dot{\sim}_{\circ R}$ is compatible with all operations individually, which we show with separate lemmas.

Compatibility for the so called "dynamic" operations, which disappear after a step of reduction, is straightforward.

Lemma 7.8 If $P \mathrel{\dot{\sim}_R} Q$, then $\alpha.P \mathrel{\dot{\sim}_{\circ R}} \alpha.Q$ and $\alpha.P \mathrel{\dot{\sim}_R} \alpha.Q$.

Proof If $R\,P\,Q$ then $b\,R\,(\alpha.P)\,(\alpha.Q)$. The statement then follows using Fact 7.4. ■

Lemma 7.9 $\dot{\sim}_{\circ R}$ respects choice, if $P_1 \dot{\sim}_{\circ R} P_2$, and $Q_1 \dot{\sim}_{\circ R} Q_2$ then $P_1 + Q_1 \dot{\sim}_{\circ R} P_2 + Q_2$.

Proof Follows immediately by unfolding the definitions using Fact 7.4. ■

The remaining proofs are more involved and it makes sense to reduce the number of cases by symmetry. Note that all assumptions are symmetric and hence it suffices to consider one step of simulation, instead of simulation in both directions.

Lemma 7.10 If $\dot{\sim}_R$ is a congruence, then $\dot{\sim}_{\circ R}$ respects parallel composition, if $P_1 \mathrel{\dot{\sim}_{\circ R}} P_2$, and $Q_1 \mathrel{\dot{\sim}_{\circ R}} Q_2$ then $P_1 \parallel Q_1 \mathrel{\dot{\sim}_{\circ R}} P_2 \parallel Q_2$

Proof Since all assumptions are symmetric, it suffices to show one direction of the definition of b, i.e., that

$$\begin{aligned} & s\,(\sim_R)\,(P_1[\tau] \parallel Q_1[\tau])\,(P_2[\tau] \parallel Q_2[\tau]) \\ = \;& \forall \alpha U.\;\; (P_1[\tau] \parallel Q_1[\tau] \overset{\alpha}{\rightsquigarrow} U) \;\to\; \exists V.\;\; (P_2[\tau] \parallel Q_2[\tau] \overset{\alpha}{\rightsquigarrow} V) \;\wedge\; U \sim_R V \end{aligned}$$

holds for all closing substitutions τ.

Let $P_1[\tau] \parallel Q_1[\tau] \overset{\alpha}{\rightsquigarrow} U$, we have to find a process V such that $P_2[\tau] \parallel Q_2[\tau] \overset{\alpha}{\rightsquigarrow} V$ and $U \sim_R V$.

We proceed by case analysis on the reduction $P_1[\tau] \parallel Q_1[\tau] \overset{\alpha}{\rightsquigarrow} U$. Formally, there are four cases to consider of which two follow by symmetry.

- Communication between $P_1[\tau]$ and $Q_1[\tau]$: We have $P_1[\tau] \overset{a}{\rightsquigarrow} P_1'$, $Q_1[\tau] \overset{\overline{a}}{\rightsquigarrow} Q_1'$, $\alpha = \tau$, and $U = P_1' \parallel Q_1'$. By the assumptions $P_1 \mathrel{\dot{\sim}_{\circ R}} P_2$ and $Q_1 \mathrel{\dot{\sim}_{\circ R}} Q_2$ there are P_2', Q_2' such that $P_2[\tau] \overset{a}{\rightsquigarrow} P_2'$, $Q_2[\tau] \overset{\overline{a}}{\rightsquigarrow} Q_2'$, $P_1' \sim_R P_2'$ and $Q_1' \sim_R Q_2'$. We pick

$V = P_2' \parallel Q_2'$, as $P_2 \parallel Q_2 \overset{\tau}{\rightsquigarrow} P_2' \parallel Q_2$. The statement $P_1' \parallel Q_1' \sim_R P_2' \parallel Q_2'$ then follows by the assumption that $\dot{\sim}_R$ is a congruence.

- Reduction in $P_1[\tau]$: We have $P_1[\tau] \overset{\alpha}{\rightsquigarrow} P_1'$ and $U = P_1' \parallel Q_1[\tau]$. By assumption, $P_2[\tau] \overset{\alpha}{\rightsquigarrow} P_2'$ and $P_1' \sim_R P_2'$.

  We pick $V = P_2' \parallel Q_2[\tau]$, as $P_2[\tau] \parallel Q_2[\tau] \overset{\alpha}{\rightsquigarrow} P_2' \parallel Q_2[\tau]$. The statement $P_1' \parallel Q_1[\tau] \sim_R P_2' \parallel Q_2[\tau]$ follows from $Q_1[\tau] \sim_R Q_2[\tau]$ since $\sim_R$ is a congruence. This in turn follows from the assumption that $Q_1 \dot{\sim}_{\circ R} Q_2$ and Fact 7.4. ■

Using essentially the same proof we show that relative bisimilarity is compatible with restriction.

Lemma 7.11 If $\dot{\sim}_R$ is a congruence and $P \dot{\sim}_{\circ R} Q$ then $(\nu a)P \dot{\sim}_{\circ R} (\nu a)Q$.

It remains to show compatibility with recursive processes. Since the semantics of recursive processes is defined using instantiation, we also need to show that $\dot{\sim}_R$ is compatible with instantiation. It is easy to see that if $\dot{\sim}_R$ is a congruence, then it is compatible with instantiation.

Lemma 7.12 If $\dot{\sim}_R$ is a congruence and σ_1, σ_2 are two related substitutions, i.e., we have $\sigma_1\, x \dot{\sim}_R \sigma_2\, x$ for all x, then $P[\sigma_1] \dot{\sim}_R P[\sigma_2]$ for all P.

Proof By induction on P using the fact that $\sim_R$ is a congruence. The case for variables follows from the assumption on σ_1 and σ_2. ■

However, this is not quite enough to show compatibility with recursive processes. In order to show that $P \dot{\sim}_{\circ R} Q$ implies $\mu P \dot{\sim}_{\circ R} \mu Q$ we need to show that $\mu P, \mu Q$ are related after unfolding:

$$P[\mu P \cdot \mathsf{id}] \dot{\sim}_{\circ R} Q[\mu Q \cdot \mathsf{id}]$$

The substitutions involved are certainly $\dot{\sim}_R$ related by assumption, but not $\dot{\sim}_{\circ R}$ related. The latter is, after all, what we set out to show.

What saves us is the fact that unfolding fixed-points decreases the size of the reduction derivation and the fact that we already know that $P \dot{\sim}_{\circ R} Q$. This makes it possible to show a more general statement by induction on the derivation of the reduction relation.

Using Lemma 7.12 we can then show that $\dot{\sim}_{\circ R}$ is compatible with recursive processes.

Lemma 7.13 If $\dot{\sim}_R$ is a congruence and $P \dot{\sim}_{\circ R} Q$ then $\mu P \dot{\sim}_{\circ R} \mu Q$.

Proof Unfolding the definitions, it suffices to show that $\mu P \sim_{\circ R} \mu Q$ for closed processes $\mu P, \mu Q$, under the assumption that $P[S \cdot \mathsf{id}] {\sim_{\circ R}} Q[S \cdot \mathsf{id}]$ for all closed processes S.

We show a generalization by induction, namely that for all processes Q_0 we have

$$Q_0[\mu P \cdot \mathsf{id}] \sim_{\circ R} Q_0[\mu Q \cdot \mathsf{id}]$$

the statement then follows when Q_0 is the variable 0.

As in the proof of Lemma 7.10, it suffices to show that the statement holds for one step of simulation. The reverse direction follows by symmetry.

Thus we have to show that

$$\forall \alpha P'.\ \ (Q_0[\mu P \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} P')\ \rightarrow\ \exists Q'.\ \ (Q_0[\mu Q \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} Q')\ \wedge\ P' \sim_R Q'$$

We proceed by induction on the derivation of $Q_0[\mu P \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} Q'$. There are nine cases to consider in total. We illustrate three representative cases.

- $Q_0 = S \parallel T$ and $S[\mu P \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} S'$. By the inductive hypothesis, there is an S'' such that $S[\mu Q \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} S''$ and $S' \sim_R S''$. It suffices to show that

  $$S' \parallel T[\mu P \cdot \mathsf{id}] \sim_R S'' \parallel T[\mu Q \cdot \mathsf{id}].$$

  This follows from compatibility with parallel composition and instantiation (Lemma 7.12), since we have $S' \sim_R S''$ and $\mu P \sim_R \mu Q$ by assumption.

- $Q_0 = \mu S$ and $S[\mu S \cdot \mathsf{id}][\mu P \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} S'$. By the inductive hypothesis, there is an S'' such that $S[\mu S \cdot \mathsf{id}][\mu Q \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} S''$ and $S' \sim_R S''$, which is what we needed to show.

- $Q_0 = X$ and $P[\mu P \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} P'$. By the inductive hypothesis, there is a P'' such that $P[\mu Q \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} P''$ and $P' \sim_R P''$.

  By the assumption on P and Q there is a Q' such that $Q[\mu Q \cdot \mathsf{id}] \overset{\alpha}{\rightsquigarrow} Q'$ and $P'' \sim_R Q'$. We have $P' \sim_R Q'$ by transitivity of $\sim_R$ and the statement follows. ■

Theorem 7.14 Open relative bisimilarity is a congruence.

Proof By tower induction it suffices to assume that $\dot{\sim}_R$ is a congruence and to show that $\dot{\sim}_{\circ R}$ is a congruence. Compatibility with action prefixes, choice, parallel composition, restriction, and recursive processes follow from Lemma 7.8, Lemma 7.9, Lemma 7.10, Lemma 7.11, and Lemma 7.13 respectively. ■

Furthermore, we can use Lemma 7.12 to show that $\dot{\sim}_R$ is compatible with instantiation.

Lemma 7.15 If $P \mathrel{\dot{\sim}_R} Q$ and $\sigma_1\, x \mathrel{\dot{\sim}_R} \sigma_2\, x$ for all x, then $P[\sigma_1] \mathrel{\dot{\sim}_R} Q[\sigma_2]$.

Proof By the definition of $P \mathrel{\dot{\sim}_R} Q$, we have $P[\sigma_1] \mathrel{\dot{\sim}_R} Q[\sigma_1]$. The statement follows from Theorem 7.14, Lemma 7.12, and transitivity (Lemma 7.6). ■

7.3 Properties of Strong Bisimilarity

Since $\dot{\sim}_\perp = \dot{\sim}$, Theorem 7.14 in particular shows that open bisimilarity is a congruence.

Corollary 7.16 Open bisimilarity $\dot{\sim}$ is a congruence.

Proof By Theorem 7.14 and Fact 7.4. ■

We can also directly apply Theorem 7.14 to show that weakly guarded equations have unique solutions in CCS.

A **context** is an open term. A context is **weakly guarded** if every free variable appears under an action prefix, where the action in question may be τ. A context C can be filled using instantiation.

For example, the process

$$\mu(1 \parallel 0)$$

represents the context $![\,]$ for replication.

Weakly guarded equations are bisimilarities of the form $P \mathrel{\dot{\sim}} C[P \cdot \mathsf{id}]$, for weakly guarded contexts C. Systems of weakly guarded equations are the same thing at the level of substitutions.

In the case of CCS without recursive processes, such equations have unique solutions by a result of Milner [82]. Intuitively, this is because we must take a step before reaching a variable, meaning that a weakly guarded equation can be seen as a productive corecursive definition for a process. We can formalize this intuition in terms of relative bisimilarity, which leads to a simple proof.

Lemma 7.17 If C is weakly guarded and $\sigma_1 \sim_R \sigma_2$, then $C[\sigma_1] \sim_{\circ R} C[\sigma_2]$.

Proof By induction on C, using Theorem 7.14 and Lemma 7.15 in particular using Lemma 7.8 to move from guarded relative bisimilarity to relative bisimilarity. ■

Corollary 7.18 (Unique Solutions) If σ is a pointwise weakly guarded substitution and θ_1, θ_2 are two substitutions such that $\theta_1 \dot{\sim} \sigma \circ \theta_1$ and $\theta_2 \dot{\sim} \sigma \circ \theta_2$, then $\theta_1 \dot{\sim} \theta_2$.

In particular, if C is weakly guarded with a single free variable and P, Q are two processes such that $P \dot{\sim} C[P \cdot \mathsf{id}]$ and $Q \dot{\sim} C[Q \cdot \mathsf{id}]$ then $P \dot{\sim} Q$.

Proof By Lemma 7.5, it suffices to show $\theta_1 \dot{\sim}_{\circ R} \theta_2$, assuming that $\theta_1 \dot{\sim}_R \theta_2$. Using Lemma 7.17, we have $(\sigma\, x)[\theta_1] \dot{\sim}_{\circ R} (\sigma\, x)[\theta_2]$ and the statement follows by transitivity, as

$$\theta_1\, x \dot{\sim} (\sigma\, x)[\theta_1] \dot{\sim}_{\circ R} (\sigma\, x)[\theta_2] \dot{\sim} \theta_2\, x.$$ ∎

for all x.

Finally, we show that Theorem 7.14 implies the soundness of bisimilarity up-to context, transitivity, symmetry and bisimilarity. Given a relation R, we build the relation $\mathcal{C}_R$, the closure of R under these operations.

$$\frac{R\,P\,Q}{\mathcal{C}_R\,P\,Q} \qquad \frac{P \sim Q}{\mathcal{C}_R\,P\,Q} \qquad \frac{\mathcal{C}_R\,Q\,P}{\mathcal{C}_R\,P\,Q} \qquad \frac{\mathcal{C}_R\,P\,T \qquad \mathcal{C}_R\,T\,Q}{\mathcal{C}_R\,P\,Q} \qquad \frac{\forall x.\, \mathcal{C}_R\,(\tau_1\,x)\,(\tau_2\,x)}{\mathcal{C}_R\;P[\tau_1]\;P[\tau_2]}$$

Corollary 7.19 The function $\mathcal{C}$ is a sound up-to function for bisimilarity.

Proof For all $\mathcal{C}_R\,P\,Q$ we have $P \sim_R Q$, by induction on the derivation of $\mathcal{C}_R\,P\,Q$ using Theorem 7.14. The statement follows from Lemma 6.32. ∎

7.4 Discussion

This chapter can be seen as the practical counterpart to Chapter 6. In Chapter 6 we introduce the tower-based companion construction and prove many of its properties. In this chapter we apply the companion construction to strong bisimilarity in CCS.

The main advantage of the companion construction over the bisimulation proof method is its modularity. Our proofs that relative bisimilarity is a congruence are very similar to the proof that bisimilarity is a congruence in [82]. The difference is that the result for relative bisimilarity is much stronger. The congruence property of relative bisimilarity has a number corollaries which are otherwise discharged with completely separate proofs.

Up-to techniques for bisimilarity. The study of up-to techniques for bisimilarity originates with Milner [82]. Milner considers bisimulation up-to bisimilarity to keep proofs of bisimilarity manageable. Practical applications usually require combining several different up-to functions. Even the toy example we considered in Chapter 7.1

requires bisimulation up-to context, bisimilarity and transitivity to mimic the proof using the companion.

In [99], Pous presents a proof of soundness of bisimulation up-to context for CCS with replication. The main difference between Pous' development and the development in this chapter is that we consider CCS with recursive processes. Technically, the proofs are very different: Pous uses point-free reasoning with a second-order companion construction where we use tower induction and the notion of relative bisimilarity.

The extension to open processes is not without difficulty, compared to the development of CCS with replication. We have previously compared both in [105]. The companion construction allows a direct treatment of bisimilarity for open processes.

The soundness of bisimulation up-to contexts for CCS with recursive processes appears to be a new result.

8 Axiomatic Semantics for Compiler Verification

In this chapter we present an approach to compiler verification for two idealized imperative languages based on axiomatic semantics.

We formulate the semantics of a language with an "evaluation functional", a monotone function between predicate transformers. The least fixed-point of evaluation provides the weakest precondition transformer wp, while the greatest fixed-point provides weakest liberal preconditions wlp.

Intuitively, the weakest precondition $\mathrm{wp}\ s\,Q$ for a program s and a predicate Q is a predicate which holds of a state σ iff the execution of s starting in state σ terminates with a state satisfying Q. Similarly, the weakest liberal precondition $\mathrm{wlp}\ s\,Q$ is a predicate which holds at σ iff Q holds for all final states of s starting from state σ.

From the weakest liberal preconditions we obtain partial correctness judgments $\{\sigma\}\,s\,\{Q\}$ and Hoare-triples $\{P\}\,s\,\{Q\}$

$$
\begin{aligned}
\{\sigma\}\,s\,\{Q\} &:= \sigma \in \mathrm{wlp}\ s\,Q \\
\{P\}\,s\,\{Q\} &:= P \subseteq \mathrm{wlp}\ s\,Q
\end{aligned}
$$

Hoare triples relate a program with a specification consisting of a pre- and a postcondition. We show that correctness judgments admit a simple inductive characterization.

Symmetrically, weakest preconditions provide a predicate for total correctness judgments $\langle\sigma\rangle\,s\,\langle Q\rangle$. Total correctness judgments give rise to Hoare-style total correctness statements.

$$
\begin{aligned}
\langle\sigma\rangle\,s\,\langle Q\rangle &:= \sigma \in \mathrm{wp}\ s\,Q \\
\langle P\rangle\,s\,\langle Q\rangle &:= P \subseteq \mathrm{wp}\ s\,Q
\end{aligned}
$$

We show that total correctness judgments admit a simple inductive specification.

For the languages we consider, we show that the wp and wlp functions can be computed by structural recursion on programs using a fixed-point operator for predicate transformers.

The first language we study is a Dijkstra-style guarded command language [41] which we call GC. GC is operationally underspecified in that it leaves open which guarded command is executed in case several guards are satisfied. This form of

underspecification is easily dealt with by an axiomatic semantics, which talks about predicates on states rather than single states. The rules of our correctness judgments for GC are reminiscent of the rules of a conventional big-step semantics, with the essential difference that we use postconditions rather than final states. This way the relevant behavior of an underspecified program can be described with a single judgment, which is not possible with a big-step semantics. While we do not define our axiomatic semantics in terms of invariants or termination functions, we show that such a presentation is admissible.

The second language we study is a low-level language with linear sequential composition and lexically scoped gotos which we call IC. We see the declaration of a target for a goto as the definition of an argumentless, possibly recursive procedure to be used as a continuation.

We define the semantics of IC with an operational small-step semantics, and show it equivalent to an axiomatic semantics. The axiomatic semantics of IC facilitates compiler verification. Both the weakest preconditions and the correctness judgments of IC turn out to be useful for proofs establishing properties of IC.

We give a compiler from GC to IC and verify its correctness. The compiler linearizes sequential compositions and realizes loops with continuations. Based on the axiomatic semantics of GC and IC, we express the correctness of the compiler as preservation of specifications. That is, given a GC program s, the compiler must yield an IC program t such that $\langle \sigma \rangle\, t\, \langle Q \rangle$ whenever $\langle \sigma \rangle\, s\, \langle Q \rangle$ (or, equivalently, $\langle P \rangle\, t\, \langle Q \rangle$ whenever $\langle P \rangle\, s\, \langle Q \rangle$), and similarly for the partial correctness judgments.

For the languages we consider, we show that the predicate transformers described by programs are continuous. This makes it possible to obtain the fixed-points for loops and continuations with ω-iteration.

Axiomatic semantics are well adapted to languages with underspecified execution order (e.g., C or OCaml). Treating compiler correctness for such languages with a nondeterministic small-step semantics seems complex and tedious. At the example of GC we see that compiler correctness can be expressed elegantly and without overspecification using axiomatic semantics.

We have covered axiomatic semantics for compiler verification from GC to IC in [107]. The main contribution of this chapter over this previous study is the introduction of evaluation functionals and partial correctness.

Partial correctness is most naturally specified coinductively and so we require both inductive and coinductive proof techniques to reason about axiomatic semantics. The tower construction plays a crucial role in providing a powerful induction and coinduction principle. Since we are not interested in up-to techniques we can further

refine the tower construction based on the proof of Pataraia's theorem [88]. This leads us to identify the (co-)directed (co-)induction principle (Chapter 8.1), which is the main workhorse for most proofs in this chapter.

Before we present the axiomatic semantics of GC and IC we first consider axiomatic semantics based on an evaluation functional in an abstract setting (Chapter 8.2). The main insight is that there are a number of properties of the wp and wlp semantics which can be shown simultaneously for partial and total correctness based on the evaluation functional.

We next present the semantics of GC (Chapter 8.3) and IC (Chapter 8.4). IC has both an operational and an axiomatic semantics (Chapter 8.5) and we show how to connect the two (Chapter 8.6). Finally, we show the correctness of a compiler from GC to IC (Chapter 8.7).

In this section, we often use set notation such as $\bigcap M$ instead of the lattice notation $\bigwedge M$ from Chapter 6. The reason is that many of the results in this section do not hold for all complete lattices, but instead are specific to lattices of predicate transformers[1].

The content of this chapter is based on [107].

8.1 (Co-)Directed (Co-)Induction

In the remainder we will need a stronger form of the tower induction principle from Chapter 6, for induction as well as coinduction.

Tower induction justifies the following coinduction principle.

Lemma 8.1 Let f be a monotone function on a complete lattice X and let P be an inf-closed predicate on X. Then $P(\nu f)$ holds whenever $P\,x$ implies $P(f\,x)$ for all x.

Proof We have $\nu\, f = t_f\,\bot$ by Lemma 6.18 and the result follows by Theorem 6.17. ■

We need the dual result for induction, i.e., for least fixed-points, which naturally requires P to be a sup-closed predicate. However, sup-closure turns out to be too restrictive for some applications. For example, consider a monotone function $f : (X \to X \to \mathcal{P}) \to (X \to X \to \mathcal{P})$ between relations. Suppose that f preserves functional relations, i.e., that $f\,R$ is functional whenever R is. The intersection of a family of functional relations is functional, thus we can conclude that νf is functional. The union of a family of functional relations on the other hand may not be functional, so we cannot conclude that μf is functional.

Using classical logic, we know that the tower is linear (Corollary 6.37) and hence we would only need P to be closed under suprema of chains. The union of a chain of

[1] More generally, the results in question, e.g., Lemma 8.17, hold in all complete Heyting algebras.

functional relations is again functional, so classically we could conclude that μf is functional.

For general complete lattices we adapt a theorem of Pataraia [88] to obtain a (co-)directed (co-)induction principle, which only requires P to be closed under suprema (infima) of (co-)directed families. Observe that the union of a directed family of functional relations is functional. Continuing our example, we conclude that μf is functional.

We begin by recalling the definitions of (co-)directed families and (co-)directed closed predicates.

Definition 8.2 A family $F : I \to X$ over a pre-ordered type X is **directed** if for all $i, j : I$ there exists some $k : I$ such that $F\,i \leq F\,k \geq F\,j$. Dually, F is **codirected** if for all i, j there exists some k such that $F\,i \geq F\,k \leq F\,j$.

A predicate $P : X \to \mathcal{P}$ on a complete lattice X is **directed-closed** if for all directed families F we have $P(\bigvee F)$ whenever $P(F\,i)$ for all i. Dually, P is **codirected-closed** if for all codirected families F, the statement $P(\bigwedge F)$ holds whenever $P(F\,i)$ holds for all i.

The remaining development is symmetric with regards to directed or codirected-closed predicates. For easier comparison to Chapter 6 we describe the case for codirected-closed predicates and obtain the case for directed-closed predicates by duality.

Definition 8.3 The **codirected tower** T_f^* for f is the smallest codirected-closed predicate which is closed under f. Formally, the predicate T_f^* is inductively defined by the following rules.

$$\frac{x \in T_f^*}{f\,x \in T_f^*} \qquad\qquad \frac{\forall i.\ F\,i \in T_f^* \qquad F \text{ codirected}}{\bigwedge F \in T_f^*}$$

The codirected tower is clearly a refinement of the ordinary tower construction. Since f is clear from the context we drop the suffix and write T^* in the remainder.

Our goal is to show that the greatest fixed-point of f is contained in T^*. From this result we can obtain the codirected coinduction principle by induction on the derivation of $\nu f \in T^*$.

The following two facts carry over from the corresponding statements for the tower construction.

Fact 8.4 $\top \in T^*$

Proof We have $\top = \bigwedge \emptyset$, where $\emptyset$ is the unique family over the empty type, which is obviously codirected. ■

Fact 8.5 If f is monotone, then every $x \in T^*$ is a prefixed-point of f, i.e., $f\,x \leq x$.

Proof By induction on the derivation of $x \in T^*$. ■

While it is easy to show that $\nu\, f$ is the least element of the ordinary tower construction (Lemma 6.18), there does not seem to be any straightforward way to show that $\nu\, f$ is in T^*. In fact, there is a priori no reason to assume that the codirected tower has a least element: T^* has a least element iff it is codirected. Attempting to prove this directly runs into the same issues as attempting to prove that the tower is linear.

Intuitively, the points of the tower are of the form $f^\alpha\,\top$. Given just two points $f^\alpha\,\top$ and $f^\beta\,\top$ it is difficult to compute a lower bound in T^*, unless we already know that they are comparable. Pataraia's idea [88] is to look at the functions f^α instead of at the points themselves. Given two functions f^α and f^β it is trivial to compute a lower bound on them: since the functions in question are decreasing we can just take their composition $f^\alpha \circ f^\beta$.

We formalize this by identifying functions of the form f^α as certain functions which are *contractive on* T^*. The family of contractive functions is codirected and closed under infima of codirected families and hence has a least element, which we then use to compute the least element of T^*. By the same argument as in Lemma 6.18, we conclude that the least element of T^* is the greatest fixed-point of f.

Definition 8.6 A function $g : X \to X$ is **contractive on** T^*_f, or **contractive**, if it is

- monotone,
- decreasing on T^*, i.e., $g\,x \leq x$ for $x \in T^*$, and
- preserves T^*, i.e., $g\,x \in T^*$ if $x \in T^*$.

Fact 8.7 The identity function is contractive and the composition of two contractive functions is contractive.

Proof The identity function is clearly contractive by expanding the definition. For two contractive functions g, h we have that $g \circ h$ is monotone and decreasing on T^* since for $x \in T^*$ we have

$$h(g\,x) \leq g\,x \leq x$$

as g preserves T^* and h, g are decreasing on T^*. Similarly, $g \circ h$ preserves T^* since h, g do so individually. ■

Fact 8.8 f is contractive whenever it is monotone.

Proof By definition, f preserves T^*. The function f is monotone by assumption and thus also decreasing on T^* by Fact 8.5. ■

Finally, the family of contractive functions is codirected in the following sense.

Lemma 8.9 Let $x \in T^*$, then the family

$$M_x := \{ g\,x \mid g \text{ contractive} \}$$

is codirected.

Proof Let g, h be two contractive functions. By Fact 8.7 we know that $g \circ h$ is contractive.

We have $h(g\,x) \leq g\,x$ since $g\,x \in T^*$ and h is decreasing on T^*. For $h\,x$, we have $h(g\,x) \leq h\,x$ since h is monotone and g is decreasing on T^*.

Hence $h(g\,x)$ is a lower bound on $g\,x$ and $h\,x$. ■

Since contractive functions are also closed under infima of codirected families, it is clear that there is a smallest contractive function.

Lemma 8.10 Define the function f^∞ as the infimum of M_x.

$$f^\infty\,x := \bigwedge M_x$$

The function f^∞ is contractive and thus is the smallest contractive function.

Proof Let $x \leq y$, we have $f^\infty\,x \leq f^\infty\,y$ just when $f^\infty\,x \leq g\,y$ for all contractive functions g. This follows since $f^\infty\,x \leq g\,x \leq g\,y$ by definition of f^∞ and monotonicity of g.

We have $f^\infty\,x \leq x$ since the identity function is contractive by Fact 8.7.

Finally, we have $f^\infty\,x \in T^*$ whenever $x \in T^*$, since T^* is codirected-closed by definition and $f^\infty\,x = \bigwedge\{ g\,x \mid g \text{ contractive} \}$ is codirected by Lemma 8.9 and $g\,x \in T^*$ since each contractive function preserves T^*. ■

At this point we can repeat the proof of Lemma 6.18 with T^* in place of T.

Lemma 8.11 When f is monotone we have $\nu\,f = f^\infty\,\top$.

Proof To show $\nu\,f \leq f^\infty\,\top$ it suffices to show that $f^\infty\,\top \leq f(f^\infty\,\top)$, but this is clear since both f and f^∞ are contractive and contractive functions are closed under composition by Fact 8.8, Lemma 8.10, and Fact 8.7.

In the reverse direction we have $f^\infty \top \in T^*$ since $\top \in T^*$ by Fact 8.4 and f^∞ preserves T^* by Lemma 8.10. But then clearly $x \in T$, since T^* has fewer constructors than T, and hence $\nu\, f \leq x$ Lemma 6.18. ■

The induction principle of T^* yields the codirected coinduction principle.

Theorem 8.12 (Codirected Coinduction) Let $f : X \to X$ be a monotone function on a complete lattice, P a codirected-closed predicate on X. Then $P(\nu f)$ holds whenever $P\,x$ implies $P(f\,x)$ for all x.

Proof By Lemma 8.11 we have $\nu f = f^\infty \top \in T^*$ and it suffices to show that $P\,x$ holds for all $x \in T^*$. This follows by induction on the derivation of $x \in T^*$. ■

By duality we obtain the corresponding result for least fixed-points.

Theorem 8.13 (Directed Induction) Let $f : X \to X$ be a monotone function on a complete lattice, P a directed-closed predicate on X. Then $P(\mu f)$ holds whenever $P\,x$ implies $P(f\,x)$ for all x.

Proof By Theorem 8.12 instantiated with the reverse complete lattice X^r. The function f is still monotone when seen as a function on X^r and P is a codirected closed predicate on X^r. Hence $P(\mu f)$ holds, since the least fixed-point on X is the greatest fixed-point on X^r. ■

8.2 Abstract Axiomatic Semantics

Before we describe the axiomatic semantics of GC and IC we first discuss common features of axiomatic semantics. An abstract predicate transformer is given by a function pt mapping programs into monotone predicate transformers from a postcondition to a corresponding precondition. In the general case, pt has the following type.

$$\mathrm{pt} \;:\; \mathbb{T} \to (Y \to \mathcal{P}) \to_m X \to \mathcal{P}$$

Where $\mathbb{T}$ is the type of programs, X is the type of initial states and Y is the type of final states. We write $A \to_m B$ for the type of monotone functions from A to B.

As described in the introduction we base our axiomatic semantics on a more primitive notion of an “evaluation functional”. An abstract evaluation functional is a monotone function E between predicate transformers. Hence, an abstract evaluation functional has the following type.

$$E \;:\; (\mathbb{T} \to (Y \to \mathcal{P}) \to_m X \to \mathcal{P}) \;\to_m\; (\mathbb{T} \to (Y \to \mathcal{P}) \to_m X \to \mathcal{P})$$

From the evaluation functional we obtain the weakest precondition (wp) and the weakest liberal precondition (wlp), total and partial correctness judgments ($\langle\sigma\rangle\, s\, \langle Q\rangle$, $\{\sigma\}\, s\, \{Q\}$), and Hoare triples for total and partial correctness using the least and greatest fixed-point of evaluation respectively (Theorem 6.11).

$$\begin{aligned}
\mathrm{wp} &= \mu E \\
\mathrm{wlp} &= \nu E \\
\langle\sigma\rangle\, s\, \langle Q\rangle &= \mathrm{wp}\, s\, Q\, \sigma \\
\{\sigma\}\, s\, \{Q\} &= \mathrm{wlp}\, s\, Q\, \sigma \\
\langle P\rangle\, s\, \langle Q\rangle &= P \subseteq \mathrm{wp}\ s\, Q \\
\{P\}\, s\, \{Q\} &= P \subseteq \mathrm{wlp}\ s\, Q
\end{aligned}$$

Intuitively, E describes the weakest precondition of a program without fixing the notion of partial or total correctness. The predicate transformers wp and wlp then describe the weakest precondition transformer and weakest liberal precondition transformer of a program.

From the definition it is clear that total correctness implies partial correctness.

Fact 8.14 If $\langle\sigma\rangle\, s\, \langle Q\rangle$ then $\{\sigma\}\, s\, \{Q\}$. Equivalently, $\mathrm{wp} \subseteq \mathrm{wlp}$.

Proof We have $\mu f \leq \nu f$ for all monotone functions f. ∎

We only consider monotone predicate transformers throughout and so we have the following properties for wp and wlp.

Fact 8.15 For $P \subseteq Q$ we have $\mathrm{wp}\, s\, P \subseteq \mathrm{wp}\, s\, Q$ and $\mathrm{wlp}\, s\, P \subseteq \mathrm{wlp}\, sQ$.

We can prove structural properties of wp and wlp from properties of E using directed induction and coinduction.

For example, if E was defined on potentially non-monotone predicate transformers we would still obtain monotone predicate transformers wp and wlp as long as E can be shown to preserve monotonicity.

In the remainder of this section we assume that E is a fixed evaluation functional.

Typically, predicate transformers are distributive over the postcondition.

Definition 8.16 A predicate transformer pt is **distributive** if for all s, P, Q we have

$$\mathrm{pt}\ s\, P \cap \mathrm{pt}\ s\, Q \subseteq \mathrm{pt}\ s\, (P \cap Q)$$

Note that the reverse direction holds by monotonicity and so a predicate transformer is distributive if

$$\mathrm{pt}\ s\,P \cap \mathrm{pt}\ s\,Q \;=\; \mathrm{pt}\ s\,(P \cap Q)$$

which further justifies the name.

Lemma 8.17 Whenever E preserves distributivity, both wp and wlp are distributive.

Proof Being distributive is inf-closed and directed-closed, hence the result follows by (co)directed (co)induction. ■

Usually the following stronger form of distributivity of wlp over wp holds.

Definition 8.18 A predicate transformer pt' **distributes over** a predicate transformer pt if

$$\mathrm{pt}\ s\,P \cap \mathrm{pt}'\,s\,Q \;\subseteq\; \mathrm{pt}\ s\,(P \cap Q)$$

holds for all s, P, Q.

Lemma 8.19 If $E\ \mathrm{pt}'$ distributes over $E\ \mathrm{pt}$ whenever pt' distributes over pt, then

- wlp distributes over wp ($\mathrm{wp}\,s\,P \cap \mathrm{wlp}\,s\,Q = \mathrm{wp}\,s\,(P \cap Q)$)
- wlp is distributive ($\mathrm{wlp}\,s\,P \cap \mathrm{wlp}\,s\,Q = \mathrm{wlp}\,s\,(P \cap Q)$)
- wp is distributive ($\mathrm{wp}\,s\,P \cap \mathrm{wp}\,s\,Q = \mathrm{wp}\,s\,(P \cap Q)$)
- $\langle\sigma\rangle\ s\ \langle Q\rangle$ holds iff $\{\sigma\}\ s\ \{Q\}$ and $\langle\sigma\rangle\ s\ \langle\top\rangle$ hold

Proof The predicate which states that wlp distributes over a given predicate transformer is both inf-closed and sup-closed and so the first two statements follow by (co)directed (co)induction. The third statement follows from Fact 8.14, while the final statement is a special case of the first. ■

Many predicate transformer semantics are continuous in the sense that they commute with non-empty directed suprema. In particular, this implies that the predicate transformer commutes with suprema of ω-chains[2]

[2] Classically, commuting with suprema of ω-chains is often taken to be the definition of continuity. In the presence of excluded middle and choice axioms the two notions coincide, but constructively the definition of continuity in terms of directed families is stronger.

Definition 8.20 A predicate transformer pt is **continuous**, if it commutes with unions of non-empty directed families, i.e., if we have

$$\mathrm{pt}\ s\left(\bigcup \mathcal{D}\right) \subseteq \bigcup_{Q \in \mathcal{D}} \mathrm{pt}\ s\, Q$$

for every program s and directed family $\mathcal{D}$ of postconditions.

By monotonicity, a predicate transformer is continuous if the equation

$$\bigcup_{Q \in \mathcal{D}} \mathrm{pt}\ s\, Q = \mathrm{pt}\ s\left(\bigcup \mathcal{D}\right)$$

holds for all non-empty directed families $\mathcal{D}$.

Lemma 8.21 If E preserves continuity, then wp is continuous.

Proof We show that being continuous is sup-closed and the result follows by directed induction.

Let pt_i be a family of continuous predicate transformers. We have

$$\begin{aligned} \bigcup_{Q \in \mathcal{D}} \bigcup_{i} \mathrm{pt}_i\ s\, Q &= \bigcup_{i} \bigcup_{Q \in \mathcal{D}} \mathrm{pt}_i\ s\, Q \\ &= \bigcup_{i} \mathrm{pt}_i\ s\left(\bigcup \mathcal{D}\right) \end{aligned}$$

■

Note that continuity is not inf-closed and thus even if E preserves continuity, wlp may not be continuous. In fact, wlp usually is not continuous in this sense. This is the main reason why we consider only monotone, but not necessarily continuous predicate transformers.

On the other hand, continuity implies that the definition of fixed points can be simplified as in Kleene's fixed point theorem.

Lemma 8.22 Let X be a complete lattice, $f : X \to X$ be a continuous function on X, i.e., a monotone function such that

$$f\left(\bigcup F\right) \leq \bigcup_{i} f(F\, i)$$

for every non-empty directed family F.

Then the least fixed point of f can be reached by ω-iteration.

$$\mu f = \bigcup_{n:\mathbb{N}} f^n \perp$$

Finally, for deterministic languages we have a stronger form of continuity.

Definition 8.23 A predicate transformer pt is **deterministic** if

$$\mathrm{pt}\ s\,Q \subseteq \bigcup_{\tau \in Q} \mathrm{pt}\ s\,\{\tau\}$$

for all s, Q, where $\{\tau\}$ refers to the singleton postcondition $(\lambda\tau'.\ (\tau = \tau'))$.

Another way to state this property is to say that pt commutes with arbitrary suprema.

$$\mathrm{pt}\ s\,(\bigcup M) \ = \ \bigcup_{\tau \in \bigcup M} \mathrm{pt}\ s\,\{\tau\} \ = \ \bigcup_{Q \in M} \bigcup_{\tau \in Q} \mathrm{pt}\ s\,\{\tau\} \ = \ \bigcup_{Q \in M} \mathrm{pt}\ s\,Q$$

The converse follows since $Q = \bigcup_{\tau \in Q} \{\tau\}$.

In particular, we have the following.

Fact 8.24 A deterministic predicate transformer is continuous.

Lemma 8.25 If E preserves deterministic predicate transformers, then wp is deterministic.

Proof Being a deterministic predicate transformer is sup-closed and the result follows by directed induction. ■

Conversely, continuity can be seen as a weakening of determinism in which a program may have only “finitely many” results instead of at most one. This is due to the fact that non-empty directed unions commute with finite meets under mild assumptions.

Lemma 8.26 Let $F : I \to J \to \ldots$ be a family of predicate transformers, directed in I, with I non-empty, and let s be a finite list over J. Then we have

$$\bigcap_{j \in s} \bigcup_{i:I} F\,i\,j \ = \ \bigcup_{i:I} \bigcap_{j \in s} F\,i\,j$$

Proof The right-to-left direction holds without assumptions in every complete lattice. We show the left-to-right direction by induction on s.

If s is empty, then

$$\bigcap_{j \in []} \bigcup_{i} F\,i\,j \ = \ \top \ = \ \bigcup_{i} \top \ = \ \bigcup_{i} \bigcap_{j \in []} F\,i\,j$$

since J is assumed to be non-empty.

If s is of the form $a \cdot t$ we have

$$\begin{aligned}
\bigcap_{j \in a \cdot t} \bigcup_i F\,i\,j &= (\bigcup_i F\,i\,a) \cap (\bigcap_{j \in t} \bigcup_i F\,i\,j) \\
&= \bigcup_i F\,i\,a \cap (\bigcup_i \bigcap_{j \in s} F\,i\,j) && \text{by induction} \\
&= \bigcup_{i,i'} F\,i\,a \cap (\bigcap_{j \in s} F\,i'\,j) \\
&= \bigcup_i F\,i\,a \cap (\bigcap_{j \in s} F\,i\,j) && F \text{ directed} \\
&= \bigcup_i \bigcap_{j \in a \cdot s} F\,i\,j
\end{aligned}$$

■

Note that neither Lemma 8.26, Lemma 8.17, not Lemma 8.19 hold in arbitrary complete lattices, since the proofs made use of the fact that we can commute binary meets with suprema ($P \cap \bigcup D = \bigcup_{Q \in D} P \cap Q$). This is the defining property of a complete Heyting algebra, and indeed these lemmas hold in every complete Heyting algebra.

8.3 Guarded Commands

Dijkstra's language of guarded commands [41] is an imperative language with underspecified execution order. We introduce an abstract guarded command language GC whose states are taken from an abstract type Σ. Assignments are replaced with **actions**, which are abstract functions from states to states. **Guards** are modeled as boolean predicates on states. The syntax of GC is as follows:

$$\begin{aligned}
\sigma, \tau &: \Sigma && \text{states} \\
a &: \Sigma \to \Sigma && \text{actions} \\
b &: \Sigma \to \mathbb{B} && \text{guards} \\
\mathbb{C} \ni s, t &:= \textsf{skip} \mid a \mid s; t \mid \textsf{if } G \mid \textsf{do } G && \text{programs} \\
\mathbb{G} \ni G &:= b_1 \Rightarrow s_1 \parallel \ldots \parallel b_n \Rightarrow s_n && (n \geq 0)
\end{aligned}$$

Conditionals if G and **loops do** G work on a **guarded command set** G, which is realized as a non-empty list of **guarded commands** $b \Rightarrow s$. The term set is justified since the order of the guarded commands does not matter semantically. We write $\varnothing$ for the empty guarded command set and $(b \Rightarrow s) \in G$ for list inclusion.

We describe the execution of conditionals and loops informally. The execution

of a conditional **if** G selects a guarded command in G whose guard is satisfied and executes the program of the command. If no guard in G is satisfied, the execution is stuck. The execution of a loop **do** G repeatedly executes guarded commands from G whose guard is satisfied. Execution of the loop terminates once none of the guards are satisfied. When the guards of several commands are satisfied, any of the commands may be chosen for execution. We say that the execution order of GC is underspecified.

We give informal examples of programs where we instantiate Σ, a, b with assignments and assignment statements respectively.

Example 8.27 (Greatest Common Divisor) The following program computes the gcd of two positive integers x and y.

$$\textbf{do } x > y \Rightarrow x := x - y \parallel y > x \Rightarrow y := y - x$$

We formalize the semantics of GC with an evaluation functional as described in Chapter 8.2.

We write $\underline{\mathrm{wp}}$ for the application of the evaluation functional to the predicate transformer wp. Furthermore, we write wp^* for the extension of wp to guarded command sets and $\widehat{G}$ for the (decidable) predicate which states that there is some guard which is satisfied.

$$\begin{aligned}
\underline{\mathrm{wp}}\ \textbf{skip}\ Q &:= Q \\
\underline{\mathrm{wp}}\ a\ Q &:= a \circ Q \\
\underline{\mathrm{wp}}\ (s;t)\ Q &:= \mathrm{wp}\ s\ (\mathrm{wp}\ t\ Q) \\
\underline{\mathrm{wp}}\ (\textbf{if}\ G)\ Q &:= \widehat{G} \cap \mathrm{wp}^*\ G\ Q \\
\underline{\mathrm{wp}}\ (\textbf{do}\ G)\ Q &:= \lambda\sigma. \begin{cases} \mathrm{wp}^*\ G\ (\mathrm{wp}(\textbf{do}\ G)\ Q)\ \sigma & \text{if } \widehat{G}\,\sigma \\ Q\,\sigma & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\widehat{G} &:= \lambda\sigma.\ \exists(b \Rightarrow s) \in G.\ b\,\sigma \\
\mathrm{wp}^*\ G\,Q &:= \lambda\sigma.\ \forall(b \Rightarrow s) \in G.\ b\,\sigma \to \mathrm{wp}\ s\ Q\,\sigma
\end{aligned}$$

The definition of $\underline{\mathrm{wp}}$ formalizes the intuitive explanation of the semantics of GC. The **skip** command and sequencing $s;t$ are realized with the identity predicate transformer and the composition of predicate transformers respectively. Individual actions operate on the postcondition by precomposition. The precondition of a conditional **if** G is only satisfied in a state where some guard in G is satisfied, i.e., $\widehat{G}$ holds. Moreover, every command in G whose guard is satisfied by a state σ must

satisfy the postcondition for σ.

Writing $b \Rightarrow P$ for the predicate $\lambda\sigma.\ b\,\sigma \rightarrow P\,\sigma$, we can characterize wp^* as follows.

$$\mathrm{wp}^*\ G\,Q \ = \ \bigcap\{\,b \Rightarrow \mathrm{wp}\,s\,Q \mid (b \Rightarrow s) \in G\,\}$$

Finally, the semantics of loops are handled by unfolding. When a guard in G is satisfied, the loop **do** G is equivalent to the program **if** G; **do** G, otherwise the loop is equivalent to the empty program **skip**.

We can also characterize the semantics of **do** G as follows.

$$\underline{\mathrm{wp}}\,(\textbf{do}\ G)\,Q \ = \ (\widehat{G} \Rightarrow \mathrm{wp}^*\ G\,(\mathrm{wp}\,(\textbf{do}\ G)\,Q)) \ \cap \ (\neg\widehat{G} \Rightarrow Q)$$

It is clear that $\underline{\mathrm{wp}}$ preserves monotonicity and is itself a monotone function between monotone predicate transformers. Thus, as explained in Chapter 8.2, we obtain weakest and weakest liberal preconditions, total and partial correctness judgments, and total and partial Hoare-triples from the evaluation functional.

Apart from the semantics of loops $\underline{\mathrm{wp}}\,(\textbf{do}\ G)$, the definition of $\underline{\mathrm{wp}}$ is modular in that the semantics of a program is defined in terms of the semantics of its parts. We give a fully modular recursive specification of wp and wlp by computing the semantics of loops via least or greatest fixed-points.

Lemma 8.28 The weakest precondition transformer (wp) of GC satisfies the following equations.

$$\begin{aligned}
\mathrm{wp}\,\textbf{skip}\,Q &= Q \\
\mathrm{wp}\,a\,Q &= a \circ Q \\
\mathrm{wp}(s;t)\,Q &= \mathrm{wp}\,s\,(\mathrm{wp}\,t\,Q) \\
\mathrm{wp}(\textbf{if}\ G)\,Q &= \widehat{G} \cap \mathrm{wp}^*\ G\,Q \\
\mathrm{wp}(\textbf{do}\ G)\,Q &= \mu\,(\lambda P\sigma.\ \text{if}\ \widehat{G}\,\sigma\ \text{then}\ \mathrm{wp}^*\ G\,P\,\sigma\ \text{else}\ Q\,\sigma)
\end{aligned}$$

Analogously, the weakest liberal preconditions of GC satisfy the following equations

$$\begin{aligned}
\mathrm{wlp}\,\textbf{skip}\,Q &= Q \\
\mathrm{wlp}\,a\,Q &= a \circ Q \\
\mathrm{wlp}(s;t)\,Q &= \mathrm{wlp}\,s\,(\mathrm{wlp}\,t\,Q) \\
\mathrm{wlp}(\textbf{if}\ G)\,Q &= \widehat{G} \cap \mathrm{wlp}^*\ G\,Q \\
\mathrm{wlp}(\textbf{do}\ G)\,Q &= \nu\,(\lambda P\sigma.\ \text{if}\ \widehat{G}\,\sigma\ \text{then}\ \mathrm{wlp}^*\ G\,P\,\sigma\ \text{else}\ Q\,\sigma)
\end{aligned}$$

Proof The cases for **skip**, a, $s;t$, and **if** G follow by unfolding fixed-points. The case for **do** G follows from monotonicity and (co)directed (co)induction. ∎

Lemma 8.28 can be used to give a recursive definition of wp and wlp. Meanwhile, the total and partial correctness judgments of GC admit inductive definitions.

Lemma 8.29 The total correctness judgment of GC $\langle\sigma\rangle\, s\, \langle Q\rangle$ can be inductively defined by the following rules.

$$\frac{Q\,\sigma}{\langle\sigma\rangle\,\textbf{skip}\,\langle Q\rangle} \qquad \frac{Q(a\,\sigma)}{\langle\sigma\rangle\, a\,\langle Q\rangle} \qquad \frac{\langle\sigma\rangle\, s\,\langle P\rangle \qquad \langle P\rangle\, t\,\langle Q\rangle}{\langle\sigma\rangle\, s;t\,\langle Q\rangle}$$

$$\frac{\widehat{G}\,\sigma \qquad \forall (b \Rightarrow s) \in G.\; b\sigma \to \langle\sigma\rangle\, s\,\langle Q\rangle}{\langle\sigma\rangle\,\textbf{if}\ G\,\langle Q\rangle} \qquad \frac{\langle\sigma\rangle\,\textbf{if}\ G\,\langle P\rangle \qquad \langle P\rangle\,\textbf{do}\ G\,\langle Q\rangle}{\langle\sigma\rangle\,\textbf{do}\ G\,\langle Q\rangle}$$

$$\frac{\neg(\widehat{G}\,\sigma) \qquad Q\sigma}{\langle\sigma\rangle\,\textbf{do}\ G\,\langle Q\rangle}$$

The partial correctness judgment of GC $\{\sigma\}\, s\,\{Q\}$ can be coinductively defined by the same rules, or inductively defined by the following rules.

$$\frac{Q\,\sigma}{\{\sigma\}\,\textbf{skip}\,\{Q\}} \qquad \frac{Q(a\,\sigma)}{\{\sigma\}\, a\,\{Q\}} \qquad \frac{\{\sigma\}\, s\,\{P\} \qquad \{P\}\, t\,\{Q\}}{\{\sigma\}\, s;t\,\{Q\}}$$

$$\frac{\widehat{G}\,\sigma \qquad \forall (b \Rightarrow s) \in G.\; b\sigma \to \{\sigma\}\, s\,\{Q\}}{\{\sigma\}\,\textbf{if}\ G\,\{Q\}} \qquad \frac{I\,\sigma \qquad \{I \cap \widehat{G}\}\,\textbf{if}\ G\,\{I\} \qquad I \cap \neg\widehat{G} \subseteq Q}{\{\sigma\}\,\textbf{do}\ G\,\{Q\}}$$

Proof The characterization of total correctness judgments is clear from the definition by unfolding fixed-points, using monotonicity in the case of sequencing and loops. The characterization of partial correctness judgments follows from Lemma 8.28 with the characterization of the greatest fixed-point from Theorem 6.11. ∎

The rule for sequential compositions $s;t$ in Lemma 8.29 uses a predicate P that serves as postcondition for s and as precondition for t. We refer to P as an **interpolant**. The rule for loops also uses an interpolant. The structure of the rules for the total correctness judgments is similar to the structure of the rules for a big-step semantics, where interpolants and postconditions appear as single states.

Additionally, the characterization of partial correctness judgments is close to the typical presentation of axiomatic semantics using Hoare triples. To establish a triple $\{\sigma\}$ **do** $s\,\{Q\}$ we have to find an interpolant I which holds at σ and is preserved by s.

The usual proof rules for Hoare triples $\{P\}\,s\,\{Q\}$ can be derived from Lemma 8.29 by expanding the definitions.

Corollary 8.30 The following rules are admissible for partial correctness judgments.

$$\frac{}{\{P\}\,\textbf{skip}\,\{P\}} \qquad \frac{}{\{a \circ Q\}\,a\,\{Q\}} \qquad \frac{\{P\}\,s\,\{Q\} \qquad \{Q\}\,t\,\{R\}}{\{P\}\,s;t\,\{R\}}$$

$$\frac{P \subseteq \widehat{G} \qquad \{P\}\,G\,\{Q\}}{\{P\}\,\textbf{if}\;G\,\{Q\}} \qquad \frac{\{I \cap \widehat{G}\}\,G\,\{I\}}{\{I\}\,\textbf{do}\;G\,\{I \setminus \widehat{G}\}}$$

$$\{P\}\,G\,\{Q\} \;:=\; \forall (b \Rightarrow s) \in G.\ \{P \cap b\}\,s\,\{Q\}$$

The predicate transformer semantics of GC is distributive in the sense of Definition 8.16.

Lemma 8.31 wp and wlp are distributive and wlp distributes over wp.

Proof By Lemma 8.19 it suffices to show that $\underline{\mathrm{wlp}}$ distributes over $\underline{\mathrm{wp}}$ whenever wlp distributes over wp for arbitrary predicate transformers $\mathrm{wlp}, \mathrm{wp}$.

This holds by unfolding definitions in the case of **skip** and actions a. For sequencing, we have

$$\begin{aligned}
\underline{\mathrm{wp}}\,(s;t)\,P \cap \underline{\mathrm{wlp}}\,(s;t)\,Q &= \mathrm{wp}\,s\,(\mathrm{wp}\,t\,P\,) \cap \mathrm{wlp}\,s\,(\mathrm{wlp}\,t\,Q\,) \\
&= \mathrm{wp}\,s\,(\mathrm{wp}\,t\,P\, \cap \mathrm{wlp}\,t\,Q\,) \\
&= \mathrm{wp}\,s\,(\mathrm{wp}\,t\,(P \cap Q)\,) \\
&= \underline{\mathrm{wp}}\,(s;t)\,(P \cap Q)
\end{aligned}$$

For conditionals, we have

$$\begin{aligned}
\underline{\mathrm{wp}}\,(\textbf{if}\;G)\,P \cap \underline{\mathrm{wlp}}\,(\textbf{if}\;G)\,Q &= \widehat{G} \cap \mathrm{wp}^*\;G\,P \;\cap\; \widehat{G} \cap \mathrm{wlp}^*\;G\,Q \\
&= \widehat{G} \cap \bigcap \{\, b \Rightarrow \mathrm{wp}\,s\,P\, \cap \mathrm{wlp}\,s\,Q \mid (b \Rightarrow s) \in G\,\} \\
&= \widehat{G} \cap \bigcap \{\, b \Rightarrow \mathrm{wp}\,s\,(P \cap Q) \mid (b \Rightarrow s) \in G\,\} \\
&= \widehat{G} \cap \mathrm{wp}^*\;G\,(P \cap Q) \\
&= \underline{\mathrm{wp}}\,(\textbf{if}\;G)\,(P \cap Q)
\end{aligned}$$

For loops, we have

$$
\begin{aligned}
\underline{\mathrm{wp}}\,(\textbf{do}\ G)\,P \cap \underline{\mathrm{wlp}}\,(\textbf{do}\ G)\,Q \;&=\; (\widehat{G} \Rightarrow \mathrm{wp}^*\ G\,(\mathrm{wp}\,(\textbf{do}\ G)\,P)) \cap (\neg\widehat{G} \Rightarrow P) \\
&\cap\; (\widehat{G} \Rightarrow \mathrm{wlp}^*\ G\,(\mathrm{wlp}\,(\textbf{do}\ G)\,Q)) \cap (\neg\widehat{G} \Rightarrow Q) \\
&=\; (\widehat{G} \Rightarrow \mathrm{wp}^*\ G\,(\mathrm{wp}\,(\textbf{do}\ G)\,P) \cap \mathrm{wlp}^*\ G\,(\mathrm{wlp}\,(\textbf{do}\ G)\,Q)) \\
&\cap\; (\neg\widehat{G} \Rightarrow P \cap Q) \\
&=\; (\widehat{G} \Rightarrow \mathrm{wp}^*\ G\,(\mathrm{wp}\,(\textbf{do}\ G)\,P \cap \mathrm{wlp}\,(\textbf{do}\ G)\,Q)) \\
&\cap\; (\neg\widehat{G} \Rightarrow P \cap Q) \\
&=\; (\widehat{G} \Rightarrow \mathrm{wp}^*\ G\,(\mathrm{wp}\,(\textbf{do}\ G)\,(P \cap Q))) \\
&\cap\; (\neg\widehat{G} \Rightarrow P \cap Q) \\
&=\; \underline{\mathrm{wp}}\,(\textbf{do}\ G)\,(P \cap Q)
\end{aligned}
$$
∎

While GC is not deterministic, the predicate transformer semantics of GC is continuous.

Lemma 8.32 The predicate transformer wp is continuous.

Proof By Lemma 8.21 it suffices to show that $\underline{\mathrm{wp}}$ is continuous whenever wp is.

First note that wp^* is continuous whenever wp is. Let $\mathcal{D}$ be a non-empty directed family of postconditions. We have

$$
\begin{aligned}
\mathrm{wp}^*\ G\,(\bigcup\mathcal{D}) &= \bigcap \left\{ b \Rightarrow \mathrm{wp}\ s\ \left(\bigcup\mathcal{D}\right) \mid (b \Rightarrow s) \in G \right\} \\
&= \bigcap \left\{ b \Rightarrow \bigcup_{Q\in\mathcal{D}} \mathrm{wp}\ s\,Q \mid (b \Rightarrow s) \in G \right\} \\
&= \bigcap_{(b\Rightarrow s)\in G}\ \bigcup_{Q\in\mathcal{D}} b \Rightarrow \mathrm{wp}\ s\,Q \\
&= \bigcup_{Q\in\mathcal{D}} \bigcap \{ b \Rightarrow \mathrm{wp}\ s\,Q \mid (b \Rightarrow s) \in G \} && \text{by Lemma 8.26} \\
&= \bigcup_{Q\in\mathcal{D}} \mathrm{wp}^*\ G\,Q
\end{aligned}
$$

Where we have used the directedness of $\mathcal{D}$ (and hence of $b \Rightarrow \mathrm{wp}\ s\,Q$ for $Q \in \mathcal{D}$) to commute the finite intersection over the supremum (Lemma 8.26).

The remainder of the proof consists of equational reasoning analogous to Lemma 8.31. ∎

8.4 Imperative Continuations

The second language we study is a low-level language with linear sequential composition and lexically scoped gotos which we call IC. We see the declaration of a target for a goto as the definition of an argumentless, possibly recursive procedure to be used as a continuation. IC is an idealized version of an intermediate language IL used in previous work on compiler verification [109].

We define the syntax of IC with a family of terms $\mathbb{T} : \mathbb{N} \to \mathcal{U}$ indexed by the size of a context of terms.

$$\mathbb{T}^k \ni s^k, t^k \; := \; a; s^k \mid \textbf{if}\; b \;\textbf{then}\; s^k \;\textbf{else}\; t^k \mid \textbf{def}\; s^{k+1} \;\textbf{in}\; t^{k+1} \mid f^k \qquad \text{where } f^k \in \mathsf{I}_k$$

Actions a and guards b are as in GC. Capture-avoiding instantiation $s[\theta]$ is defined on programs as usual. We write θ for substitutions, since we are using σ, τ to refer to states.

Instantiation satisfies the following equations, along with the general laws of the σ-calculus.

$$\begin{aligned}(a; s)[\theta] &= a; s[\theta] \\ (\textbf{if}\; b \;\textbf{then}\; s \;\textbf{else}\; t)[\theta] &= \textbf{if}\; b \;\textbf{then}\; s[\theta] \;\textbf{else}\; t[\theta] \\ (\textbf{def}\; s \;\textbf{in}\; t)[\theta] &= \textbf{def}\; s[\Uparrow\theta] \;\textbf{in}\; t[\Uparrow\theta] \\ f[\theta] &= \theta\, f\end{aligned}$$

Operational Semantics We define a small-step operational semantics for IC. The semantics is formulated as an inductive predicate providing judgements $(s, \sigma) \rhd (t, \tau)$ describing single execution steps between configurations. A configuration (s, σ) is a pair of a program and a state.

$$\frac{}{(a; s, \sigma) \rhd (s, a\sigma)} \qquad \frac{}{(\textbf{def}\; s \;\textbf{in}\; t, \sigma) \rhd (t[\textbf{def}\; s \;\textbf{in}\; s \cdot \mathsf{id}], \sigma)}$$

$$\frac{b\sigma}{(\textbf{if}\; b \;\textbf{then}\; s \;\textbf{else}\; t, \sigma) \rhd (s, \sigma)} \qquad \frac{\neg b\sigma}{(\textbf{if}\; b \;\textbf{then}\; s \;\textbf{else}\; t, \sigma) \rhd (t, \sigma)}$$

Things are arranged such that terminal configurations are pairs (f, σ) consisting of a label and a state. We see such configurations as calls to external continuations. By design, a closed program such as

$$\Omega \; := \; \textbf{def}\; 0 \;\textbf{in}\; 0$$

must loop forever.

Local continuations are introduced with programs of the form **def** s **in** t, where both s and t have an additional free variable. Informally, execution of a program **def** s **in** t binds the free variable to the program s and proceeds with the execution of the program t. The small-step semantics realizes this idea by reducing the program **def** s **in** t to its unfolding $t[\textbf{def}\ s\ \textbf{in}\ s \cdot \mathsf{id}]$. Since s is bound in itself as well as in the body of the definition this rule provides for recursive and lexically scoped continuations.

Example 8.33 (Greatest Common Divisor in IC) The following program computes the greatest common divisor of x and y and terminates with a call to the external continuation ret.

$$\begin{aligned}&\textbf{def}\ \ \textbf{if}\ x > y\ \textbf{then}\ x := x - y; 0\\ &\qquad\ \textbf{else if}\ y > x\ \textbf{then}\ y := y - x; 0\ \textbf{else}\ \mathsf{ret}\\ &\textbf{in}\ 0\end{aligned}$$

8.5 Axiomatic Semantics of IC

Just as in GC, we describe the axiomatic semantics of IC using an evaluation functional. We write $\underline{\mathrm{wp}}$ for the application of the evaluation functional to the predicate transformer wp. Unlike in GC, in IC we make a distinction between initial and final states. The initial states of IC are drawn from the type Σ as before, but the final states are pairs of an index, which represents the name of the final continuation, and a final state drawn from Σ. Hence predicate transformers for IC would have the type

$$\forall n.\ \mathbb{T}^n \to (\mathsf{I}_n \times \Sigma \to \mathcal{P}) \to \Sigma \to \mathcal{P}$$

To ease notation and to reuse the "scons" operation on substitutions we will curry the postcondition and work with predicate transformers taking a family of postconditions instead.

$$\mathrm{wp}\ :\ \forall n.\ \mathbb{T}^n \to (\mathsf{I}_n \to \Sigma \to \mathcal{P}) \to \Sigma \to \mathcal{P}$$

With this convention, we define the evaluation functional of IC as follows.

$$\begin{aligned}
\underline{\mathrm{wp}}\,(a; s)\,Q &:= a \circ \mathrm{wp}\ s\,Q \\
\underline{\mathrm{wp}}\,f\,Q &:= Q\,f \\
\underline{\mathrm{wp}}\,(\textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t)\,Q &:= \lambda\sigma. \begin{cases} \mathrm{wp}\ s\,Q\,\sigma & \text{if } b\,\sigma \\ \mathrm{wp}\ t\,Q\,\sigma & \text{otherwise} \end{cases} \\
\underline{\mathrm{wp}}\,(\textbf{def}\ s\ \textbf{in}\ t)\,Q &:= \mathrm{wp}\ t\,(\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ s) \cdot Q)
\end{aligned}$$

Just as for GC, the predicate transformer semantics of IC is monotone and admits a modular characterization.

Lemma 8.34 The weakest precondition transformer (wp) of IC satisfies the following equations.

$$\begin{aligned}
\mathrm{wp}\,(a; s)\,Q &= a \circ \mathrm{wp}\ s\,Q \\
\mathrm{wp}\ f\,Q &= Q\,f \\
\mathrm{wp}\,(\textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t)\,Q &= \lambda\sigma. \begin{cases} \mathrm{wp}\ s\,Q\,\sigma & \text{if } b\,\sigma \\ \mathrm{wp}\ t\,Q\,\sigma & \text{otherwise} \end{cases} \\
\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ t)\,Q &= \mathrm{wp}\ t\ (I \cdot Q) \\
\text{where } I &= \mu\,(\lambda P.\ \mathrm{wp}\ s\ (P \cdot Q))
\end{aligned}$$

Analogously, the weakest liberal preconditions of IC satisfy the following equations.

$$\begin{aligned}
\mathrm{wlp}\,(a; s)\,Q &= a \circ \mathrm{wlp}\ s\,Q \\
\mathrm{wlp}\ f\,Q &= Q\,f \\
\mathrm{wlp}\,(\textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t)\,Q &= \lambda\sigma. \begin{cases} \mathrm{wlp}\ s\,Q\,\sigma & \text{if } b\,\sigma \\ \mathrm{wlp}\ t\,Q\,\sigma & \text{otherwise} \end{cases} \\
\mathrm{wlp}\,(\textbf{def}\ s\ \textbf{in}\ t)\,Q &= \mathrm{wlp}\ t\ (I \cdot Q) \\
\text{where } I &= \nu\,(\lambda P.\ \mathrm{wlp}\ s\ (P \cdot Q))
\end{aligned}$$

Proof The cases for actions $a; s$, calls f, and conditionals **if** b **then** s **else** t follow by unfolding fixed-points. The for for recursive definitions **def** s **in** t follows by monotonicity and (co)directed (co)induction.

We illustrate the case for wp. Let $\underline{\mathrm{wp}}$ be a predicate transformer for which the

equation holds. We have

$$\begin{aligned}\underline{\mathrm{wp}}\,(\textbf{def}\ s\ \textbf{in}\ t)\,Q &= \mathrm{wp}\ t\,(\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ s)\,Q\cdot Q)\\ &= \mathrm{wp}\ t\,((\mathrm{wp}\ s\,(\mu(\lambda P.\ \mathrm{wp}\ s\,(P\cdot Q))\cdot Q)\ \cdot\ Q))\\ &= \mathrm{wp}\ t\,(\mu(\lambda P.\ \mathrm{wp}\ s\,(P\cdot Q))\cdot Q)\end{aligned}$$

by folding the fixed-point. ■

This modular characterization of wp and wlp implies that the predicate transformer semantics of IC are compatible with instantiation. For renaming this also follows directly by (co-)directed (co-)induction[3].

We prove this in two steps following the structure of instantiation, by first considering the case of renamings.

Lemma 8.35 For every renaming ξ we have

$$\begin{aligned}\mathrm{wp}\ s[\xi]\,Q &= \mathrm{wp}\ s\,(\xi\circ Q)\\ \mathrm{wlp}\ s[\xi]\,Q &= \mathrm{wlp}\ s\,(\xi\circ Q)\end{aligned}$$

Proof Since the statement is evidently inf- and sup-closed, it suffices to show that it is preserved by the evaluation functional and the result follows by (co)directed (co)induction.

The only interesting case is the one for recursive definitions. Assuming that the statement holds for a predicate transformer wp, we have

$$\begin{aligned}\underline{\mathrm{wp}}\,((\textbf{def}\ s\ \textbf{in}\ t)[\xi])\,Q &= \underline{\mathrm{wp}}\,(\textbf{def}\ s[\Uparrow\xi]\ \textbf{in}\ t[\Uparrow\xi])\,Q\\ &= \mathrm{wp}\,(t[\Uparrow\xi])\,(\mathrm{wp}\,(\textbf{def}\ s[\Uparrow\xi]\ \textbf{in}\ s[\Uparrow\xi])\,Q\ \cdot\ Q)\\ &= \mathrm{wp}\ t\ ((\Uparrow\xi)\circ((\mathrm{wp}\,((\textbf{def}\ s\ \textbf{in}\ s)[\xi])\,Q)\ \cdot\ Q))\\ &= \mathrm{wp}\ t\ (\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ s)\,(\xi\circ Q)\ \cdot\ \xi\circ Q)\end{aligned}$$

Since by the general laws of the σ-calculus $(\Uparrow\xi)\circ(P\cdot Q) = P\cdot(\xi\circ Q)$. ■

In particular, we obtain the weakening statement for IC.

Corollary 8.36 For all s, P, Q we have

$$\begin{aligned}\mathrm{wp}\ s[\uparrow]\,(P\cdot Q) &= \mathrm{wp}\ s\,Q\\ \mathrm{wlp}\ s[\uparrow]\,(P\cdot Q) &= \mathrm{wlp}\ s\,Q\end{aligned}$$

[3] This implies the soundness of (co-)induction up-to renaming.

Using weakening we obtain compatibility with instantiation.

Lemma 8.37 For every substitution σ we have

$$\begin{aligned} \text{wp}\ s[\sigma]\, Q &= \text{wp}\ s\,(\text{wp}\ \sigma\, Q) \\ \text{wlp}\ s[\sigma]\, Q &= \text{wlp}\ s\,(\text{wlp}\ \sigma\, Q) \end{aligned}$$

where $\text{wp}\ \sigma\, Q \ := \ \lambda i.\ \text{wp}\,(\sigma\, i)\, Q$ and analogously for $\text{wlp}\ \sigma\, Q$.

Proof By induction on s but otherwise analogous to the proof of Lemma 8.35, using Corollary 8.36 in the case of recursive definitions. ∎

And in particular, compatibility with single variable instantiation.

Corollary 8.38 For all s, t we have

$$\begin{aligned} \text{wp}\ s[t \cdot \mathsf{id}]\, Q &= \text{wp}\ s\,(\text{wp}\ t\, Q\ \cdot\ Q) \\ \text{wlp}\ s[t \cdot \mathsf{id}]\, Q &= \text{wlp}\ s\,(\text{wlp}\ t\, Q\ \cdot\ Q) \end{aligned}$$

Just as in GC, the predicate transformer semantics of IC is distributive.

Lemma 8.39 wp and wlp are distributive and wlp distributes over wp in IC.

Proof By Lemma 8.19 it suffices to show that $\underline{\text{wlp}}$ distributes over $\underline{\text{wp}}$ whenever wlp distributes over wp.

This follows directly by unfolding the definitions. The case of recursive definitions is analogous to the case for sequencing in GC. ∎

From the operational semantics of IC it is clear that IC is deterministic. The same holds for the predicate transformer semantics.

Lemma 8.40 The weakest precondition transformer wp of IC is deterministic.

Proof By Lemma 8.25 it suffices to show that the evaluation functional of IC preserves deterministic predicate transformers. The cases of actions, calls, and conditionals are clear by unfolding the definitions.

Assume that wp is a deterministic predicate transformer. In the case of recursive

definitions we have

$$\begin{aligned}
\underline{\mathrm{wp}}\,(\textbf{def}\ s\ \textbf{in}\ t)\,Q &= \mathrm{wp}\ t\,(\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ s)\,Q\ \cdot\ Q) \\
&= \mathrm{wp}\ t\,((\bigcup_{\tau\in Q} \mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ s)\,\{\tau\})\ \cdot\ \bigcup_{\tau\in Q}\{\tau\}) \\
&= \mathrm{wp}\ t\,(\bigcup_{\tau\in Q} (\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ s)\,\{\tau\}\cdot\{\tau\})) \\
&= \bigcup_{\tau\in Q} \mathrm{wp}\ t\,(\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ s)\,\{\tau\}\ \cdot\ \{\tau\}) \\
&= \bigcup_{\tau\in Q} \underline{\mathrm{wp}}\,(\textbf{def}\ s\ \textbf{in}\ t)\,\{\tau\}
\end{aligned}$$

■

In particular, wp is continuous, hence the least fixed-point in Lemma 8.34 can be defined in terms of finite unfolding of recursive definitions.

Lemma 8.41 Define the n-th unfolding of s, written s_n, as follows

$$\begin{aligned}
s_0 &:= \Omega \\
s_{n+1} &:= s[s_n \cdot \mathsf{id}]
\end{aligned}$$

We have

$$\mathrm{wp}\,(\textbf{def}\ s\ \textbf{in}\ t)\,Q = \bigcup_{n\in\mathbb{N}} \mathrm{wp}\ t[s_n\cdot\mathsf{id}]\,Q$$

Proof By Lemma 8.37 and Lemma 8.40 we have

$$\begin{aligned}
\bigcup_{n\in\mathbb{N}} \mathrm{wp}\ t[s_n\cdot\mathsf{id}]\,Q &= \bigcup_{n\in\mathbb{N}} \mathrm{wp}\ t\,(\mathrm{wp}\,(s_n\cdot\mathsf{id})\,Q) \\
&= \bigcup_{n\in\mathbb{N}} \mathrm{wp}\ t\,(\mathrm{wp}\ s_n\,Q\ \cdot\ Q) \\
&= \mathrm{wp}\ t\,((\bigcup_{n\in\mathbb{N}} \mathrm{wp}\ s_n\,Q)\ \cdot\ Q)
\end{aligned}$$

By Lemma 8.34 it suffices to show that

$$\bigcup_{n\in\mathbb{N}} \mathrm{wp}\ s_n\,Q = \mu(\lambda P.\ \mathrm{wp}\ s\,(P\cdot Q))$$

But this is precisely the content of Lemma 8.22 after unfolding the definitions using Lemma 8.37 and noting that $\mathrm{wp}\ \Omega\,Q = \bot$. ■

8.6 Operational Correctness Statements

In this section we connect the operational and axiomatic semantics of IC. In particular, we show that the total correctness judgments of IC characterize the small-step operational semantics (see Theorem 8.46):

$$\langle\sigma\rangle\, s\, \langle Q\rangle \;\leftrightarrow\; \exists \tau f.\, (s,\sigma) \rhd^* (f,\tau) \wedge Q\, f\, \tau$$

The axiomatic semantics of IC is compatible with small-step reduction.

Lemma 8.42 If $(s,\sigma) \rhd (t,\tau)$, then

$$\langle\sigma\rangle\, s\, \langle Q\rangle \;\leftrightarrow\; \langle\tau\rangle\, t\, \langle Q\rangle$$
$$\{\sigma\}\, s\, \{Q\} \;\leftrightarrow\; \{\tau\}\, t\, \{Q\}$$

Proof By case analysis on the step relation using the WP semantics of IC. For local definitions we use Corollary 8.38. All other cases are immediate. ■

Corollary 8.43 If $(s,\sigma) \rhd^* (f,\tau)$, then $\langle\sigma\rangle\, s\, \langle Q\rangle \leftrightarrow Q\, f\, \tau$ and $\{\sigma\}\, s\, \{Q\} \leftrightarrow Q\, f\, \tau$.

In particular, for terminating programs, partial and total correctness coincide.

In order to show that $\langle\sigma\rangle\, s\, \langle Q\rangle$ implies termination of s, we generalize in a way reminiscent of logical relations. According to Lemma 8.37, a post-condition can be read as a semantic substitution. Conversely, if $\langle\sigma\rangle\, s\, \langle Q\rangle$ holds, then s terminates under all substitutions which are compatible with Q.

Lemma 8.44 Assume that $\langle\sigma\rangle\, s\, \langle Q\rangle$ holds and let θ be a substitution such that

$$\forall f \tau.\, Q\, f\, \tau \to (\theta f, \sigma)\Downarrow$$

then $(s[\theta], \sigma)\Downarrow$.

Proof By directed induction. ■

Corollary 8.45 $\langle\sigma\rangle\, s\, \langle Q\rangle \to (s,\sigma)\Downarrow$

We can now show that the axiomatic semantics of IC coincides with the small-step semantics.

Theorem 8.46 $\langle\sigma\rangle\, s\, \langle Q\rangle \;\leftrightarrow\; \exists \tau f.\, (s,\sigma) \rhd^* (f,\tau) \;\wedge\; Q\, f\, \tau$

Proof In the left-to-right direction, we use Corollary 8.45 to obtain (f,τ) such that $(s,\sigma) \rhd^* (f,\tau)$ from $\langle\sigma\rangle\, s\, \langle Q\rangle$. By Corollary 8.43 we obtain $Q\, f\, \tau$. The right-to-left direction is a special case of Corollary 8.43.

8.7 Translation from GC to IC

In this section, we verify a compiler $\mathcal{C}$ from GC to IC. Formally, we translate a GC program into an IC program with at least one free label. The compiler arranges the target program such that a call to the label 0 indicates successful termination. The compiler correctness statement we prove is preservation of total and partial specifications.

$$\begin{aligned} \mathrm{wp}\ s\,Q &\subseteq \mathrm{wp}\,(\mathcal{C}\,s)\,(Q \cdot \bot) \\ \mathrm{wlp}\ s\,Q &\subseteq \mathrm{wlp}\,(\mathcal{C}\,s)\,(Q \cdot \bot) \end{aligned}$$

Where $\bot$ is the empty specification, which implies that $\mathcal{C}\,s$ may only terminate with a jump to label 0. Together with Theorem 8.46, this yields the following correctness statement, which connects the axiomatic semantics of GC to the small-step semantics of IC.

$$\langle\sigma\rangle\, s\,\langle Q\rangle \;\rightarrow\; \exists\tau.\ (\mathcal{C}s, \sigma) \rhd^{*} (0, \tau) \;\wedge\; Q\,\tau$$

The compiler is defined in Figure 8.1.

The main transformations are the sequentialization of guarded command sets and the translation of loops to recursive functions. The compiler exploits underspecification of GC and the fact that the compiled program only needs to be correct for states σ in which the program cannot get stuck.

8.7.1 Sequencing Guarded Command Sets

The compiler realizes guarded command sets with nested IC conditionals.

Example 8.47 (Translation of Conditionals) The program

$$(\textbf{if}\ \ b_1 \Rightarrow a_1 \parallel b_2 \Rightarrow a_2 \parallel b_3 \Rightarrow a_3); s$$

can be translated to

$$\textbf{if}\ b_2\ \textbf{then}\ a_2; \mathcal{C}\ s\ \textbf{else if}\ b_3\ \textbf{then}\ a_3; \mathcal{C}\ s\ \textbf{else}\ a_1; \mathcal{C}\ s$$

Underspecification in GC allows us to test the guards b_2, b_3 in any order. The guard b_1 does not need to be tested, because if the GC programs terminates, then one of the guards b_1, b_2, b_3 is satisfied.

The flattening function $\mathcal{F}$ maps a guarded command set to nested IC conditionals. $\mathcal{F}$ takes two arguments, a guarded command set to translate and an IC program v.

$$\begin{aligned}
\mathcal{C}\,_ &: \mathbb{C} \to \mathbb{T}^{n+1} \\
\mathcal{C}\,s &:= \mathcal{T}\,s\,0
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}\,_\,_ &: \mathbb{C} \to \mathbb{T}^{n} \to \mathbb{T}^{n} \\
\mathcal{T}\,\textbf{skip}\,u &:= u \\
\mathcal{T}\,a\,u &:= a; u \\
\mathcal{T}\,(s;t)\,u &:= \mathcal{T}\,s\,(\mathcal{T}\,t\,u) \\
\mathcal{T}\,(\textbf{if}\ \varnothing)\,u &:= \Omega \\
\mathcal{T}\,(\textbf{if}\ b \Rightarrow s \parallel G)\,u &:= \textbf{let}\ u\ \textbf{in}\ \mathcal{F}\,G\,(\mathcal{C}\,s) \\
\mathcal{T}\,(\textbf{do}\ G)\,u &:= \textbf{def}\,\mathcal{F}\,G\ u[\uparrow]\ \textbf{in}\,0
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}\,_\,_ &: \mathbb{G} \to \mathbb{T}^{n+1} \to \mathbb{T}^{n+1} \\
\mathcal{F}\,\varnothing\,v &:= v \\
\mathcal{F}\,(b \Rightarrow s \parallel G)\,v &:= \textbf{if}\ b\ \textbf{then}\ \mathcal{C}\,s\ \textbf{else}\ \mathcal{F}\,G\,v
\end{aligned}$$

$$\begin{aligned}
\textbf{let}\,_\,\textbf{in}\,_ &: \mathbb{T}^{n} \to \mathbb{T}^{n+1} \to \mathbb{T}^{n} \\
\textbf{let}\,s\,\textbf{in}\,t &:= \begin{cases} \textbf{def}\ s[\uparrow]\ \textbf{in}\,t & \text{if } |s| > 1 \\ t[s \cdot \mathsf{id}] & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 8.1.: Compiler from GC to IC

$\mathcal{F}$ translates each guarded command to the corresponding conditional and places the program v in the final else case. The program $\mathcal{F}\,(b_1 \Rightarrow s_1 \parallel \ldots \parallel b_n \Rightarrow s_n)\, v$ tests the guards b_i in order, and continues execution with the first program s_i for which b_i is satisfied. If no guard b_i is satisfied, execution continues with v.

The compiler uses $\mathcal{F}$ to translate conditionals **if** $b \Rightarrow s \parallel G$ to nested conditionals in IC by placing s in the final else case. The compiler is allowed to translate the empty conditional **if** $\varnothing$ to any program, because the empty conditional is stuck.

8.7.2 Linearizing GC

In addition to sequencing guarded command sets, the compiler translates loops to tail-recursive continuations. We call this translation **linearization**, because full sequentialization $s; t$ as it occurs in GC is translated to the linear sequenzialization $a; s$ as it is available in IC.

Example 8.48 (Translation of Loops) The program

$$(\textbf{do } b_1 \Rightarrow a_1 \parallel b_2 \Rightarrow a_2); s$$

is translated to

$$\textbf{def if } b_1 \textbf{ then } a_1; 0 \textbf{ else if } b_2 \textbf{ then } a_2; 0 \textbf{ else } (\mathcal{C}\, s)[\uparrow] \textbf{ in } 0$$

Underspecification in GC allows the guards b_1, b_2 to be tested in any order. The final else case is reached if none of the guards b_1, b_2 is satisfied. In this case, the loop terminates and execution continues with u.

Linarization is implemented by a function $\mathcal{T}\, s\, u$, which takes a GC program s and an IC program u to be used as continuation. Intuitively, the program $\mathcal{T}\, s\, u$ is the sequentialization of s and u: First s is executed, and then execution continues with u.

8.7.3 Avoiding Exponential Blowup

A direct translation of guarded command sets may cause exponential blowup. Consider the duplication of the continuation s in Example 8.47.

We avoid duplicating the continuation by capturing it in a local definition in the compilation of conditionals. Using a local definition for every continuation works, but can introduce more continuations than necessary. In particular, we do not want a new definition if the continuation is already a call to a continuation. We define an auxiliary function **let**, which introduces definitions only when necessary.

Semantically, **let** behaves like a non-recursive definition.

Lemma 8.49 For all s, t, Q we have

$$\begin{aligned} \mathrm{wp}(\textbf{let}\, s\, \textbf{in}\, t)\, Q &= \mathrm{wp}\, t\, (\mathrm{wp}\ s\ Q\ \cdot\ Q) \\ \mathrm{wlp}(\textbf{let}\, s\, \textbf{in}\, t)\, Q &= \mathrm{wlp}\, t\, (\mathrm{wlp}\ s\ Q\ \cdot\ Q) \end{aligned}$$

Proof If $|s| \le 1$ then $(\textbf{let}\, s\, \textbf{in}\, t) = t[s \cdot \mathsf{id}]$ and the result follows by Corollary 8.38. Otherwise, $\textbf{let}\, s\, \textbf{in}\, t \;=\; \textbf{def}\, s[\uparrow]\, \textbf{in}\, t$ and by Lemma 8.34 and Corollary 8.36 we have

$$\begin{aligned} \mathrm{wp}(\textbf{let}\, s\, \textbf{in}\, t)\, Q &= \mathrm{wp}\, t\, (\mu(\lambda P.\ \mathrm{wp}\, s[\uparrow]\, (P \cdot Q))\ \cdot\ Q) \\ &= \mathrm{wp}\, t\, (\mu(\lambda P.\ \mathrm{wp}\, s\, Q)\ \cdot\ Q) \\ &= \mathrm{wp}\, t\, (\mathrm{wp}\, s\, Q\ \cdot Q) \end{aligned}$$

with the same proof in the case of $\mathrm{wlp}(\textbf{let}\, s\, \textbf{in}\, t)\, Q$. ■

8.7.4 Compiler Correctness

Since the definition of $\mathcal{C}$, $\mathcal{T}$, and $\mathcal{F}$ are mutually recursive, we also show the compiler correctness by mutual induction. The cases for wp and wlp are entirely analogous and so we only illustrate the case for wp.

We have already given the correctness statement for $\mathcal{C}$ in the beginning. The semantics of $\mathcal{T}$ is sequencing a GC program in front of an IC program. The semantics of flattening $\mathcal{F}$ corresponds to wp^*.

Theorem 8.50 For all GC programs s, IC programs u, GC postconditions P and IC postconditions Q we have

$$\begin{aligned} \mathrm{wp}\, s\, P &\subseteq \mathrm{wp}\ (\mathcal{C}\, s)\, (P \cdot Q) \\ \mathrm{wp}\, s\, (\mathrm{wp}\, u\, Q) &\subseteq \mathrm{wp}\ (\mathcal{T}\, s\, t)\, Q \end{aligned}$$

For guarded commands G and IC programs u we have

$$\begin{aligned} \widehat{G} \cap \mathrm{wp}^*\ G\, P &\subseteq \mathrm{wp}\ (\mathcal{F}\, G\, u)\, (P \cdot Q) \\ \neg\widehat{G} \cap \mathrm{wp}\ u\, Q &\subseteq \mathrm{wp}\ (\mathcal{F}\, G\, u)\, Q \end{aligned}$$

Proof The correctness of $\mathcal{C}$ follows from the correctness of $\mathcal{T}$, as $P = \mathrm{wp}\, 0\, (P \cdot Q)$.

We show the correctness of $\mathcal{T}$ and $\mathcal{F}$ by induction on GC programs. The cases for empty programs **skip**, actions a, and sequencing $s; t$ can be shown by equational reasoning. In the case of conditionals and loops it is easier to argue in terms of correctness judgments.

- For the empty program **skip** we have

$$\mathrm{wp}\ \textbf{skip}\,(\mathrm{wp}\ u\,Q) \;=\; \mathrm{wp}\ u\,Q \;=\; \mathrm{wp}\,(\mathcal{T}\,\textbf{skip}\,u)\,Q$$

- For actions a we have

$$\begin{aligned}\mathrm{wp}\ a\,(\mathrm{wp}\ u\,Q) &= a \circ \mathrm{wp}\ u\,Q\\ &= \mathrm{wp}\,(a;u)\,Q\\ &= \mathrm{wp}\,(\mathcal{T}\,a\,u)\,Q\end{aligned}$$

- For sequential composition $s;t$ we have

$$\begin{aligned}\mathrm{wp}\,(s;t)\,(\mathrm{wp}\ u\,Q) &= \mathrm{wp}\ s\,(\mathrm{wp}\ t\,(\mathrm{wp}\ u\,Q))\\ &\subseteq \mathrm{wp}\ s\,(\mathrm{wp}\,(\mathcal{T}\,t\,u)\,Q)\\ &\subseteq \mathrm{wp}\,(\mathcal{T}\,s\,(\mathcal{T}\,t\,u))\,Q\\ &= \mathrm{wp}\,(\mathcal{T}\,(s;t)\,u)\,Q\end{aligned}$$

- For conditionals **if** G, the semantics of GC ensures that G is non-empty, since $\mathrm{wp}(\textbf{if}\ \varnothing)\,Q = \widehat{\varnothing} \cap \ldots = \bot$. Thus we can assume that G is of the form $b \Rightarrow s \parallel G$. By assumption we have $\langle\sigma\rangle\ \textbf{if}\ b \Rightarrow s \parallel G\,\langle \mathrm{wp}\,u\,Q\rangle$ and we have to show that $\langle\sigma\rangle\,\textbf{let}\,u\,\textbf{in}\,\mathcal{F}\,G\,(\mathcal{C}\,s)\,\langle Q\rangle$.

  By Lemma 8.49 it suffices to show that $\langle\sigma\rangle\,\mathcal{F}\,G\,(\mathcal{C}\,s)\,\langle \mathrm{wp}\ u\,Q\ \cdot\ Q\rangle$ holds.

  Our assumption implies that either $b\,\sigma$ or $\widehat{G}\,\sigma$ hold. Assume that $\widehat{G}\,\sigma$, and hence also $\mathrm{wp}^*\ G\,(\mathrm{wp}\,u\,Q)$ hold. The result follows from the correctness of $\mathcal{F}$.

  Otherwise assume that $\widehat{G}\,\sigma$ does not hold but $b\,\sigma$ does. Then by assumption, we have $\langle\sigma\rangle\,s\,\langle \mathrm{wp}\,u\,Q\rangle$ and by the correctness of $\mathcal{C}$ we conclude that $\langle\sigma\rangle\,(\mathcal{C}\,s)\,\langle \mathrm{wp}\,u\,Q\ \cdot\ Q\rangle$ holds. The result follows by the correctness of flattening in the case where no guard is satisfied.

- In the case of loops **do** G, the semantics of the source program gives

$$\mathrm{wp}\,(\textbf{do}\ G)\,(\mathrm{wp}\ u\,Q) \;=\; \mu\left(\lambda P\sigma.\ \text{if}\ \widehat{G}\,\sigma\ \text{then}\ \mathrm{wp}^*\ G\,P\,\sigma\ \text{else}\ \langle\sigma\rangle\,u\,\langle Q\rangle\right)$$

The semantics of the target program is

$$\begin{aligned}
\mathrm{wp}(\mathcal{T}\,(\textbf{do}\ G)\,u)\,Q &= \mathrm{wp}(\textbf{def}\,\mathcal{F}\,G\,u[\uparrow]\,\textbf{in}\,0)\,Q \\
&= \mathrm{wp}\,0\,\mu(\lambda P.\ \mathrm{wp}\ (\mathcal{F}\,G\,u[\uparrow])\,(P\cdot Q))\ \cdot Q \\
&= \mu(\lambda P.\ \mathrm{wp}\ (\mathcal{F}\,G\,u[\uparrow])\,(P\cdot Q))
\end{aligned}$$

And so we need to show an inclusion between two least fixed-points. Since taking fixed-points is a monotone operation, this inclusion holds if the statement

$$\text{if}\ \widehat{G}\,\sigma\ \text{then}\ \mathrm{wp}^*\ G\,P\,\sigma\ \text{else}\ \langle\sigma\rangle\,u\,\langle Q\rangle$$

implies

$$\langle\sigma\rangle\,\mathcal{F}\,G\,u[\uparrow]\,\langle P\cdot Q\rangle$$

for all P, σ.

We proceed by case analysis on $\widehat{G}\,\sigma$.

If $\widehat{G}\,\sigma$ holds we can assume that $\mathrm{wp}^*\ G\,P$ holds. By the correctness of flattening this implies that $\langle\sigma\rangle\,\mathcal{F}\,G\,u[\uparrow]\,\langle P\cdot Q\rangle$ holds, which is what we needed to show.

Otherwise, if $\neg\widehat{G}\,\sigma$ holds, we can assume that $\langle\sigma\rangle\,u\,\langle Q\rangle$ holds and hence by Corollary 8.36 also $\langle\sigma\rangle\,u[\uparrow]\,\langle P\cdot Q\rangle$. The statement again follows by the correctness of flattening.

The correctness of flattening follows by induction on G. If G is empty we have $\neg\widehat{G}\,\sigma$ and the statement follows since $\langle\sigma\rangle\,u\,\langle Q\rangle = \langle\sigma\rangle\,\mathcal{F}\varnothing u\,\langle Q\rangle$.

If G is of the form $b \Rightarrow s \parallel G'$, we translate it as a conditional.

$$\begin{aligned}
\langle\sigma\rangle\,\mathcal{F}\,(b\Rightarrow s\parallel G')\,u\,\langle Q\rangle &= \langle\sigma\rangle\,\textbf{if}\ b\ \textbf{then}\ \mathcal{C}\,s\ \textbf{else}\ \mathcal{F}\,G'\,u\,\langle Q\rangle \\
&= \text{if}\ b\,\sigma\ \text{then}\ \langle\sigma\rangle\,\mathcal{C}\,s\,\langle Q\rangle\ \text{else}\ \langle\sigma\rangle\,\mathcal{F}\,G'\,u\,\langle Q\rangle
\end{aligned}$$

Now assume that $\widehat{G}\,\sigma$ does not hold and that $\langle\sigma\rangle\,u\,\langle Q\rangle$ holds. Then in particular, $b\,\sigma$ does not hold and the statement reduces to $\langle\sigma\rangle\,\mathcal{F}\,G'\,u\,\langle Q\rangle$ which follows by induction.

On the other hand, let us assume that $\widehat{G}\,\sigma$ and $\mathrm{wp}^*\ G\,P$ hold. We proceed by case analysis on $b\,\sigma$. If $b\,\sigma$ holds, we have $\mathrm{wp}\,s\,P$ by assumption and thus $\langle\sigma\rangle\,\mathcal{C}\,s\,\langle P\cdot Q\rangle$ follows from the correctness of compilation. Otherwise, we have $\mathrm{wp}^*\ G'\,P$ and the statement follows by induction. ■

Since the proof of Theorem 8.50 proceeds by induction on programs and then shows the inclusion of fixed-points by monotonicity we can use exactly the same proof with weakest liberal preconditions.

From this, we obtain the following correctness statement for the compiler.

Corollary 8.51

$$\begin{aligned} \mathrm{wp}\, s\, Q &\subseteq \mathrm{wp}\ (\mathcal{C}\, s)\, (Q \cdot \bot) \\ \mathrm{wlp}\, s\, Q &\subseteq \mathrm{wlp}\ (\mathcal{C}\, s)\, (Q \cdot \bot) \end{aligned}$$

8.8 Discussion

There are three contributions in this chapter. The (co-)directed (co-)induction principles, the axiomatic semantics for GC and IC, and the verification of a compiler from GC to IC.

(Co-)directed (co-)induction. We identify the (co)directed (co)induction principle as a general method for proving statements about least and greatest fixed-points in complete lattices. While induction is readily available in type theory, it only applies to inductive definitions which pass a syntactic guardedness check. For GC, the definition of wp in terms of the evaluation functional would not pass this guardedness check. Indeed, this definition is only monotone between monotone predicate transformers, not between arbitrary predicate transformers (consider the definition of $\underline{\mathrm{wp}}\, (s; t)\, Q$).

Furthermore, there is no analogue to induction for coinductive definition. We claim that codirected coinduction can be used as a stand-in for such a native coinduction principle. Most proofs in this chapter are completely symmetrical between their inductive and coinductive variants.

In a complete lattice, inductive and coinductive definitions are dual, yet the simple reasoning principles offered by Tarski's theorem are inadequate for mechanized developments. Tarski's theorem allows us to show statements of the form $\mu f \leq x$ whenever we can show that $f x \leq x$ and dually for νf. One problem with using this principle in a mechanized development is that we need to reformulate the statement we are trying to show in the form of an inequality $\mu f \leq x$. Directed induction does not suffer from this problem. It applies to arbitrary goals of the form $P\, (\mu f)$ for a predicate P, subject to a side-condition on P which is usually easily discharged.

Directed induction in particular has several applications to simple statements which are otherwise difficult to show. For example, in Example 6.13 we introduced the notion of similarity as a greatest fixed point and argued that it is transitive. This follows directly by tower induction, since the set of transitive relations is inf-closed. The set of transitive relations is not sup-closed, but it is closed under directed suprema. By directed induction, the corresponding least fixed point is transitive.

Axiomatic Semantics for GC and IC. The second contribution consists of a novel presentation of axiomatic semantics for two idealized programming languages GC and IC.

We present an axiomatic semantics for Dijkstra's GC language [41] with an evaluation functional. This is in contrast to the presentation in [107], which was based on an inductive definition of the correctness judgments $\langle\sigma\rangle\, s\, \langle Q\rangle$. The main reason for this difference in presentation is that we wanted to handle total and partial correctness in a unified fashion. It is possible to define the partial correctness judgment $\{\sigma\}\, s\, \{Q\}$ coinductively, since the type theory underlying Coq has some support for coinductive definitions. Unfortunately, there is an asymmetry between inductive and coinductive definitions in type theory, which makes coinductive definitions more difficult to handle. In order to obtain a more symmetric definition, we define both total and partial correctness directly using least and greatest fixed-points in a complete lattice.

At this point we are no longer restricted to defining inductive relations. Instead we define weakest (liberal) preconditions as least and greatest fixed-points in the lattice of *monotone* predicate transformers. This way we obtain some properties for free, since every predicate transformer is automatically monotone.

We present this approach in a more abstract setting and show that many desirable properties of weakest (liberal) preconditions can be shown from properties of the evaluation functional. The (co)directed (co)induction principle plays a large role in this abstract developments. For example, we do not know of any more direct proof of Lemma 8.17, even though concrete instances of this theorem may admit a simpler proof.

We show that the definition of partial correctness using the greatest fixed-point of the evaluation functional admits the usual Hoare logic proof rules for partial correctness.

At the same time, the weakest preconditions admit a recursive definition on the syntax of programs. Dijkstra's original semantic specification of GC [41] is such a recursive function computing weakest preconditions. We characterize the weakest precondition of loops using a fixed-point operator, while Dijkstra uses ω-iteration. We show that the predicate transformers computing weakest preconditions for GC and IC programs are continuous. This ensures that the fixed-points that occur in our definitions can be characterized in terms of ω-iteration.

Besides GC, we consider a low-level language IC with a substitution-based small-step semantics. We give an axiomatic semantics for IC and show that it coincides with the small-step semantics. The axiomatic semantics allows us to give particularly

simple characterizations for the semantics of programs under instantiation in IC. We have used this characterization throughout this chapter to show properties of IC.

Compiler Verification. We verify a compiler from GC to IC. We formulate compiler correctness in terms of the axiomatic semantics as preservation of specifications. The axiomatic semantics makes it easy to account for the semantic underspecification of GC. The correctness proof makes use of both the recursive and the inductive characterization of the semantics of GC and IC. Simple cases such as the translation of sequential composition are handled by equational reasoning, while the slightly optimized translation of conditionals is better handled using correctness judgments. Since the proof did not proceed by induction on derivations, but rather by induction on programs with a nested induction in the case of loops we can adapt it without change to show the preservation of total and partial correctness.

In terms of operational semantics, compiler correctness for non-deterministic languages has to establish a backwards simulation of the form

$$\forall \tau.\ \ (\mathcal{C}\, s, \sigma) \Downarrow \tau \ \rightarrow\ (s, \sigma) \Downarrow \tau$$

This is problematic to prove directly [76], since we would have to analyze all possible reductions of the compiled program $\mathcal{C}\, s$. Depending on the complexity of $\mathcal{C}$, this can easily become infeasible.

In contrast to this, proving compiler correctness using axiomatic semantics remains straightforward in the presence of non-determinism.

We show preservation of partial correctness judgments

$$\{\sigma\}\, s\, \{Q\} \ \rightarrow\ \{\sigma\}\, \mathcal{C}\, s\, \{Q\}$$

It is possible to show an operational correctness statement of the form

$$\{\sigma\}\, s\, \{Q\} \ =\ \forall \tau.\ (s, \sigma) \Downarrow \tau \ \rightarrow\ Q\, \tau$$

for partial correctness. Assuming that we have operational semantics for both the source and target languages, this implies

$$(\forall \tau.\ (s, \sigma) \Downarrow \tau \rightarrow Q\, \tau) \ \rightarrow\ \forall \tau.\ (\mathcal{C}\, s, \sigma) \Downarrow \tau \rightarrow Q\, \tau$$

for $Q\, \tau := (s, \sigma) \Downarrow \tau$ the premise is vacuously true and we obtain

$$\forall \tau.\ (\mathcal{C}\, s, \sigma) \Downarrow \tau \ \rightarrow\ (s, \sigma) \Downarrow \tau$$

It follows that preservation of partial specifications implies *backwards* simulation, even though we only had to analyze the possible source programs. Preservation of total specifications supplements this and implies preservation of termination.

9 Autosubst: Automation for de Bruijn Substitutions

In this chapter we describe the design and implementation of the AUTOSUBST library for Coq [106]. AUTOSUBST is available at

https://www.ps.uni-saarland.de/autosubst/

It should be apparent by now that there is a great deal of redundancy when formalizing proofs about syntax with binders. The definition of instantiation and the laws of the σ-calculus are fixed as soon as we know the syntax of terms and the location of binders.

The idea behind AUTOSUBST is that we can encode this information directly in the definition of an inductive type and generate the corresponding definitions for instantiation and proofs of substitution lemmas. Coq has a built-in metaprogramming language called Ltac [40], which is adequate for this purpose.

We also use Ltac to define automation tactics asimpl and autosubst. The tactic asimpl uses the laws of the σ-calculus, in addition to the definitional simplification built-into type theory to normalize all substitution expressions which occur in a goal or context. The tactic autosubst tries to solve substitution lemmas.

While the σ-calculus is complete in theory, there are some practical issues to consider as well. For one, the definitions of the substitution operations are not opaque and have to interact with other user-defined functions. This is why the autosubst tactic sometimes has to perform additional cases analyses, which would be undesirable in asimpl.

For efficiency we also extend the σ-calculus with n-ary shift operations $(+n)$, which creates a number of critical pairs in the rewriting system of the σ-calculus which need to be handled. However, the process of completing the extended rewriting system is entirely mechanical.

Due to inherent restrictions in Ltac at the time of this writing, we only provide support for pure de Bruijn terms, not for well-scoped syntax. Additionally, we provide a different implementation of heterogeneous substitutions which occur in multi-sorted languages such as System F. Instead, of using a single vector substitution operation as presented in Chapter 5 we provide different type and term substitutions and commutativity laws between them.

These restrictions are mostly resolved in AUTOSUBST 2 [116], which at the time of this writing is still work in progress. We touch upon the differences between again briefly at the end of this chapter.

Since this chapter deals with implementation details specific to the Coq proof assistant, we use the concrete syntax of Coq in this chapter.

The content of this chapter is based on [106].

9.1 Automating de Bruijn Substitutions in Coq

We describe the implementation of AUTOSUBST on the example of the untyped λ-calculus.

In Coq we define the pure de Bruijn representation of untyped λ-terms as an inductive type with variable and binding annotations.

$$
\begin{aligned}
&\textbf{Inductive}\ \mathsf{tm}\ :\ \textbf{Type} := \\
&|\ \mathsf{Var}\ :\ (x\ :\ \mathsf{var}) \\
&|\ \mathsf{App}\ :\ (s\,t\ :\ \mathsf{tm}) \\
&|\ \mathsf{Lam}\ :\ (s\ :\ \{\mathsf{bind}\,\mathsf{tm}\}).
\end{aligned}
$$

Every term type must contain exactly one constructor for de Bruijn indices, i.e., a constructor which takes a single argument of type var. The type var is an alias for $\mathbb{N}$.

Additionally, all binders must be marked. In this example, Lam is the only binder. It introduces a new index into the scope of its argument s. The annotation $\{\mathsf{bind}\,\mathsf{tm}\}$ is definitionally equal to tm.

Substitutions are represented as functions $\sigma, \tau : \mathbb{N} \to \mathrm{tm}$, with the cons operation defined generically for functions of type $\mathbb{N} \to X$ for every type X. We reuse the notation $s \cdot \sigma$ to refer to this cons operation in this section. Similarly, we write ids for the identity substitution, which is the constructor Var in the case of λ-terms. By post-composing with the identity substitution, we can lift arbitrary renamings (functions $\xi : \mathbb{N} \to \mathbb{N}$) to substitutions. In Coq we write $f \ggg g$ for forward composition of functions. For readability, we also introduce notation for the coercion of renamings into substitutions, $\mathsf{ren}\,\xi := \xi \ggg \mathsf{ids}$. The shift renaming $\uparrow$ is written $(+1)$.

From the definition of the inductive type with annotations, AUTOSUBST generates

the following definition for instantiation under renamings.

```
Fixpoint rename (ξ : ℕ → ℕ) (s : tm) : tm :=
  match s with
    | Var x  ⇒  ids (ξ x)
    | App s t  ⇒  App (rename ξ s) (rename ξ t)
    | Lam s  ⇒  Lam (rename (0 · ξ >>> (+1)) s)
  end
```

AUTOSUBST uses rename to define instantiation. In particular, using rename we define $\mathsf{up}\,\sigma := \mathrm{ids}\,0 \cdot \sigma >>> \mathsf{rename}(+1)$. Using up, we define the full instantiation operation on terms.

```
Fixpoint inst (σ : ℕ → tm) (s : tm) : tm :=
  match s with
    | Var x  ⇒  σ x
    | App s t  ⇒  App (inst σ s) (inst σ t)
    | Lam s  ⇒  Lam (inst (up σ) s)
  end
```

With instantiation, we define substitution composition $\sigma \gg \tau := \sigma >>> \mathsf{inst}\,\tau$. We write $s[\sigma]$ for $\mathsf{inst}\,\sigma\,s$.

To complete the definition we need to show that these definitions satisfy the laws of the σ-calculus. The only definitions that are specific to the term language at hand are the identity substitution, and the definition of instantiation. In order to show that our definitions yield a model of the σ-calculus, we only need to know that renaming is a special case of instantiation and that instantiation and the identity substitution behave correctly. For this, it suffices to show

$$\begin{aligned} \mathsf{rename}\,\xi\,s &= s[\mathsf{ren}\,\xi] \\ (\mathrm{ids}\,x)[\sigma] &= \sigma\,x \\ s[\mathrm{ids}] &= s \\ s[\sigma][\tau] &= s[\sigma \gg \tau] \end{aligned}$$

It is easy to check that given these four equations all the other equations in the σ-calculus follow without any other assumptions about ids or instantiation.

In the case of the untyped λ-calculus, the first equation $(\text{ids}\,x)[\sigma] = \sigma\,x$ holds by definition, while the second follows by a straightforward term induction. The third equation is more interesting, since the proof has to follow the inductive structure of the instantiation operation, as described in Chapter 3.

Formally, the proof proceeds in three steps. First we show $(\mathsf{rename}\,\xi\,s)[\tau] = s[\xi \ggg \tau]$, by induction on s. Using this result we can show that $(\mathsf{rename}\,\xi\,s[\sigma]) = s[\sigma \ggg \mathsf{rename}\,\xi]$, again by induction on s. We finally show the full equation, with another term induction. This proof boils down to showing that $\Uparrow\sigma \gg \Uparrow\tau = \Uparrow(\sigma \gg \tau)$, which depends on both specializations.

Realization in Autosubst In type theory there is no way to distinguish between var and $\mathbb{N}$, or between $\{\mathsf{bind}\,\mathsf{tm}\}$ and tm. In Coq, we use the metaprogramming facilities of Ltac [40] to detect annotations and guide code generation.

The Ltac language is designed for proof automation and works on a *goal*, which is a type in a context. Ltac provides operations, called tactics, for building recursive definitions (fix), case analysis (destruct), as well as a powerful matching facility based on higher-order unification. An Ltac match can inspect the types in the current goal and syntactically match it against a *pattern*, i.e., an open type.

We use this facility to locate the variable constructor and all binding arguments.

Formally, the notation $\{\mathsf{bind}\,\mathsf{tm}\}$ expands to an application $_\mathsf{bind}\,\mathsf{tm}\,\mathsf{tm}\,1$ where $_\mathsf{bind}$ is a constant function defined as follows.

$$\textbf{Definition}\ _\mathsf{bind}\ (T_1\ :\ \textbf{Type})\ (T_2\ :\ \textbf{Type})\ (n\ :\ \mathbb{N})\ :=\ T_2.$$

The annotation $_\mathsf{bind}\,T_1\,T_2\,n$ means that we bind n arguments of type T_1 in a term of type T_2.

Using Ltac, this is enough information to generate renaming and instantiation operations. Consider the definition of renaming. Renaming is a recursive definition of type $(\mathbb{N} \to \mathsf{tm}) \to \mathsf{tm} \to \mathsf{tm}$. We can generate this definition in Ltac by first generating a recursive definition, introducing the two arguments and proceeding by case analysis on the second argument. Notice that the definition of renaming and instantiation are both homomorphic in the term structure. Thus we can handle each case independently. In the case of a constructor C, we apply the same constructor again, generating subgoals for each argument of the constructor.

Unfortunately it is not possible to directly remember the current constructor with Ltac's primitive case analysis tactic. The case analysis tactic will add additional assumptions to the context, but will not automatically report the form of the argument after case analysis. Our workaround is to use another annotation on the goal. We

first transform the goal from the form tm into the definitionally equal form $K\,\mathsf{tm}\,s$, where s is the name of the argument on which we will perform the case analysis and K is the constant function $K\,a\,b := a$. We then perform case analysis and match the resulting goal against the pattern $K\,_\,?\mathsf{s}$ to recover the shape of the current constructor.

After applying the constructor we obtain subgoals for each argument position. AUTOSUBST distinguishes five cases.

- The subgoal is of type var, in which case we apply the renaming.
- The subgoal is of some constant type $T \neq \mathsf{tm}$. This marks a constant argument to the constructor and we leave it unchanged.
- The subgoal is of type tm. This is a recursive position without a binder, so we insert a recursive call to the renaming operation.
- The subgoal is of the form $_\mathsf{bind}\,T_2\,\mathsf{tm}\,n$. In this case we are binding n additional terms of type T_2 in the argument position using the appropriate up operation on renamings. For now we consider only the case where $T_2 = \mathsf{tm}$, since we need additional structure to deal with the general case.
- The subgoal is a "container" with elements of type tm. For example, the subgoal could be of type $\mathsf{list}\,\mathsf{tm}$, $\mathsf{tm} + \mathsf{tm}$, or $\mathbb{N} \to \mathsf{tm}$. In this case we have to recurse through the container and apply the renaming at each position. Again, we need additional structure to handle this case, which we introduce in Chapter 9.2.

With this we can build the definitions of renaming and instantiation in Coq.

We use Coq's type class mechanism [114] to obtain common names for operations and lemmas. We use operational type classes [115] to obtain common notations for the identity substitution and the instantiation operation.

The substitution lemmas are proven using Ltac's usual proof automation techniques, following the structure of the proof from Chapter 3.

At this point we can solve substitution lemmas by rewriting.

Internally, AUTOSUBST differs from a straightforward implementation of the σ-calculus in several ways. The σ-calculus contains a number of equations which are specific to the untyped λ-calculus. If we wanted to use the same approach for a new term language, we would need to extend the rewriting system with equations specific to the new term language. Instead, we construct a definition of instantiation with the appropriate simplification behavior. The automation tactics use a combination of rewriting and term simplification.

There are two complications that arise when trying to obtain appropriate simplification behavior.

Internally, instantiation is defined in terms of renaming. We hide this fact by making the definition of up opaque. The automation tactics work by unfolding up in terms of instantiation instead of renaming (as $\mathsf{up}\,\sigma = \mathrm{ids}\,0 \cdot \sigma \gg \mathsf{ren}(+1)$). This effectively hides the renaming operation.

The second point is a technical problem with the notation for instantiation. Using operational type classes, we bind the notation for instantiation to a function Inst. The name Inst does not refer to the actual definition of instantiation, instead it is merely an annotation. The type class mechanism will replace the notation $s[\sigma]$ with an application $\mathsf{Inst}\,\mathsf{inst}\,\sigma\,s$, which is definitionally equal to $\mathsf{inst}\,\sigma\,s$. When unfolding the definition of instantiation in $(\mathsf{App}\,s\,t)[\sigma]$ we obtain the term $\mathsf{App}\,(\mathsf{inst}\,\sigma\,s)\,(\mathsf{inst}\,\sigma\,t)$, loosing the notation and making it impossible to apply further rewriting rules with lemmas that refer to the notation. Instead, we replace the recursive calls to inst by annotated calls to Inst inst during code generation. Fortunately, Coq's termination checker is clever enough to resolve this indirect recursive call.

For the λ-calculus, AUTOSUBST ends up generating the following definition for instantiation.

$$
\begin{aligned}
&\textbf{Fixpoint}\ \mathsf{inst}\ (\sigma\ :\ \mathbb{N} \to \mathsf{tm})\ (s\ :\ \mathsf{tm})\ :\ \mathsf{tm} :=\\
&\quad \textbf{match}\ s\ \textbf{with}\\
&\quad\quad |\ \mathsf{Var}\,x\ \Rightarrow\ \sigma\,x\\
&\quad\quad |\ \mathsf{App}\,s\,t\ \Rightarrow\ \mathsf{App}\,(\mathsf{Inst}\,\mathsf{inst}\,\sigma\,s)\,(\mathsf{Inst}\,\mathsf{inst}\,\sigma\,t)\\
&\quad\quad |\ \mathsf{Lam}\,s\ \Rightarrow\ \mathsf{Lam}\,(\mathsf{Inst}\,\mathsf{inst}\,(\mathsf{up}\,\sigma)\,s)\\
&\quad \mathsf{end}
\end{aligned}
$$

9.2 Traversable Containers

We use a type class to register types as containers. For example, we can consider the type $\mathsf{list}\,X$ as a container with elements of type X. We can operate on the elements of a container using a function map.

$$\mathsf{map}\ :\ (A \to B) \to \mathsf{list}\,A \to \mathsf{list}\,B$$

The function map satisfies the usual categorical laws of a functor: compatibility with identities $\mathsf{map}\,\mathsf{id} = \mathsf{id}$ and composition $\mathsf{map}\,g \circ \mathsf{map}\,f = \mathsf{map}(g \circ f)$. These laws suffice to generate a well-behaved instantiation operation. Compatibility with

identities implies the identity substitution law. Compatibility with composition implies associativity of substitution composition.

However, sometimes we are dealing with types which are not functors in the categorical sense, yet still contain terms. This is why internally we use a typeclass $\mathsf{MMap}\, X\, Y$[1], which expresses that the type Y contains elements of type X. In Coq we define

$$\textbf{Class}\ \mathsf{MMap}\ (X\, Y\ :\ \textbf{Type})\ :=\ \mathsf{mmap}\ :\ (X \to X) \to Y \to Y.$$

$$\begin{aligned}
&\textbf{Class}\ \mathsf{MMapLemmas}\ (X\, Y\ :\ \textbf{Type})\ `\{\mathsf{MMap}\ X\, Y\}\ :=\ \{\\
&\quad \mathsf{mmap_id}\, x\ :\ \mathsf{mmap}\,\mathsf{id}\, x\ =\ x;\\
&\quad \mathsf{mmap_comp}\, f\, g\, x\ :\ \mathsf{mmap}\, f\, (\mathsf{mmap}\, g\, x)\ =\ \mathsf{mmap}\, (g \ggg f)\, x\\
&\}.
\end{aligned}$$

It is important to use an operational type class for mmap, even though we do not care about binding a notation to mmap. The generated code for renaming and instantiation will contain recursive calls in the argument of mmap. In order for Coq's termination checker to accept these calls it has to inline the definition of an instance of mmap and find the recursive call in a non-recursive position. This also means that the actual definition of mmap needs to be written so as not to confuse the termination checker[2].

This is why we provide another tactic for deriving the definition of mmap.

9.3 Heterogeneous Substitutions

In order for the notations and setup from the previous sections to work, we need all instantiation operations to have a type of the following form.

$$\mathsf{inst}\ :\ (\mathbb{N} \to X) \to X \to X$$

This means that we cannot introduce vector parallel substitutions for multi-sorted language like we did in Chapter 5.

Instead we will introduce a new operation hinst for heterogeneous instantiation of the following type.

$$\mathsf{hinst}\ :\ (\mathbb{N} \to X) \to Y \to Y$$

[1] MMap stands for "monomorphic map".

[2] Essentially, we have to introduce the function argument before building the fixed-point for the recursive definition.

The idea is that hinst will instantiate terms of type X which occur in terms of type Y. We write $s[\![\theta]\!]$ for heterogeneous instantiation and $\sigma \bullet \theta$ for heterogeneous composition.

As before, we shall introduce the necessary machinery with a concrete example. In this section we consider a two-sorted presentation of System F.

$$A, B ::= X \mid A \to B \mid \forall A$$
$$s, t ::= x \mid s\,t \mid \lambda A s \mid \Lambda s \mid s\,A$$

Instantiating types in types works as before. Term instantiation is complicated by the fact that terms contain type binders.

$$x[\sigma] = \sigma(x) \qquad (s\,A)[\sigma] = s[\sigma]\,A$$
$$(s\,t)[\sigma] = s[\sigma]\,t[\sigma] \qquad (\Lambda s)[\sigma] = \Lambda s[\sigma \bullet \uparrow]$$
$$(\lambda A s)[\sigma] = \lambda A s[\Uparrow \sigma]$$

Upon traversing a type binder, we need to increment all type variables in σ. In order to substitute types in terms, we introduce heterogeneous instantiation and composition operations. To avoid confusion with term substitutions, we will write θ, θ' for type substitutions.

$$x[\![\theta]\!] = x \qquad (s\,A)[\![\theta]\!] = s[\![\theta]\!]\,A[\theta]$$
$$(s\,t)[\![\theta]\!] = s[\![\theta]\!]\,t[\![\theta]\!] \qquad (\Lambda s)[\![\theta]\!] = \Lambda s[\![\Uparrow \theta]\!]$$
$$(\lambda A s)[\![\theta]\!] = \lambda A[\theta] s[\![\theta]\!] \qquad (\sigma \bullet \theta)\,x = (\sigma\,x)[\![\theta]\!]$$

We need a number of lemmas about heterogeneous instantiation in order to work with it using equational reasoning. The situation is analogous to the case of ordinary instantiation, except that heterogeneous substitutions have to be invariant in the image of the identity substitution on terms.

$$(\mathrm{ids}\,x)[\![\theta]\!] = \mathrm{ids}\,x$$
$$s[\![\mathrm{ids}]\!] = s$$
$$s[\![\theta]\!][\![\theta]\!]' = s[\![\theta \circ \theta']\!]$$

Note that the identity substitution in the first equation is the identity substitution on terms, while in the second equation, it refers to the identity substitution on types.

Furthermore, in order to show the substitution lemmas for terms, we need to know something about the interaction between the two kinds of substitutions. The fact to take care of is that terms may contain types, but types do not contain terms. Thus, we can push a type substitution under a term substitution, so long as we take care

that the type substitution was applied in the image of the term substitution.

$$s[\sigma][\![\theta]\!] = s[\![\theta]\!][\sigma \bullet \theta]$$

This approach to instantiation for multi-sorted theories is less general and uniform than the vector parallel substitutions in Chapter 5. However, it has the advantage that it can be implemented using exactly the same technique that we have already described for deriving instantiation operations.

9.4 Automation Tactics

The automation tactics work by unfolding definitions and rewriting substitution expressions to an internal normal form. In the case of asimpl, we then fold the expressions back into a more readable format.

We start by unfolding up in terms of instantiation as described previously, as well as unfolding substitution composition in term of function composition, i.e. $\sigma \gg \tau$ becomes $\sigma \ggg \mathsf{inst}\,\tau$. We also unfold the notation $\mathsf{ren}\,\xi$ as $\xi \ggg \mathrm{ids}$. This makes it simpler to handle heterogeneous substitutions, since it reduces the identity and associativity laws into the corresponding laws for function composition (which holds definitionally) and simpler rules for (heterogeneous) instantiation. For example, we have

$$\begin{aligned}
\mathsf{ren}\,(+1) \gg \mathsf{up}\,\sigma &= ((+1) \ggg \mathrm{ids}) \ggg \mathsf{inst}\,(\mathrm{ids}\,0 \cdot \sigma \ggg \mathsf{inst}\,((+1) \ggg \mathrm{ids}) \\
&= (+1) \ggg \mathrm{ids} \ggg \mathsf{inst}\,(\mathrm{ids}\,0 \cdot \sigma \ggg \mathsf{inst}\,((+1) \ggg \mathrm{ids}) \\
&= (+1) \ggg (\mathrm{ids}\,0 \cdot \sigma \ggg \mathsf{inst}\,((+1) \ggg \mathrm{ids})) \\
&= \sigma \ggg \mathsf{inst}\,((+1) \ggg \mathrm{ids})
\end{aligned}$$

Since $\mathrm{ids} \ggg \mathsf{inst}\,\sigma = \sigma$. Apart from this equation and the unfolding, all other laws used in this derivation hold definitionally.

Finally, we fold the expression $\sigma \ggg \mathsf{inst}\,((+1) \ggg \mathrm{ids})$ back to $\sigma \gg \mathsf{ren}\,(+1)$.

9.5 Related Work

There are a number of libraries and tools supporting proofs about syntax. We mention the ones that are available for Coq. What sets AUTOSUBST apart from the related work is our usage of parallel substitutions. This allows us to offer an automation tactic for solving substitution lemmas and simplifying terms involving substitutions based on a clearly defined equational theory. Additionally, AUTOSUBST is completely

implemented within Coq and does not require external tools.

- CFGV [15] uses a generic type of context free grammars with variable binding. Custom term types are obtained by instantiating the generic construction. The library provides a number of general lemmas to work with CFGVs. The representation is named and explicitly deals with issues of α-equivalence.
- DBGen [96] is an external tool that generates de Bruijn substitution operations and proofs of corresponding lemmas. It includes support for syntax defined by mutual recursion. It generates a single-variable substitution operation.
- DBLib [97] is a Coq library for de Bruijn substitutions. It offers a generic type class interface and support for defining single-variable substitution operations in a uniform way. The substitution lemmas are solved by generic tactics. It offers tactics that nicely unfold the substitution operations. Also, DBLib offers some automation in the form of a hint database with frequently needed lemmas.
- GMeta [75] uses a generic term type and automatically generated isomorphisms between it and custom term types. It supports a de Bruijn and a locally nameless interface. Moreover, it has support for mutually recursive syntax and the corresponding heterogeneous substitutions. It supports single-variable substitutions, but lacks automation for substitution lemmas.
- Lambda Tamer [31] is a small library of general purpose automation tactics. It provides support for working with higher-order and dependently-typed abstract syntax.
- LNGen [16] is a generator for single-variable locally-nameless substitution operations and the corresponding lemmas for Coq. It is based on the specification syntax of the Ott tool [110].
- Needle & Knot [68] generates code for unscoped, single-point pure de Bruijn substitutions based on the Knot specification language. The distinguishing feature of Needle & Knot is its support for type systems and well-scopedness predicates.

9.6 Discussion

We have described the implementation of AUTOSUBST [106]. Apart from the support for heterogeneous instantiation and the lack of well-scoped syntax, AUTOSUBST offers

a faithful implementation of the strategies for formalizing syntax with binders that we present in this thesis.

There are some restrictions and design decisions which were forced upon us in order to implement AUTOSUBST completely in Coq using Ltac. For example, Ltac does not handle mutually recursive definitions, thus AUTOSUBST cannot generate code for mutually recursive instantiation operations. There are examples, such as the call-by-value presentation of System F in Chapter 5, or Levy's Call-by-push-value calculus [77], which feature mutually inductive types of terms and values. In particular, in CBPV values are bound in terms, while values contain variables and terms but no binders of their own. This example is beyond the simple code generation in AUTOSUBST and requires a global analysis of the dependencies between term sorts [65].

Since the implementation of AUTOSUBST, we have discovered vector parallel substitutions [116] as a unifying framework for instantiation in multi-sorted languages. There are languages which do not fit into the framework for heterogeneous instantiation as implemented in AUTOSUBST. For example, the $\lambda\mu\tilde{\mu}$-calculus of Curien and Herbelin [37] contains mutually inductive sorts of terms and stacks with binders for both. The corresponding heterogeneous instantiation operations do not satisfy a commutation law, since terms contain stacks and stacks contain terms. Vector parallel substitutions do not suffer from these problems.

The implementation using Ltac also does not permit us to implement useful error reporting, since our code generation only has a local view of single constructors.

We are currently implementing all of these improvements in AUTOSUBST 2 [116].

Nevertheless, using Ltac for the implementation of AUTOSUBST also had advantages. Since AUTOSUBST is completely implemented in Coq it could be distributed as a library with no external dependencies. This helped with adoption, since it is easy to integrate AUTOSUBST into existing developments.

Containers The class MMap is an example of a Haskell "lens" [48]. In the context of Haskell lenses, MMap is usually called Setter and mmap is called over. The intended laws in both cases are the same.

Formalizations using AUTOSUBST AUTOSUBST has been successfully used in numerous formalizations. We have already presented several case studies in this thesis, all of which are also available for AUTOSUBST with their original publications.

We give a short selection of other formalizations implemented using AUTOSUBST or AUTOSUBST 2.

- Forster et al. [47] present a formalization of the metatheory of Call-by-push-value.

- Kaiser et al. [67] formally verify the correspondence between the two sorted presentation of System F and its presentation as a pure type system.
- Mizuno and Sumii [84] formally verify the correspondence between call-by-need and call-by-name. Their development also includes a proof of the standardization theorem for λ-calculus.
- Pottier [98] presents a machine checked development of the CPS translation for the pure λ-calculus with a let construct.
- Timany et al. [123] present a logical relations model of a higher-order functional programming language with impredicative and imperative features and show that scoped effectful computations are observationally pure.
- Timany and Birkedal [122] build a tool for interactive mechanized relational verification of programs written in a concurrent higher-order imperative programming language with continuations.
- Wand et al. [128] present a complete reasoning principle for contextual equivalence in an untyped probabilistic programming language.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. „Explicit Substitutions“. In: **Journal of Functional Programming** 1.04 (Oct. 1991), pp. 375–416 (cit. on pp. 1, 4, 5, 10, 17, 20, 25).

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. „Containers: Constructing Strictly Positive Types“. In: **Theoretical Computer Science** 342.1 (Sept. 2005), pp. 3–27 (cit. on pp. 14, 88).

[3] Andreas Abel. **Strong Normalization for Lambda-Calculus**. Lecture given for the Logic and Computation Group, McGill University, Montreal, Canada. Dec. 2012 (cit. on p. 62).

[4] Andreas Abel. „Typed Applicative Structures and Normalization By Evaluation for System Fω“. In: **Lecture Notes in Computer Science** (2009), pp. 40–54 (cit. on p. 61).

[5] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. „Copatterns: programming infinite structures by observations“. In: **ACM SIGPLAN Notices**. Vol. 48. 1. ACM. 2013, pp. 27–38 (cit. on p. 4).

[6] Robin Adams. „Formalized Metatheory With Terms Represented By an Indexed Family of Types“. In: **Types for Proofs and Programs** (2006), pp. 1–16 (cit. on p. 25).

[7] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. „Type-and-scope safe programs and their proofs“. In: **Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs**. ACM. 2017, pp. 195–207 (cit. on p. 45).

[8] Thorsten Altenkirch. „Constructions, inductive types and strong normalization“. PhD thesis. University of Edinburgh, 1993 (cit. on p. 55).

[9] Thorsten Altenkirch and Ambrus Kaposi. „Type Theory in Type Theory Using Quotient Inductive Types“. In: **ACM SIGPLAN Notices** 51.1 (Jan. 2016), pp. 18–29 (cit. on pp. 42, 166).

[10] Thorsten Altenkirch and Bernhard Reus. „Monadic Presentations of Lambda Terms Using Generalized Inductive Types“. In: **Lecture Notes in Computer Science** (1999), pp. 453–468 (cit. on pp. 25, 165).

[11] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor Mcbride, and Peter Morris. „Indexed Containers“. In: **Journal of Functional Programming** 25 (2015) (cit. on pp. 14, 82).

[12] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. „Observational Equality, Now!“ In: **PLPV** 7 (2007), pp. 57–68 (cit. on p. 15).

[13] Thosten Altenkirch, James Chapman, and Tarmo Uustalu. „Monads Need Not Be Endofunctors“. In: **Logical Methods in Computer Science** 11.1 (Mar. 2015) (cit. on pp. 25, 166).

[14] Nada Amin and Ross Tate. „Java and Scala's Type Systems Are Unsound: the Existential Crisis of Null Pointers“. In: **ACM SIGPLAN Notices** 51.10 (Oct. 2016), pp. 838–848 (cit. on p. 3).

[15] Abhishek Anand and Vincent Rahli. „A Generic Approach to Proofs about Substitution“. In: **Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice**. ACM. 2014, p. 5 (cit. on p. 144).

[16] Brian E. Aydemir and Stephanie Weirich. **LNgen: Tool support for locally nameless representations**. Tech. rep. University of Pennsylvania, 2010 (cit. on p. 144).

[17] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, et al. „Mechanized Metatheory for the Masses: the Poplmark Challenge“. In: **Theorem Proving in Higher Order Logics** (2005), pp. 50–65 (cit. on pp. 25, 163, 165).

[18] Henk Barendregt. „The Lambda Calculus: Its Syntax and Semantics“. In: **Studies in Logic and the Foundations of Mathematics** (1984) (cit. on pp. 4, 21, 161).

[19] Henk Barendregt, Wil Dekkers, and Richard Statman. **Lambda calculus with types**. Cambridge University Press, 2013 (cit. on p. 61).

[20] Henning Basold, Damien Pous, and Jurriaan Rot. „Monoidal Company for Accessible Functors“. In: **7th Conference on Algebra and Coalgebra in Computer Science**. 2017 (cit. on p. 87).

[21] Andrej Bauer and Peter Lefanu Lumsdaine. „On the Bourbaki–Witt principle in toposes“. In: **Mathematical Proceedings of the Cambridge Philosophical Society**. Vol. 155. 1. Cambridge University Press. 2013, pp. 87–99 (cit. on p. 86).

[22] Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. „Strongly Typed Term Representations in Coq“. In: **Journal of automated reasoning** 49.2 (2012), pp. 141–159 (cit. on p. 165).

[23] Stefano Berardi. „Type Dependence and Constructive Mathematics“. PhD thesis. Universita di Torino, 1990 (cit. on p. 14).

[24] Jean-Philippe Bernardy and Moulin Guilhem. „Type-Theory in Color“. In: **ACM SIGPLAN Notices** 48.9 (2013), pp. 61–72 (cit. on p. 160).

[25] Richard S. Bird and Ross Paterson. „De Bruijn Notation As a Nested Datatype“. In: **Journal of Functional Programming** 9.1 (Jan. 1999), pp. 77–91 (cit. on pp. 4, 25, 164).

[26] Garrett Birkhoff. **Lattice theory**. American Mathematical Soc., 1940 (cit. on p. 65).

[27] Errett Bishop and Douglas Bridges. **Constructive analysis**. Springer-Verlag, 1985 (cit. on p. 86).

[28] Martin Bodin, Arthur Charguerand, Daniele Filaretti, et al. „A Trusted Mechanised Javasript Specification“. In: **Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14** (2014) (cit. on p. 3).

[29] Nicolas Bourbaki. „Sur Le théorème De Zorn“. In: **Archiv der Mathematik** 2.6 (Nov. 1949), pp. 434–437 (cit. on p. 86).

[30] N.G de Bruijn. „Lambda Calculus Notation With Nameless Dummies, a Tool for Automatic Formula Manipulation, With Application To the Church-Rosser Theorem“. In: **Indagationes Mathematicae (Proceedings)** 75.5 (1972), pp. 381–392 (cit. on pp. 1, 25, 161, 163).

[31] Adam Chlipala. „Parametric higher-order abstract syntax for mechanized semantics“. In: **ACM Sigplan Notices**. Vol. 43. 9. ACM. 2008, pp. 143–156 (cit. on pp. 144, 160).

[32] Alonzo Church. „A Set of Postulates For the Foundation of Logic“. In: **The Annals of Mathematics** 34.4 (Oct. 1933), p. 839 (cit. on pp. 1, 17, 161).

[33] Alonzo Church and J. B. Rosser. „Some Properties of Conversion“. In: **Transactions of the American Mathematical Society** 39.3 (Mar. 1936), pp. 472–472 (cit. on p. 30).

[34] Thierry Coquand and Christine Paulin. „Inductively defined types“. In: **Proceedings of the International Conference on Computer Logic**. Springer-Verlag. 1988, pp. 50–66 (cit. on p. 3).

[35] Karl Crary. „Logical Relations and a Case Study in Equivalence Checking“. In: **Advanced topics in types and programming languages**. Ed. by Benjamin C Pierce. MIT Press, 2005. Chap. 6 (cit. on p. 61).

[36] P. L. Curien, T. Hardin, and A. Ríos. „Strong Normalization of Substitutions“. In: **Lecture Notes in Computer Science** (1992), pp. 209–217 (cit. on p. 21).

[37] Pierre-Louis Curien and Hugo Herbelin. „The duality of computation“. In: **ACM sigplan notices**. Vol. 35. 9. ACM. 2000, pp. 233–243 (cit. on p. 145).

[38] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. „Confluence Properties of Weak and Strong Calculi of Explicit Substitutions“. In: **Journal of the ACM** 43.2 (Mar. 1996), pp. 362–397 (cit. on pp. 17, 21).

[39] Nils Anders Danielsson. „Up-To Techniques Using Sized Types“. In: **Proceedings of the ACM on Programming Languages** 2.POPL (Dec. 2017), pp. 1–28 (cit. on p. 87).

[40] David Delahaye. „A Tactic Language for the System Coq“. In: **Lecture Notes in Artificial Intelligence** (), pp. 85–95 (cit. on pp. 135, 138).

[41] Edsger W. Dijkstra. „Guarded Commands, Nondeterminacy and Formal Derivation of Programs“. In: **Commun. ACM** 18.8 (1975), pp. 453–457 (cit. on pp. 1, 101, 112, 132).

[42] Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. **Semantics of Type Systems Lecture Notes**. 2018 (cit. on pp. 45, 60).

[43] Sophia Drossopoulou and Susan Eisenbach. „Java is type safe — probably“. In: **European Conference on Object-Oriented Programming**. Springer. 1997, pp. 389–418 (cit. on p. 3).

[44] Peter Dybjer. „Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics“. In: **Logical frameworks** 2 (1991), pp. 280–306 (cit. on p. 3).

[45] Amy Felty and Alberto Momigliano. „Hybrid“. In: **Journal of Automated Reasoning** 48.1 (2012), pp. 43–105 (cit. on p. 158).

[46] M. Fiore, G. Plotkin, and D. Turi. „Abstract Syntax and Variable Binding“. In: **Proceedings. 14th Symposium on Logic in Computer Science**. July 1999, pp. 193–202 (cit. on p. 25).

[47] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. „Call-by-push-value in Coq: Operational, Equational, and Denotational Theory“. In: **Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs**. CPP 2019. Cascais, Portugal: ACM, 2019, pp. 118–131 (cit. on pp. 7, 62, 145).

[48] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. „Combinators for Bidirectional Tree Transformations: a Linguistic Approach To the View-Update Problem“. In: **ACM Transactions on Programming Languages and Systems (TOPLAS)** 29.3 (2007), p. 17 (cit. on p. 145).

[49] Andrew Gacek. „The Abella interactive theorem prover (system description)“. In: **International Joint Conference on Automated Reasoning**. Springer. 2008, pp. 154–161 (cit. on p. 158).

[50] Gerhard Gentzen. „Untersuchungen über Das Logische Schließen. I“. In: **Mathematische Zeitschrift** 39.1 (Dec. 1935), pp. 176–210 (cit. on pp. 1, 24).

[51] Jean-Yves Girard. „Une extension de l'interpretation de Godel a l'analyse et son application a l'elimination des coupures dans l'analyse et la theorie des types“. In: **Proc. 2nd Scandinavian Logic Symp.** North-Holland. 1971, pp. 63–92 (cit. on pp. 1, 2, 45, 60).

[52] Jean-Yves Girard, Paul Taylor, and Yves Lafont. **Proofs and types**. Cambridge University Press Cambridge, 1989 (cit. on pp. 45, 60, 61, 165).

[53] Kurt Gödel. „Über Eine Bisher Not Nicht benützte Erweiterung Des Finiten Standpunktes“. In: **Dialectica** 12.3-4 (Dec. 1958), pp. 280–287 (cit. on p. 60).

[54] Healfdene Goguen and James McKinna. „Candidates for Substitution“. In: **LFCS report series-Laboratory for Foundations of Computer Science ECS LFCS** (1997) (cit. on pp. 6, 43).

[55] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. **A Small Scale Reflection Extension for the Coq system**. Research Report RR-6455. Inria Saclay Ile de France, 2016 (cit. on p. 26).

[56] Andrew D Gordon. „A mechanisation of name-carrying syntax up to alpha-conversion“. In: **Higher Order Logic Theorem Proving and its Applications**. Springer, 1994, pp. 413–425 (cit. on p. 162).

[57] Robert Harper. **Practical foundations for programming languages**. Cambridge University Press, 2016 (cit. on pp. 1, 62).

[58] Robert Harper and John C. Mitchell. „Parametricity and Variants of Girard's J Operator“. In: **Information Processing Letters** 70.1 (Apr. 1999), pp. 1–5 (cit. on p. 2).

[59] Martin Hofmann. „Extensional concepts in intensional type theory“. PhD thesis. University of Edinburgh. College of Science and Engineering. School of Informatics., July 1995 (cit. on p. 10).

[60] Martin Hofmann. „Semantical analysis of higher-order abstract syntax“. In: **Logic in Computer Science, 1999. Proceedings. 14th Symposium on**. IEEE. 1999, pp. 204–213 (cit. on p. 159).

[61] Martin Hofmann. „Syntax and Semantics of Dependent Types“. In: **Semantics and Logics of Computation** (1997). Ed. by Andrew M. Pitts and P.Editors Dybjer, pp. 79–130 (cit. on p. 43).

[62] Gérard Huet. „Confluent Reductions: Abstract Properties and Applications To Term Rewriting Systems“. In: **Journal of the ACM** 27.4 (Oct. 1980), pp. 797–821 (cit. on p. 11).

[63] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. „The power of parameterization in coinductive proof“. In: **ACM SIGPLAN Notices**. Vol. 48. 1. ACM. 2013, pp. 193–206 (cit. on pp. 6, 65, 72, 74, 75, 84).

[64] Martin Hyland and John Power. „The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads“. In: **Electronic Notes in Theoretical Computer Science** 172 (Apr. 2007), pp. 437–458 (cit. on p. 25).

[65] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. „Autosubst 2: Towards Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions“. In: **LFMTP**. Sept. 2017 (cit. on p. 145).

[66] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. „Binder Aware Recursion Over Well-Scoped De Bruijn Syntax“. In: **Certified Programs and Proofs - 7th International Conference, CPP 2018, Los Angeles, USA, January 8-9, 2018** (Jan. 2018) (cit. on pp. 43, 45).

[67] Jonas Kaiser, Tobias Tebbi, and Gert Smolka. „Equivalence of system F and $\lambda 2$ in Coq based on context morphism lemmas“. In: **Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs**. ACM. 2017, pp. 222–234 (cit. on p. 146).

[68] Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. „Needle & Knot: Binder boilerplate tied up“. In: **European Symposium on Programming Languages and Systems**. Springer. 2016, pp. 419–445 (cit. on p. 144).

[69] Stephen Cole Kleene. **Introduction To Metamathematics**. North-Holland, 1950 (cit. on p. 79).

[70] B. Knaster. „Un théorème Sur Les Fonctions D’ensembles“. In: **Annales Soc. Polonaise** 6 (1928), pp. 133–134 (cit. on p. 69).

[71] Neelakantan R Krishnaswami and Derek Dreyer. „Internalizing relational parametricity in the extensional calculus of constructions“. In: **Computer Science Logic**. Vol. 23. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2013 (cit. on p. 166).

[72] Jean-Louis Krivine. „A Call-By-Name Lambda-Calculus Machine“. In: **Higher-order and symbolic computation** 20.3 (2007), pp. 199–207 (cit. on p. 63).

[73] Joachim Lambek. „From Lambda-Calculus To Cartesian Closed Categories“. In: **To HB Curry: essays on combinatory logic, lambda calculus and formalism** (1980), pp. 375–402 (cit. on p. 164).

[74] Serge Lang. **Algebra**. Addison-Wesley, 1993 (cit. on p. 86).

[75] Gyesik Lee, Bruno César dos Santos Oliveira, Sungkeun Cho, and Kwangkeun Yi. „GMeta: A Generic Formal Metatheory Framework for First-Order Representations“. In: **Programming Languages and Systems**. Vol. 7211. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 436–455 (cit. on p. 144).

[76] Xavier Leroy. „Formal Verification of a Realistic Compiler“. In: **Communications of the ACM** 52.7 (2009), pp. 107–115 (cit. on pp. 4, 133).

[77] Paul Blain Levy. „Call-by-push-value: A subsuming paradigm“. In: **International Conference on Typed Lambda Calculi and Applications**. Springer. 1999, pp. 228–243 (cit. on pp. 62, 145).

[78] Zhaohui Luo. „ECC, an Extended Calculus of Constructions“. In: **Fourth Annual Symposium on Logic in Computer Science** (1989) (cit. on pp. 1, 6, 27, 30, 42).

[79] Zhaohui Luo. **Notes on Universes in Type Theory**. Lecture notes for a talk at Institute for Advanced Study, Princeton. 2012 (cit. on p. 42).

[80] Per Martin-Löf. „An intuitionistic theory of types: Predicative part“. In: **Studies in Logic and the Foundations of Mathematics**. Vol. 80. Elsevier, 1975, pp. 73–118 (cit. on pp. 3, 9).

[81] Conor McBride. „Everybody's Got To Be Somewhere“. In: **Electronic Proceedings in Theoretical Computer Science** 275 (July 2018), pp. 53–69 (cit. on p. 26).

[82] Robin Milner. **Communication and Concurrency**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989 (cit. on pp. 1, 84, 85, 89, 91, 97, 98).

[83] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. **The Definition of Standard ML**. MIT Press, 1997 (cit. on p. 3).

[84] Masayuki Mizuno and Eijiro Sumii. „Formal Verification of the Correspondence Between Call-by-Need and Call-by-Name“. In: **International Symposium on Functional and Logic Programming**. Springer. 2018, pp. 1–16 (cit. on p. 146).

[85] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. „Contextual Modal Type Theory“. In: **ACM Transactions on Computational Logic (TOCL)** 9.3 (2008), p. 23 (cit. on p. 158).

[86] Vincent van Oostrom. „Developing Developments“. In: **Theoretical Computer Science** 175.1 (Mar. 1997), pp. 159–181 (cit. on p. 44).

[87] Joachim Parrow and Tjark Weber. „The Largest Respectful Function“. In: **Logical Methods in Computer Science** 12 (2016) (cit. on pp. 70, 76, 85, 87).

[88] Dito Pataraia. **A constructive proof of Tarski's fixed-point theorem for dcpo's**. Presented in the 65th Peripatetic Seminar on Sheaves and Logic, Aarhus, Denmark. Nov. 1997 (cit. on pp. 103–105).

[89] Frank Pfenning and Conal Elliott. „Higher-order abstract syntax“. In: **ACM SIGPLAN Notices**. Vol. 23. 7. ACM. 1988, pp. 199–208 (cit. on pp. 27, 157).

[90] Brigitte Pientka and Joshua Dunfield. „Beluga: A framework for programming and reasoning with deductive systems (system description)“. In: **International Joint Conference on Automated Reasoning**. Springer. 2010, pp. 15–21 (cit. on p. 158).

[91] Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rebecca Zucchini. „Cocon: Computation in Contextual Type Theory“. In: **arXiv preprint arXiv:1901.03378** (2019) (cit. on p. 158).

[92] Andrew M Pitts. „Nominal Logic, a First Order Theory of Names and Binding“. In: **Information and computation** 186.2 (2003), pp. 165–193 (cit. on p. 158).

[93] Andrew M Pitts and Ian Stark. „Operational Reasoning for Functions With Local State“. In: **Higher order operational techniques in semantics** (1998), pp. 227–273 (cit. on p. 60).

[94] Andrew M Pitts, Justus Matthiesen, and Jasper Derikx. „A Dependent Type Theory With Abstractable Names“. In: **Electronic Notes in Theoretical Computer Science** 312 (2015), pp. 19–50 (cit. on p. 159).

[95] Gordon Plotkin. **Lambda-definability and logical relations**. Memo SAI-RM-4, School of Artificial Intelligence, Edinburgh University. 1973 (cit. on p. 59).

[96] Emmanuel Polonowski. „Automatically Generated Infrastructure for De Bruijn Syntaxes“. In: **Interactive Theorem Proving**. Vol. 7998. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 402–417 (cit. on p. 144).

[97] François Pottier. **DBLIB, a Coq library for dealing with binding using de Bruijn indices**. `https://github.com/fpottier/dblib`. Dec. 2013 (cit. on p. 144).

[98] François Pottier. **Revisiting the Cps Transformation and Its Implementation**. 2017 (cit. on p. 146).

[99] Damien Pous. „Coinduction All the Way Up“. In: **Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science - LICS ’16** (2016) (cit. on pp. 1, 5–7, 70, 72, 73, 82, 84, 85, 87, 99).

[100] Damien Pous. „Complete lattices and up-to techniques“. In: **Asian Symposium on Programming Languages and Systems**. Vol. 4867. LNCS. Springer. 2007, pp. 351–366 (cit. on pp. 5, 85).

[101] Damien Pous and Jurriaan Rot. „Companions, codensity and causality“. In: **International Conference on Foundations of Software Science and Computation Structures**. Springer. 2017, pp. 106–123 (cit. on pp. 83, 87, 88).

[102] John C. Reynolds. „Towards a Theory of Type Structure“. In: **Programming Symposium** (1974), pp. 408–425 (cit. on pp. 1, 45).

[103] Davide Sangiorgi. „On the Bisimulation Proof Method“. In: **Mathematical Structures in Computer Science** 8.5 (1998), pp. 447–479 (cit. on pp. 73, 75, 83, 85, 89).

[104] Vijay Saraswat. **Java is not type-safe**. 1997 (cit. on p. 3).

[105] Steven Schäfer and Gert Smolka. „Tower Induction and Up-to Techniques for CCS with Fixed Points“. In: **International Conference on Relational and Algebraic Methods in Computer Science**. Springer. 2017, pp. 274–289 (cit. on pp. 65, 89, 94, 99).

[106] Steven Schäfer, Tobias Tebbi, and Gert Smolka. „Autosubst: Reasoning With De Bruijn Terms and Parallel Substitutions“. In: **Lecture Notes in Computer Science** (2015), pp. 359–374 (cit. on pp. 1, 17, 135, 136, 144).

[107] Steven Schäfer, Sigurd Schneider, and Gert Smolka. „Axiomatic Semantics for Compiler Verification“. In: **Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs**. CPP 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 188–196 (cit. on pp. 102, 103, 132).

[108] Steven Schäfer, Gert Smolka, and Tobias Tebbi. „Completeness and Decidability of De Bruijn Substitution Algebra in Coq“. In: **Proceedings of the 2015 Conference on Certified Programs and Proofs - CPP ’15** (2015) (cit. on pp. 17, 21).

[109] Sigurd Schneider, Gert Smolka, and Sebastian Hack. „A Linear First-Order Functional Intermediate Language for Verified Compilers". In: **Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings**. 2015, pp. 344–358 (cit. on pp. 1, 118).

[110] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, et al. „Ott: Effective Tool Support for the Working Semanticist". In: **Journal of Functional Programming** 20.1 (2010), p. 71 (cit. on p. 144).

[111] Michael Shulman. „Linear Logic for Constructive Mathematics". In: **arXiv preprint arXiv:1805.07518** (2018) (cit. on p. 86).

[112] Gert Smolka, Steven Schäfer, and Christian Doczkal. „Transfinite constructions in classical type theory". In: **International Conference on Interactive Theorem Proving**. Springer. 2015, pp. 391–404 (cit. on pp. 5–7, 65, 70, 76).

[113] Raymond M. Smullyan and Melvin Fitting. **Set Theory and the Continuum Problem**. Clarendon Press, Oxford, 1996 (cit. on pp. 77, 86).

[114] Matthieu Sozeau and Nicolas Oury. „First-Class Type Classes". In: **Theorem Proving in Higher Order Logics** (2008), pp. 278–293 (cit. on p. 139).

[115] Bas Spitters and Eelis van der Weegen. „Type Classes for Mathematics in Type Theory". In: **Mathematical Structures in Computer Science** 21.04 (July 2011), pp. 795–825 (cit. on p. 139).

[116] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. „Autosubst 2: Reasoning with Multi-sorted De Bruijn Terms and Vector Substitutions". In: **Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs**. CPP 2019. Cascais, Portugal: ACM, 2019, pp. 166–180 (cit. on pp. 136, 145).

[117] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, et al. „Dependent types and multi-monadic effects in F^*". In: **ACM SIGPLAN Notices**. Vol. 51. 1. ACM. 2016, pp. 256–270 (cit. on p. 26).

[118] William Walker Tait. „Intensional Interpretations of Functionals of Finite Type I". In: **The Journal of Symbolic Logic** 32.02 (Aug. 1967), pp. 198–212 (cit. on p. 59).

[119] Masako Takahashi. „Parallel Reductions in λ-calculus". In: **Journal of Symbolic Computation** 7.2 (Feb. 1989), pp. 113–123 (cit. on p. 43).

[120] Alfred Tarski. „A Lattice-Theoretical Fixpoint Theorem and Its Applications". In: **Pacific Journal of Mathematics** 5.2 (June 1955), pp. 285–309 (cit. on p. 69).

[121] The Coq Development Team. **The Coq Proof Assistant, version 8.8.0**. Apr. 2018 (cit. on pp. 1, 3, 9).

[122] Amin Timany and Lars Birkedal. **Mechanized relational verification of concurrent programs with continuations**. Draft. 2018 (cit. on p. 146).

[123] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. „A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of Runst". In: **Proceedings of the ACM on Programming Languages** 2.POPL (2017), p. 64 (cit. on p. 146).

[124] Yoshihito Toyama. „Commutativity of Term Rewriting Systems". In: **Programming of future generation computers II** (1988), pp. 393–407 (cit. on p. 44).

[125] Anne Sjerp Troelstra and Dirk Van Dalen. **Constructivism in mathematics**. Elsevier, 2014 (cit. on p. 86).

[126] The Univalent Foundations Program. **Homotopy Type Theory: Univalent Foundations of Mathematics**. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cit. on pp. 9, 15).

[127] Christian Urban, Stefan Berghofer, and Michael Norrish. „Barendregt's Variable Convention in Rule Inductions". In: **Lecture Notes in Computer Science** (2007), pp. 35–50 (cit. on p. 4).

[128] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. „Contextual Equivalence for a Probabilistic Language With Continuous Random Variables and Recursion". In: **Probabilistic Programming Languages, Semantics, and Systems** (2018) (cit. on p. 146).

[129] Ernst Zermelo. „Neuer Beweis Für Die Möglichkeit Einer Wohlordnung". In: **Mathematische Annalen** 65.1 (Mar. 1908), pp. 107–128 (cit. on p. 85).

Representing Syntax with Binders

We give an overview over some of the existing approaches to formalizing syntax with binders.

As mentioned in the introduction, the different approaches can be classified into synthetic and explicit methods. Synthetic methods encode syntax with binders at the level of logic and require special support from a proof assistant. Explicit methods work directly with an encoding of syntax with binders.

We use the types of System F as a running example. The syntax of System F types is informally given by the following grammar.

$$A, B \ ::= \ X \mid A \to B \mid \forall X.\, A$$

We will give a brief overview over existing synthetic and explicit methods followed by a more in-depth discussion of de Bruijn representations.

Synthetic Methods

We start with an overview over the existing synthetic methods.

Higher-order abstract syntax (HOAS) [89] uses functions to represent binders. The idea is that an open term may be represented as a function taking its free variables as arguments. For example, the syntax of System F types would be modeled as an "inductive type" $\mathbb{T}$ with the following constructors.

$$\begin{aligned} _ \to _ &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{A} &: (\mathbb{T} \to \mathbb{T}) \to \mathbb{T} \end{aligned}$$

The type $\forall X.\, X$ is represented using the higher order constructor A as $\mathsf{A}\,\mathsf{id}$, where id is the meta-level identity function $\mathsf{id}\, x = x$. There is no constructor for variables, as object-level variables are represented using meta-level variables.

This use of meta-level binders to represent object level binders leads to very natural definitions. Single variable instantiation is built into the HOAS approach and implemented by function application. This also implies that all definitions automatically respect instantiation.

On the flip side, the HOAS representation implies that the term function space $\mathbb{T} \to \mathbb{T}$ must correspond to the type of open types with a single free variable. Without

this restriction the encoding would contain "exotic" terms which do not correspond to any type defined by the original grammar. For example, when working in a classical metatheory we can define the following function.

$$g\,x \;=\; \begin{cases} x & \text{if } x = A \to B \\ \mathsf{A}\,\mathsf{id} & \text{otherwise} \end{cases}$$

But now $\mathsf{A}\,g$ does not correspond to a System F type.

It follows that a synthetic approach based on HOAS is incompatible with classical reasoning and more generally disallows writing any non-trivial function on terms. Moreover, since variables are represented as metalevel variables, they are implicit and we cannot write any definition which mentions them explicitly. For example, it is impossible to write a function to compute the set of free variables of a term in a HOAS representation. Within this framework, not even the statement makes sense, since every statement must be stable under instantiation.

This makes it impossible to use HOAS in a general purpose proof assistant. It is still possible to work explicitly within a model of HOAS [45], but doing so involves additional proof obligations and looses many of the benefits of the synthetic approach.

Dedicated proof assistants based on HOAS exist [49, 90], but carry restrictions especially when it comes to recursive definitions on terms.

Contextual modal type theory (CMTT) [85] can be seen as an extension of the HOAS approach with a modality that allows one to talk directly about open terms. This capability is otherwise missing from a system based on HOAS and adds a great deal of expressivity.

With this modality comes a notion of context morphism or parallel substitution. CMTT is unique among synthetic approaches in being able to talk about parallel instantiation directly.

The proof assistant Beluga [90] implements CMTT.

There is recent work [91] which incorporates some recursive definitions into CMTT. In the future this could potentially lead to an integration of CMTT into a general purpose proof assistant. At the time of this writing, such an integration is not yet available.

Nominal logic [92] is a synthetic approach that seeks to faithfully reproduce informal paper proofs. Unlike in the HOAS approach, variables are explicitly manipulated. Variables are named and have identity.

Internally, every term carries a *support*, a finite set of variables that it may contain, and every definition is invariant under injective renaming. Freshness side conditions

and quantification over fresh variables are directly encoded as new operations in nominal logic by relying on the support. Writing $\mathbb{A}$ for the type of variables and $«\mathbb{A}»\,\mathbb{T}$ for the abstraction of a fresh name in $\mathbb{T}$ we can represent the types of System F in nominal logic as an inductive type $\mathbb{T}$ with the following constructors.

$$\begin{aligned} \mathsf{V} &: \mathbb{A} \to \mathbb{T} \\ _ \to _ &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{A} &: («\mathbb{A}»\,\mathbb{T}) \to \mathbb{T} \end{aligned}$$

Injective single variable renaming, that is swapping a variable for another potentially fresh variable, is a first-class operation. The nominal approach is unique among synthetic approaches in that it is compatible with classical reasoning.

On the flip-side, general renaming and instantiation are not first-class and have to be defined. The corresponding substitution lemmas have to be shown by hand and definitions need explicit freshness side-conditions. Essentially, Nominal Logic offers a sound approximation to the Barendregt convention. There is an operation which introduces only fresh variables, but freshness side conditions must still be kept track of explicitly.

Beyond this, the main drawback for us is that there is currently no model of nominal type theory [94] with type universes, which would be required for practical use in constructive type theory.

Weak higher-order abstract syntax [60] represents binders as functions of the form $\mathbb{V} \to \mathbb{T}$, where $\mathbb{V}$ is a special type of variables and $\mathbb{T}$ is a type of terms. For example, we represent the types of System F using an inductive type with the following constructors.

$$\begin{aligned} \mathsf{V} &: \mathbb{V} \to \mathbb{T} \\ _ \to _ &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{A} &: (\mathbb{V} \to \mathbb{T}) \to \mathbb{T} \end{aligned}$$

Analogous to the HOAS approach this implies that all definitions are automatically compatible with renaming. Single variable renaming is built-into the definition and realized by function application.

On the other hand, general instantiation has to be defined and not all definitions are compatible with instantiation.

The main advantage of weak HOAS over HOAS is that the former admits larger function spaces of the form $\mathbb{T} \to \mathbb{T}$. This makes it easier to model weak HOAS. In fact,

weak HOAS can almost be internalized in type theory in the form of polymorphic HOAS.

Polymorphic HOAS [31] can be seen as internalizing weak HOAS in a type theory with internal parametricity [24]. We model a type of terms $\mathbb{T}\,V$ parameterized over a type of variables. Binders are represented using functions $V \to \mathbb{T}\,V$. For the types of System F we obtain an inductive type with the following constructors.

$$\begin{aligned} \mathsf{V} &: V \to \mathbb{T}\,V \\ _ \to _ &: \mathbb{T}\,V \to \mathbb{T}\,V \to \mathbb{T}\,V \\ \mathsf{A} &: (V \to \mathbb{T}\,V) \to \mathbb{T}\,V \end{aligned}$$

The type of System F types is obtained by quantifying over the type V of variables as $\forall V.\ \mathbb{T}\,V$. In a type theory with internal parametricity, every term is parametric in the choice of V and this can be exploited to show that there are no exotic terms.

Polymorphic HOAS is unique among the synthetic approaches in that the parametricity axiom for a specific term type can be asserted in a general purpose proof assistant. The assumption is anti-classical, but beyond that PHOAS offers a practical methodology for working with weak HOAS inside of a proof assistant.

Discussion In general, for synthetic approaches we build in certain well-formedness conditions and only allow well formed definitions. Well-formedness is always some form of compatibility with instantiation. At one end of the spectrum we have Nominal Logic, where definitions are only compatible with injetive renaming. On the other end we have HOAS where all definitions are compatible with arbitrary instantiations. The weak HOAS approach sits in between and offers compatibility with arbitrary renamings.

Ultimately, we should strive for a synthetic approach to syntax with binding, yet at this point it is not clear which features an ideal synthetic framework should have.

For many proofs (e.g., of normalization properties) it is convenient to talk about full parallel instantiation. Similarly, compatibility with instantiation is a fundamental property that all of our definitions should satisfy. This combination is realized to some extent in CMTT. Every other synthetic approach lacks a good notion of parallel instantiation, or offers only partial compatibility with instantiation.

Yet CMTT currently does not offer good support for specifying recursive definitions on terms.

From all synthetic approaches, only weak HOAS can be axiomatized in a general purpose proof assistant in the form of PHOAS. However, weak HOAS offers few guarantees and makes parallel substitutions more difficult to handle.

In practice, we prefer to work with an explicit encoding of syntax with binders.

Explicit Methods

The following are some of the existing explicit approaches to representing syntax with binders.

Named terms modulo α-equivalence [32, 18] are by far the closest to informal paper proofs. Writing $\mathbb{V}$ for a fixed countably infinite type of variables (for instance, we can take $\mathbb{V} = \mathbb{N}$) we represent the types of System F with an inductive type $\mathbb{T}$ with the following constructors.

$$\begin{aligned} \mathsf{V} &: \mathbb{V} \to \mathbb{T} \\ _ \to _ &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{A} &: \mathbb{V} \to \mathbb{T} \to \mathbb{T} \end{aligned}$$

We represent α-equivalence with an inductively defined relation $A \sim_\alpha B$ between types. Informally, we then obtain the representation of System F types as the quotient of types by α-equivalence.

There is a large gap between informal presentations and a completely formal treatment of named terms modulo α-equivalence. There are many side conditions and complicated substitution lemmas when working with named terms modulo α-equivalence.

On the practical side, the type theory underlying Coq currently does not support quotient types. This forces us to work with a concrete representation of equivalence classes of terms. A clever choice of representation for these equivalence classes then leads to the de Bruijn and locally nameless representations.

De Bruijn representation [30] is based on the simple observation that we can handle α-equivalence by switching to a canonical nameless representation where α-equivalence is term equality. Names for variables are useful for communication, but are not relevant for the semantics of terms. Formally, a variable represents a reference to a binder or to a context outside of a term. In de Bruijn representation, these binders are implemented using natural numbers — the so-called de Bruijn indices — where the number n refers to the n-th enclosing binder counting from 0.

In de Bruijn representation, the types of System F are represented by an inductive

type $\mathbb{T}$ with the following constructors.

$$\begin{aligned} \mathsf{V} &: \mathbb{N} \to \mathbb{T} \\ _ \to _ &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{A} &: \mathbb{T} \to \mathbb{T} \end{aligned}$$

Note that it is not clear from the type where variables are bound.

One oft cited drawback of de Bruijn representation is that individual terms become less readable. For example, the type $\forall XY.\ X \to Y \to X$ is represented by the de Bruijn term $\mathsf{A}(\mathsf{A}(1 \to 0 \to 1))$.

Locally nameless representation [56] partially addresses this issue by representing bound variable using de Bruin indices and free variables with explicit names. This makes it easier to translate informal proofs about syntax with binders.

For example, assuming that there is a type of names $\mathbb{A}$, System F types are represented with an inductive type $\mathbb{T}$ with the following constructors.

$$\begin{aligned} \mathsf{V}_B &: \mathbb{N} \to \mathbb{T} \\ \mathsf{V}_F &: \mathbb{A} \to \mathbb{T} \\ _ \to _ &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{A} &: \mathbb{T} \to \mathbb{T} \end{aligned}$$

In particular, there are two variable constructors. The constructor V_B represents bound variables and V_F represents free variables.

Unfortunately, working with locally nameless terms leads to duplication of work, since there are now two different forms of instantiation: instantiation on named terms and de Bruijn instantiation. This means that every substitution lemma appears in two forms. Alternatively, it is possible to restrict attention to terms without free de Bruijn indices. This requires a different set of side conditions, which are somewhat analogous to the freshness side conditions in a named approach.

Finally, locally nameless syntax offers only a partial fix to the readability problem of de Bruijn terms. In particular, the closed terms coincide in both representations.

Discussion The main disadvantage of explicit over synthetic methods is analogous to the main disadvantage of set theory over type theory — we have to work directly with an encoding. It is unavoidable that any encoding contains more structure than the ideal we wish to formalize. In order to work effectively with an encoding we need to restrict ourselves to semantically meaningful operations. For example, variables in de Bruijn terms are natural numbers, but this is merely an encoding for a reference

to a binder or a position in a context. While constant addition on de Bruijn indices has some semantic meaning — it corresponds to context extension — there are other arithmetic operations such as multiplication which are meaningless for the de Bruijn representation.

The basic spectrum of explicit methods for representing syntax with binders is between named and nameless terms. To some extent these representations are all equivalent, but differ in whether quotient types are explicit or implicit. Named terms are more readable, but complicate the formalization and require support for quotient types. Among nameless representations, there seems to be little advantage to the locally nameless approach. Locally nameless terms inherit some problems from both the named approach (additional side conditions) and de Bruijn approach (unreadable terms).

De Bruijn Representations

The basic idea behind the nameless approach is that variables are references. There are several ways of encoding references. De Bruijn chooses to represent references as natural numbers, but there are other encodings which work equally well.

The main complication with other encodings for references is that we need to settle on an encoding for free variables. Remember that free variables represent references to an outside context — yet the context is implicit in pure de Bruijn representation. We can make this context explicit by parameterizing the type of terms by contexts in some way.

This leads to a spectrum of de Bruijn representations.

Pure de Bruijn terms As already mentioned, in de Bruijn terms [30] the context is fully implicit and terms are not parameterized. This is the simplest possible encoding, and can be used without dependent types.

On the other hand, it also offers the least amount of type safety.

In general, it requires some insight to translate an informal presentation to de Bruijn terms [17], since some side conditions must be encoded using instantiation. For example, the η-reduction rule for λ-terms states that a lambda term of the form $\lambda x.\ s\, x$ reduces to s when x does not appear free in s.

$$\frac{x \notin \mathrm{FV}\ s}{\lambda x.\ s\, x \rhd s}$$

In de Bruijn representation we render this rule by saying that the term $\lambda(s[\uparrow]\,0)$ reduces to s.

$$\frac{}{\lambda(s[\uparrow]\,0) \rhd s}$$

This is not so much a side condition as it is the only sensible way of writing this rule. Since s appears on its own on the right hand side, it must be a valid term in this context. On the left, s appears under an additional binder, hence must be shifted into this extended context. This sort of reasoning about the scope of terms is required when working with de Bruijn terms, and is usually better encoded directly into the types of terms.

Finally, since de Bruijn terms do not constrain the size of the context, we do not statically know that the context is finite, i.e., that terms contain only finitely many free variables. This is sometimes required for syntactic translations, for example, when interpreting simply typed λ-terms into cartesian closed categories [73].

Well-scoped terms [25] represent terms as a family $\mathbb{T} : \mathbb{N} \to \mathcal{U}$ of types indexed with the size of the context. Variables are positions in the context, thus encoded by the elements of the corresponding finite type.

Writing I_n for the finite type with n elements, the types of System F are represented by an inductive family $\mathbb{T} : \mathbb{N} \to \mathcal{U}$ with the following constructors.

$$\begin{aligned} \mathsf{V} &: \mathsf{I}_n \to \mathbb{T}\,n \\ _ \to _ &: \mathbb{T}\,n \to \mathbb{T}\,n \to \mathbb{T}\,n \\ \mathsf{A} &: \mathbb{T}\,(n+1) \to \mathbb{T}\,n \end{aligned}$$

Well-scoped de Bruijn terms offer more type safety compared to pure de Bruijn terms. For example, due to the types there is never any ambiguity over when shift renamings have to be inserted, as in the η-rule above. The types enforce correct scoping, which we have found to be the most error prone part of encoding definitions into de Bruijn representation.

Well-scoped terms also make the size of the context explicit and moreover, encode the assumption that the context is finite. Even beyond this, it is still helpful in that it allows us to choose a more precise encoding of contexts for type systems. For example, in the typing judgment for System F ($\Gamma \vdash s : A$) we can represent Γ as a type substitution, i.e., a mapping from the finite type of term context positions to types. This correctly encodes the fact that there is only a single context in which to typecheck a closed term, namely the empty context. In the plain de Bruijn representation we

are instead sometimes forced to supply “dummy” values.

The main drawback with using well-scoped terms over plain de Bruijn terms is that it is sometimes beneficial to talk about ill-scoped terms. For example, Girard's normalization proof for System F [52] implicitly uses ill-scoped terms to show that a certain logical relation is non-empty, by showing that it always contains variables. However, in an empty context, there would be no variables and the proof breaks down.

Monadic terms [10] represent terms as a monad $\mathbb{T} : \mathcal{U} \to \mathcal{U}$ of types indexed with the type of context positions. This naturally admits terms in a context with infinitely many variables and is more general than using well-scoped terms.

In monadic style, System F types are represented by an inductive family $\mathbb{T} : \mathcal{U} \to \mathcal{U}$ with the following constructors, where we write $X+1$ for the inductive type extending the type X by a single additional element (this type is usually called $\mathrm{option}\ X$ or $\mathrm{Maybe}\ X$).

$$\begin{aligned} \mathsf{V} &: X \to \mathbb{T}\,X \\ _ \to _ &: \mathbb{T}\,X \to \mathbb{T}\,X \to \mathbb{T}\,X \\ \mathsf{A} &: \mathbb{T}\,(X+1) \to \mathbb{T}\,X \end{aligned}$$

The main drawback with this approach is that variables do not have identity. Conceptually, the reason for this is that we do not have access to the context, merely to the type of context positions. This makes it more difficult to represent type systems which use variable identity, for example, the algorithmic typing judgement of System $F_{<:}$ which is the subject of the POPLmark challenge [17].

Beyond this, monadic terms offer the same advantages and drawbacks as well-scoped terms.

Well-typed terms [22] are the most precise representation, where terms are indexed by a typing context and type $\mathbb{T}\,\Gamma\,A$. This provides even better type safety than well-scoped or monadic terms and allows us to encode many properties of typing right into our definitions. For example, type preservation under evaluation is naturally enforced by the type of the evaluation relation.

Incidentally, well-typed System F types correspond to well-scoped System F types, since the kinding judgment on System F types is degenerate.

Well-typed terms represent a sweet spot for representing simply typed systems. For more expressive type disciplines, the additional type constraints can become difficult to manage. For example, in a well-typed representation of System F we need to

introduce the following dependently typed constructor for type abstraction.

$$\Lambda_ \; : \; \mathbb{E}\,(\Gamma \circ \uparrow)\,A \to \mathbb{E}\,\Gamma\,(\forall\,A)$$

where $\mathbb{E}$ is the type of System F terms and $\Gamma \circ \uparrow$ refers to composition of substitutions. This is only applicable to arguments whose context is exactly of the form $\Gamma \circ \uparrow$. Unfortunately, this results in difficult unification problems during elaboration, which at least the Coq proof assistant cannot solve automatically. Moreover, there are cases where the context is equal to a context of this form, but not definitionally so. For example, we have $\Gamma \circ (\uparrow \circ \uparrow) = (\Gamma \circ \uparrow) \circ \uparrow$ propositionally but not definitionally. This means that terms have to be manually transported along this equality.

The situation becomes even more complicated when we move to dependent types, where we need inductive-inductive types to represent well-typed terms of a dependently typed language [9]. With a presentation of dependent types a la Russel (with no distinction between types and terms, see Chapter 4) there does not seem to be any sensible definition of well-typed terms at all. Specifically, in a type theory a la Russel, the type of well-typed terms would have to be indexed over itself.

There is one final drawback to using well-typed terms, namely that it is sometimes beneficial to talk about syntactically ill-typed, but semantically well-typed terms. For example, Krishnaswami and Dreyer [71] use semantically well-typed terms to show the admissibility of a strong eliminator for Church-encoded Σ-types in type theory. The terms involved would not exist in a well-typed presentation.

Discussion Summarizing, well-typed terms are the most precise encoding of syntax with binders, but not always applicable. Having an explicitly sized context, as in the well-typed or well-scoped representation is also helpful in practice. In terms of type safety we have found that well-scoped, monadic, and well-typed terms perform very similarly. All variants catch most common mistakes in practice. Finally, out of the well-scoped and monadic term representations we recommend using well-scoped terms. Although it is easy to obtain well-scoped terms from monadic terms and vice versa in well-behaved examples [13], the assumption that terms depend only on a finite context is sometimes useful.