The Logic of Hereditary Harrop Formulas as a Specification Logic for Hybrid

Chelsea Battell

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of Master of Science in Mathematics[1]

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

© Chelsea Battell, Ottawa, Canada, 2016

[1]The M.Sc. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

Hybrid is a two-level logical framework that supports higher-order abstract syntax (HOAS), where a specification logic (SL) extends the class of object logics (OLs) we can reason about. We develop a new Hybrid SL and formalize its metatheory, proving weakening, contraction, exchange, and cut admissibility; results that greatly simplify reasoning about OLs in systems providing HOAS. The SL is a sequent calculus defined as an inductive type in Coq and we prove properties by structural induction over SL sequents. We also present a generalized SL and metatheory statement, allowing us to prove many cases of such theorems in a general way and understand how to identify and prove the difficult cases. We make a concrete and measurable improvement to Hybrid with the new SL formalization and provide a technique for abstracting such proofs, leading to a condensed presentation, greater understanding, and a generalization that may be instantiated to other logics.

Dedications

To Amy Felty, my advisor and role model.

Acknowledgement

Thanks and acknowledgment go to the Natural Sciences and Engineering Research Council (NSERC) of Canada, Professor Amy Felty, and the University of Ottawa for the financial support provided while completing this research.

I am extremely grateful for the guidance provided by Professor Felty to help me through the many challenges faced while completing this degree and for her feedback and assistance in editing all of the presentations and written documents related to this research.

I would also like to thank Alberto Momigliano for the initial encoding of the data structures for the specification logic and discussions through the course of this work.

Contents

List of Figures **viii**

1 Introduction **1**
1.1 Mechanized Reasoning 1
1.2 Higher-Order Abstract Syntax 2
1.3 Hybrid 3
1.4 Specification Logic Metatheory 3
1.5 Outline 4

I Background **5**

2 Coq **6**
2.1 The Calculus of Constructions 6
2.1.1 Terms 6
2.1.2 Judgments 7
2.2 Simply Typed Lambda Calculus 11
2.3 Reductions 11
2.3.1 Convertibility 12
2.4 Dependent Types 12
2.5 Higher-Order Types 13
2.5.1 Polymorphism 14
2.5.2 Type Operators 16
2.6 Interactive Proving in Coq 17
2.6.1 Proof State 17
2.6.2 Some Coq Tactics, Tacticals, and Commands 20
2.7 Induction in Coq 21
2.7.1 Mutually Inductive Types 24
2.8 Conclusion 25

3 Hybrid **27**
3.1 Object Logics 28
3.2 Ambient Logic 30
3.3 Representing Higher-Order Abstract Syntax in Hybrid 30
3.4 Specification Logic 31
3.5 Example OL Implementation 34
3.6 Comparison to Other Architectures 37

4 Hereditary Harrop Formulas **39**
4.1 Higher-Order Hereditary Harrop Formulas 39
4.2 Focusing 42

II Contributions **46**

5 Specification Logic **47**
5.1 Contexts in Coq 47
5.2 Hereditary Harrop Specification Logic in Coq 49
5.3 Mutual Structural Induction 51

6 Specification Logic Metatheory **55**
6.1 Structural Rules 56
6.2 Cut Admissibility 61
6.2.1 Subcase for *g_dyn*: Alternate Proof Attempt 64
6.2.2 Subcase for *g_dyn*: Original Proof Structure 67

7 Generalized Specification Logic **72**
7.1 SL Rules from GSL Rules 73

8 Generalized Specification Logic Metatheory **76**
8.1 GSL Induction Part I: A Restricted Theorem 76
8.1.1 Sequent Subgoals 77
8.1.2 Non-Sequent Subgoals 79
8.2 GSL Induction Part II: The Structural Rules Hold 82
8.2.1 Sequent Subgoals 82
8.2.2 Non-Sequent Subgoals 83
8.3 GSL Induction Part III: Cut Rule Proven Admissible 84
8.3.1 Sequent Subgoals 84
8.3.2 Non-Sequent Subgoals 85

9 Conclusion **87**
9.1 Related Work . . . 87
9.2 Future Work . . . 88

A Notations **90**

Bibliography **94**

Index **94**

List of Figures

2.1 Rules of CoC 9
2.2 Subtyping in CoC 12
2.3 Induction principle for type `nat` 22

3.1 High-Level Hybrid Structure 27
3.2 Terms in Hybrid 31
3.3 Typing of λ-calculus Abstractions 32
3.4 Type of SL Formulas 33
3.5 Induction Principle for `oo` 34
3.6 Example OL: Encoding Syntax in Hybrid 35
3.7 Example OL: Encoding OL Inference Rules *hodb_app* and *hodb_abs* in Hybrid 36

4.1 The logic of higher-order hereditary Harrop formulas 41
4.2 The logic of higher-order hereditary Harrop formulas with focusing 43

5.1 Goal-Reduction Rules, `grseq` : `context` → `oo` → `Prop` 50
5.2 Backchaining Rules, `bcseq` : `context` → `oo` → `atm` → `Prop` 51
5.3 SL Sequent Mutual Induction Principle 52

6.1 Coq proof of `monotone` (Theorem 6.7) 71
6.2 Coq proof of 98/105 cases of `cut_admissible` (Theorem 6.8) 71

8.1 Proof state of GSL induction after rule application 78
8.2 Incomplete proof branches for sequent premises 79
8.3 Incomplete proof branch (*g_dyn* case) 81

List of Theorems

2.1 Definition (Context) 7
2.2 Definition (Sequent, Judgment) 8
2.3 Definition (Derivation, Valid/Provable Sequent) 8
2.4 Definition (Dependent Product) 8
2.5 Definition (Inhabited-in-Context) 10
2.6 Definition (Specification, Realization) 10
2.7 Definition (Formula, Proof Object) 10
2.8 Definition ($\beta\delta\iota\zeta$-Convertible) 12
2.1 Example (Tuple as a Dependent Type) 12
2.2 Example (Parametrized Types) 13
2.3 Example (Predicates) 13
2.4 Example (Logical Connectives) 16
2.9 Definition (Proof State) 17
2.5 Example (Interactive Proof) 18
2.6 Example (Proof by Induction) 22

3.1 Example (Object Logic: Equivalence of Named and Nameless λ-terms) 28

4.1 Definition (Higher-Order Hereditary Harrop Formulas) 39
4.1 Example (Hereditary Harrop Derivations) 40
4.2 Definition (Uniform Proof) 42
4.2 Example (Hereditary Harrop Focused Derivation) 44

5.1 Lemma (`elem_inv`) 48
5.2 Lemma (`elem_sub`) 48
5.3 Lemma (`elem_self`) 48
5.4 Lemma (`elem_rep`) 48
5.5 Lemma (`context_swap`) 48
5.6 Lemma (`context_sub_sup`) 48

6.1 Theorem (`gr_weakening`) 56

6.2 Theorem (bc_weakening) 56
6.3 Theorem (gr_contraction) 56
6.4 Theorem (bc_contraction) 56
6.5 Theorem (gr_exchange) 56
6.6 Theorem (bc_exchange) 56
6.7 Theorem (monotone) 57
6.8 Theorem (cut_admissible) 61

Chapter 1

Introduction

The goal of this research is to increase the reasoning abilities of an existing system that is intended to help mechanize programming language metatheory. The system that this work contributes to is called Hybrid [8] and is part of the research program carried out by the Software Correctness and Safety Research Laboratory at the University of Ottawa under the supervision of Professor Amy Felty. Hybrid is implemented both in Coq [24] and Isabelle/HOL [18], interactive proof assistants used for applications such as formalizing mathematics, certifying compilers, and proving correctness of programs. Our application of Coq is proving metatheory of formal systems efficiently. The contributions to Hybrid described in this thesis are to the Coq implementation.

1.1 Mechanized Reasoning

Proof is essential in modern mathematics and in logic in particular. We trust and build on the work of others when we can see that they have presented a rigorous argument supporting their work. When writing proofs with many cases or details to manage, it is easy to make errors and these proofs are tedious to check. So we must trust the proof writer (and all proofs that their work builds on) or else spend an exorbitant amount of time checking proofs (and still possibly miss errors). Proof assistants such as Coq provide proof terms that can be independently checked. To trust all proof terms of theorems proven in Coq, one only needs to trust the underlying proof theory and the implementation of the system checking the proof.

Manually checking "paper and pencil" (non-formalized) proofs is not a scalable or practical technique for applications to software development in industry where financial concerns may be prioritized over software correctness and safety. The area of formal methods for software engineering is focused on (im)proving the correctness of software. Even if we develop techniques to allow software developers to prove correctness of software without needing expertise in the research area, we still need to be sure that the languages in which the programs are defined cause the expected behaviour.

We need a mechanized solution to studying programming language metatheory.

Toward this goal, the POPLmark challenge [1] was introduced to merge the concerns of the proof theory and programming languages communities. It provides a set of challenge problems to explore how to mechanize programming language metatheory using a variety of systems and techniques and illustrates the importance of formalized reasoning in programming language research. In fact, it is standard for papers presented at programming language conferences to be accompanied by formal proofs of the metatheory, again for reasons related to confidence in the correctness of the work.

One approach to mechanize reasoning about programming languages involves higher-order abstract syntax. This technique is used by Hybrid, the system that the work in this thesis contributes to.

1.2 Higher-Order Abstract Syntax

Higher-order abstract syntax (HOAS) [20], also known as λ-tree syntax [17], is a technique for representing formal systems, known as *object logics* (OLs), that we wish to reason about. A variety of systems exist that implement a HOAS approach to reasoning about OLs such as programming languages and logics. An early example is the Twelf system [23]. HOAS simplifies reasoning about OLs in these systems by allowing object-level name binding structures, also called *binders*, to be encoded in the binding structures of the meta-language that the system is defined in.

As an example, we will illustrate a HOAS encoding of the untyped λ-calculus (the OL) in a type theory (the meta-level) so that we can reason about it formally. Function abstraction in λ-calculus is a *binder* because the name of a variable is bound in the body of the abstraction. Let *tm* be the meta-level type of terms of the OL encoding. Suppose that we have constants expressing the higher-order syntax of terms, including *app* of type $tm \rightarrow tm \rightarrow tm$ and *abs* of type $(\underline{tm} \rightarrow tm) \rightarrow tm$. Then *abs* represents the function abstraction construct of the OL, an object-level binder, and it is encoded in a meta-level binder. For example, the λ-term $\lambda x.\lambda y.x\ y$ can be encoded as $abs(\lambda x\,.\,(abs(\lambda y\,.\,(app\ x\ y))))$. Note that *tm* cannot be defined inductively because of the (underlined) negative occurrence of *tm* in the type of *abs*.

When the metalanguage is an appropriate λ-calculus, we can encode object-level substitution and renaming as meta-level β-reduction and α-conversion, respectively. This avoids the requirement of implementing infrastructure consisting of libraries of definitions and lemmas to deal with issues surrounding binders and variable naming when studying an OL.

1.3 Hybrid

Coq is an implementation of a typed λ-calculus called the calculus of constructions. Hybrid is implemented as a Coq library so the meta-language is a λ-calculus. A type *expr* is defined for representing OL "programs" as meta-level terms. In Hybrid this is implemented so that terms of type *expr* expand to a nameless representation of λ-terms called de Bruijn indices [7]. Hybrid uses HOAS and object-level binders are encoded in a newly introduced abstraction binder. This binder has type $(expr \rightarrow expr) \rightarrow expr$. It can be used to directly express the syntax of OLs such as the untyped λ-calculus example in Section 1.2.

An intermediate reasoning layer called a *specification logic* (SL) is added to interface between the OL encoding and the layer implementing HOAS. Adding a SL extends the class of OLs that we can reason about efficiently using a system supporting HOAS. The relationship between these levels is the reason Hybrid is considered a *two-level* logical framework. This approach was introduced by McDowell and Miller in [14] with the $FO\lambda^{\Delta\mathbb{N}}$ logic. In such a system, the specification and (inductive) meta-reasoning are done within a single system but at different levels. In Hybrid the SL is defined as an inductive type in Coq, and OL judgments (including hypothetical and parametric judgments) are encoded in the SL.

1.4 Specification Logic Metatheory

There are many features of Hybrid that help work toward the goal of efficiently mechanizing programming language metatheory, but this thesis is focused on the SL layer. Here we makes two contributions: an extension to the reasoning power of Hybrid and new insight into proofs of properties of certain kinds of sequent calculi.

First, a new SL is implemented and structural properties of this logic are formalized and proven in the Coq proof assistant. The new SL presented here is a sequent calculus based on the logic of hereditary Harrop formulas as presented in [15]. We prove that the standard structural rules of weakening, contraction, exchange and cut are admissible in this logic. In proving admissibility of these rules, we do not have to include them in the logic as axiomatic and we still get the benefits that they provide in reasoning about OLs. The structural rules can then be used in proofs of OL theorems. For example, if the OL is a typed functional programming language, then proving subject reduction for this OL (i.e. that evaluation of expressions preserves typing) requires the cut rule. See [8] for a detailed explanation of this example and subject reduction proof. Implementing a new SL and proving the admissibility of structural rules is a concrete extension to an existing computing tool (namely Hybrid) since it improves the reasoning abilities of this system in a fully formalized way.

The second contribution is more theoretical and educational. We present a generalization of the specification logic and form of theorem statement to encapsulate

the implemented SL and desired structural rules, respectively. We show how the implemented SL can be instantiated from this generalized SL, so the results presented for the generalized SL can be applied to, and provide guidance on, the proofs of SL metatheory. The structural proofs are by induction, sometimes requiring mutual inductions and nested inductions, so they can have many cases and details to manage. This presentation allows us to see the structural proofs in a more condensed but still comprehensive way. We are also able to gain a deeper understanding of these proofs; the generalized SL helped us to partition cases for the original SL into classes with the same proof structure and isolate the difficult cases. It is our hope that this presentation will give others insight into the kind of proofs we work through and that this general framework may find some use in other applications.

1.5 Outline

This thesis is broken up into two parts: background and contributions. To understand the research described here, it is necessary to first ensure that the reader understands the logical foundations of the ambient reasoning system for this work (namely Coq), the basics of Hybrid, and the development of the style of logic used for the new SL. In Chapter 2 we review the type theory implemented by Coq and introduce the reader to using it as a proof assistant. We present an overview of Hybrid in Chapter 3. The final background chapter, Chapter 4, is on the logic of higher-order hereditary Harrop formulas and will set the stage for the SL of this thesis. Next we move to the contributions of this research. As stated above, this is focused on a new intermediate reasoning logic for Hybrid. Chapter 5 presents this logic and its metatheory is studied in Chapter 6. From here we abstract the specification logic of Chapter 5 with a generalized specification logic in Chapter 7 and prove properties of this logic in a general way in Chapter 8. We conclude in Chapter 9 with a review of the results presented and look at related and future work.

The research presented in this thesis is published in [2]. The files of the Coq formalization are available at `www.eecs.uottawa.ca/~afelty/BattellThesis/`.

Part I

Background

Chapter 2

Coq

Coq [3,24] is an implementation of the Calculus of Inductive Constructions (CIC), an extension of the Calculus of Constructions (CoC) [6], a typed lambda calculus with dependent types, polymorphism, and type operators. CIC extends CoC by adding inductive types, which will be explored in Section 2.7. Originally created by Thierry Coquand, CoC and its extensions have spurred the development of a variety of proof assistants and interactive theorem proving systems currently used in the research areas of automated deduction and formal methods for software engineering.

We will explore the Calculus of Constructions in Section 2.1, then see how it captures the simply typed λ-calculus in Section 2.2. Next we look at reductions in Section 2.3, followed by an exploration of the expressive power of CoC in Sections 2.4 and 2.5, where we will examine how the rules presented earlier in the chapter allow dependent types, polymorphism, and type operators. We see how to use Coq as an interactive proof assistant in Section 2.6 followed by information on inductive types and writing inductive proofs in Section 2.7. The notation and style used to illustrate the concepts follows the presentation in [3] and [24]. The discussion is motivated by [3] and [6].

2.1 The Calculus of Constructions

2.1.1 Terms

The terms of CoC are defined by the following grammar:

$$\begin{aligned} t_1, t_2, t_3 \ ::= \ & \mathit{Type}_i \\ & \mid \mathit{Set} \\ & \mid \mathit{Prop} \\ & \mid x_i \end{aligned}$$

$$
\begin{aligned}
&\mid t_1\ t_2 \\
&\mid \lambda x : t_1.t_2 \\
&\mid \forall x : t_1, t_2 \\
&\mid \mathit{let}\ x := t_1 : t_2\ \mathit{in}\ t_3
\end{aligned}
$$

Terms of CoC include a collection of constants indexed by natural numbers where for $i \in \mathbb{N}$, Type_i denotes the ith constant. Together with this collection, the constants *Set* and *Prop* are called *sorts*, which can be viewed as types of types. In the grammar, x_i for all $i \in \mathbb{N}$ denotes a countable collection of variables. Application is denoted by juxtaposition of terms. It is a binary operator that associates to the left (e.g. we write $(t_1\ t_2)\ t_3$ as $t_1\ t_2\ t_3$). Terms can also be λ-abstractions $\lambda x : t_1.t_2$, where x is a variable that is considered *bound* in t_2, and t_1 is considered the type of variable x. It is also possible for terms to be universal quantifications where x is again a variable of type t_1 bound in t_2. If x does not occur in t_2, then we can write this quantification as $t_1 \to t_2$. The final construction in the term grammar above is for terms denoting the definition of variable x to be t_1 of type t_2 locally bound in t_3. We will sometimes use parentheses in the binders for abstractions and quantification, writing $\lambda(x : t_1).t_2$ and $\forall(x : t_1), t_2$ to make it easier for the reader to parse these expressions.

Rules assigning types to terms will be discussed below. In CoC, there is no syntactic difference between terms and types. We will use "term" and "type" interchangeably according to what is most reasonable for the current discussion.

CoC can be used both as a theorem proving system and a functional programming language. Its type system allows for a correspondence to be observed between theorem statements and types, and between proofs and terms. This is called the Curry-Howard correspondence [12] and allows us to view proofs as programs. By the Curry-Howard correspondence, the arrow notation can be understood simultaneously as implication or the function type arrow, depending on what is appropriate for the topic under consideration. It associates to the right, so we will usually write the type $t_1 \to (t_2 \to t_3)$ as $t_1 \to t_2 \to t_3$.

In the rest of this chapter, we will write t, T, u, or U for terms and types, possibly with subscripts. We write x for variables, also possibly with subscripts.

2.1.2 Judgments

Once we are able to build terms of CoC, we want to reason about them, possibly within some context of assumptions.

Definition 2.1. ***Context***
A *context* in CoC is a list of *variable declarations*, written $x : t$ to say variable x has

type t, and *definitions*, written $x := t_1 : t_2$ to say variable x has value t_1 of type t_2. The context may be written as $[d_1; d_2; \ldots]$ to list the elements. We write $[\,]$ for the empty context and $\Gamma :: (t_1 : t_2)$ for adding an element to the end of the list.

The description of contexts in CoC in [24] includes a global environment, written E, along with the local context, written Γ. Both are lists of variable declarations and definitions. In the presentation here we do not need to distinguish between global and local assumptions, so we will have one context, usually written Γ.

Definition 2.2. ***Sequent, Judgment***
A *sequent* $\Gamma \vdash t : T$ is a *judgment* where Γ is the context and t and T are CoC terms. We call the elements of the context *antecedents* (or *assumptions* or *hypotheses*) and $t : T$ is said to be a *consequent*. We write $\vdash t : T$ as notation for $[\,] \vdash t : T$.

A sequent is notation representing a conditional assertion which may be true or false. We want to be able to determine when such assertions hold. For this, we need a set of inference rules to determine when a sequent is *provable*. The rules of CoC are in Figure 2.1.

Definition 2.3. ***Derivation, Valid/Provable Sequent***
A tree built using the rules of Figure 2.1 with $\Gamma \vdash t : T$ at the root and *Ax-Prop*, *Ax-Set*, *Ax-Type*, or *Var* at the leaves is a *derivation* of $\Gamma \vdash t : T$. If there is a derivation of $\Gamma \vdash t : T$, we say this sequent is *valid* or *provable*. We also say that t has type T in Γ or just t has type T when Γ is empty.

For terms t and T and variable x, the notation $T\{x/t\}$ denotes *substitution*, meaning the operation that replaces occurrences of x in T with t, with the usual renaming of bound variables to avoid instances of free variables becoming bound. The rules *Ax-Prop*, *Ax-Set*, and *Ax-Type* are axioms that build the hierarchy of the sorts into the logic. The *Var* rule allows a branch of a derivation to be completed by showing the consequent of a sequent to be present in the context. Here we have omitted a premise requiring that the context is well-formed and the rules for building well-formed contexts (see [24] for details); informally, we understand this to mean that additions to the context are well-typed according to the rules of Figure 2.1. The *Lam* and *App* rules are the standard rules for building terms of the typed λ-calculus.

Definition 2.4. ***Dependent Product***
A term of the form $\forall t : T, U$ is called a *dependent product*.

Notice the *Prod* rules are all for building dependent products. The notation $t_1 \leq_{\beta\delta\iota\zeta} t_2$ in rule *Conv* is to say t_1 is a subtype of t_2 and will be described in

$$\frac{}{\Gamma \vdash Prop : Type_0}\ Ax-Prop \qquad \frac{}{\Gamma \vdash Set : Type_0}\ Ax-Set$$

$$\frac{}{\Gamma \vdash Type_i : Type_{i+1}}\ Ax-Type \qquad \frac{x : T \in \Gamma \quad (\text{or } x := t : T \in \Gamma \text{ for some } t)}{\Gamma \vdash x : T}\ Var$$

$$\frac{\Gamma \vdash \forall (x:T), U : s \quad \Gamma :: (x:T) \vdash t : U}{\Gamma \vdash \lambda (x:T).t : \forall (x:T), U}\ Lam$$

$$\frac{\Gamma \vdash t : \forall (x:U), T \quad \Gamma \vdash u : U}{\Gamma \vdash (t\ u) : T\{x/u\}}\ App$$

$$\frac{\Gamma \vdash t : T \quad \Gamma :: (x := t : T) \vdash u : U}{\Gamma \vdash let\ x := t : T\ in\ u : U\{x/t\}}\ Let$$

$$\frac{\Gamma \vdash T : s \quad s \in \{Set, Prop, Type_i\} \quad \Gamma :: (x:T) \vdash U : Prop}{\Gamma \vdash \forall (x:T), U : Prop}\ Prod-Prop$$

$$\frac{\Gamma \vdash T : s \quad s \in \{Set, Prop\} \quad \Gamma :: (x:T) \vdash U : Set}{\Gamma \vdash \forall (x:T), U : Set}\ Prod-Set$$

$$\frac{\Gamma \vdash T : Type_i \quad i \leq k \quad \Gamma :: (x:T) \vdash U : Type_j \quad j \leq k}{\Gamma \vdash \forall (x:T), U : Type_k}\ Prod-Type$$

$$\frac{\Gamma \vdash U : s \quad s \in \{Set, Prop, Type_i\} \quad \Gamma \vdash t : T \quad \Gamma \vdash T \leq_{\beta\delta\iota\zeta} U}{\Gamma \vdash t : U}\ Conv$$

Figure 2.1: Rules of CoC

Section 2.3 when discussing convertibility. The various *Prod* rules, together with the *Conv* rule, are what allow CoC to have such an expressive type system; they allow simple types, dependent types, polymorphism, and higher-order types as we will see in the coming sections.

Definition 2.5. ***Inhabited-in-Context***
Let Γ be a context and T a type. We say that T is *inhabited in context* Γ if there exists t such that $\Gamma \vdash t : T$ is provable.

The type system of CoC allows for two different approaches to be taken when using the language; it can be used as a functional programming language or for formalized reasoning via the two sorts *Set* and *Prop*, respectively.

Definition 2.6. ***Specification, Realization***
If $\Gamma \vdash T : \mathit{Set}$ is provable, then T is a *specification*. If $\Gamma \vdash t : T$ is provable, then t is a *realization* of the specification T.

We can think of a specification as the type of a function and a realization as its implementation. For example, the identity function on natural numbers has specification $\mathbb{N} \to \mathbb{N}$ and a realization of this specification is the function $\lambda(x : \mathbb{N}) \,.\, x$. Note that we have not yet defined $\mathbb{N}$ in CoC (see below).

Definition 2.7. ***Formula, Proof Object***
If $\Gamma \vdash T : \mathit{Prop}$ is provable, then T is a *formula*. If $\Gamma \vdash t : T$ is provable, then T is a *theorem* and t is a *proof object* representing a proof of theorem T.

In the next few sections, as we explore the expressive power of the CoC type system, we will encounter some examples that make use of the type $\mathbb{N}$ of natural numbers. This type is inductive and cannot be properly defined until Section 2.7, but it is useful in illustrating earlier concepts. For this reason an informal definition of this type is given here as well as some (later justified) results about it.

The type $\mathbb{N}$ is an inductive type whose "elements" are constructed from the following rules:

- the number 0 has type $\mathbb{N}$
- if n has type $\mathbb{N}$, than the successor of n (written $S\ n$) has type $\mathbb{N}$

In addition, in the examples here we make use of the fact that for any context Γ, the sequent $\Gamma \vdash \mathbb{N} : \mathit{Set}$ is provable.

2.2 Simply Typed Lambda Calculus

From the rules of Figure 2.1, we can see that CoC encompasses the simply-typed λ-calculus. These rules allow the construction of simple types. This includes atomic types, referred to by their identifier (e.g. $\mathbb{N}$, $\mathbb{Z}$), and arrow types $A \to B$ where A and B are simple types and $\to$ associates to the right. Observe that by the Curry-Howard correspondence we may view $A \to B$ as either a specification (function type) or a theorem (implication), depending on the sort of the arrow type. In either case, if $t : A \to B$, then t maps either data of type A, or proofs of A, to data of type B, or proofs of B, respectively.

Abstractions and applications for simple types are build using the rules *Lam* and *App* of Figure 2.1, where the bound variable in any universal quantification does not occur in its body. The rule *Lam* makes it possible to construct a term of type $A \to B$, but we also need to be able to build this type. This is accomplished via the product rule for simple types:

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s} \; \textit{Prod-ST}$$

where $s \in \{\textit{Set}, \textit{Prop}\}$.

Notice the rule *Prod-ST* is an instance of rule *Prod-Set* with $s = \textit{Set}$ or *Prod-Prop* with $s = \textit{Prop}$. Again note that $A \to B$ is shorthand for $\forall (x : A), B$ (given x does not occur in B).

2.3 Reductions

The calculus of constructions has the strong normalization property so all terms of CoC will be reduced to an irreducible form by any sequence of reductions. We use the notation $t \rhd_{\beta\delta\iota\zeta} s$ to say that a term t evaluates to a term s by some sequence of δ-, β-, ι-, and ζ-reductions (described below).

δ-Reduction replaces an identifier with its definition. For example, if we have defined $f := \lambda(x : \textit{Type}_0) . x : \textit{Type}_0 \to \textit{Type}_0$, then $f \; \textit{Set} \rhd_\delta \textit{Set}$.

β-Reduction evaluates a term acquired by the *App* rule by replacing all occurrences of the bound variable in the body of the abstraction with the term the abstraction is applied to using standard substitution rules; defined as $(\lambda(x : T) . t) \; u \rhd_\beta t\{x/u\}$.

ι-Reduction handles computations in recursive programs; it will not be used in the examples presented.

1. if $\Gamma \vdash t =_{\beta\delta\iota\zeta} u$ then $\Gamma \vdash t \leq_{\beta\delta\iota\zeta} u$
2. for all $i, j \in \mathbb{N}$, if $i \leq j$ then $\Gamma \vdash \mathit{Type}_i \leq_{\beta\delta\iota\zeta} \mathit{Type}_j$
3. for all $i \in \mathbb{N}$, $\Gamma \vdash \mathit{Set} \leq_{\beta\delta\iota\zeta} \mathit{Type}_i$
4. $\Gamma \vdash \mathit{Prop} \leq_{\beta\delta\iota\zeta} \mathit{Set}$ and for all $i \in \mathbb{N}$, $\Gamma \vdash \mathit{Prop} \leq_{\beta\delta\iota\zeta} \mathit{Type}_i$
5. if $\Gamma \vdash T =_{\beta\delta\iota\zeta} U$ and $\Gamma :: (x : T) \vdash T' \leq_{\beta\delta\iota\zeta} U'$ then $\Gamma \vdash \forall x : T, T' \leq_{\beta\delta\iota\zeta} \forall x : U, U'$

Figure 2.2: Subtyping in CoC

ζ-Reduction deals with converting local bindings; it will not be used in the examples presented.

2.3.1 Convertibility

We say two terms t_1 and t_2 are α-equivalent, written $t_1 \cong_\alpha t_2$, if they are the same term up to renaming of bound variables.

Definition 2.8. ***$\beta\delta\iota\zeta$-Convertible***
Two terms are considered equivalent, or *$\beta\delta\iota\zeta$-convertible*, if they can be reduced to α-equivalent terms by the reductions given above. When terms t_1 and t_2 are $\beta\delta\iota\zeta$-convertible, we write $t_1 =_{\beta\delta\iota\zeta} t_2$.

Symbolically, definition 2.8 says for terms t_1, t_2, u_1, and u_2, if $t_1 \triangleright_{\beta\delta\iota\zeta} u_1$ and $t_2 \triangleright_{\beta\delta\iota\zeta} u_2$ and $u_1 \cong_\alpha u_2$, then $t_1 =_{\beta\delta\iota\zeta} t_2$.

From convertibility we can develop the notion of *subtyping* in CoC; $\leq_{\beta\delta\iota\zeta}$ is a preorder on the collection of types in the type hierarchy. This relation is defined inductively in Figure 2.2. This gives us a conversion rule for terms, the rule *Conv* of Figure 2.1.

2.4 Dependent Types

A dependent type is the result of applying a function to appropriate expressions; in particular, it is the reduced form of a function applied to an argument of type *Set* or *Prop*. By definitions 2.6 and 2.7, a dependent type is a term of CoC that depends on a choice of a *realization* of a specification (for parametric types) or a choice of *proof object* (in the logical case).

Example 2.1. ***Tuple as a Dependent Type***
Let n be a term of type $\mathbb{N}$. By definition 2.6, n is a *realization* of the *specification* $\mathbb{N}$. The type of tuples of size n, call this *tuple*, depends on the value of n and has type $\mathbb{N} \to Set$. *tuple* 1 is a dependent type of one-tuples.

Note that the type of a dependent type is a dependent product (see definition 2.4). We need to be able to build the dependent products that will allow us to define the above example. We can use a derived rule, which we will call *Prod-Dep*, for building dependent products that can then be used to build dependent types. Let $s \in \{Set, Prop\}$. This rule is:

$$\frac{\Gamma \vdash T : s \quad \Gamma :: (x : T) \vdash U : Type_i}{\Gamma \vdash \forall(x : T), U : Type_i} \; \textit{Prod-Dep}$$

We get this rule from the following derivation tree:

$$\frac{\dfrac{\dfrac{}{\Gamma \vdash Type_0 : Type_1}\,\textit{Ax-Type} \quad \Gamma \vdash T : s \quad \dfrac{}{\Gamma \vdash s \leq_{\beta\delta\iota\zeta} Type_0}\,*}{\Gamma \vdash T : Type_0}\,\textit{Conv} \quad \Gamma :: (x : T) \vdash U : Type_i}{\Gamma \vdash \forall(x : T), U : Type_i} \; \textit{Prod-Type}$$

The leaf labeled with $*$ is proven by clauses 3 and 4 in the definition of the subtype relation in Figure 2.2.

Using *Prod-Dep* it is possible to build a term $\forall(x : T), U$ of type $Type_i$, where x may occur freely in U and, most importantly for the contents of this section, $T : s$ where $s \in \{Set, Prop\}$.

Example 2.2. ***Parametrized Types***
If s is Set and U is Set, then the rule *Prod-Dep* can be used to construct the type of parametrized types. Consider the first example above, the type of tuples of size n, where n is either a variable in the context or a concrete value. We will define the name of this type to be *tuple*. Then the type of *tuple* is $\mathbb{N} \to Set$ and the dependent type is an instantiation of this type with some value n of type $\mathbb{N}$. The sequent $\Gamma \vdash \mathbb{N} \to Set : Type_i$ is proven with *Prod-Type*.

Example 2.3. ***Predicates***
Let s be Set and be U be $Prop$. Then the rule *Prod-Dep* can be used to build the type of unary predicates $T \to Prop$ where (from the rule) we also know that T has type Set. We can extend this to n-ary predicates by repeated uses of the rule.

2.5 Higher-Order Types

For all $i \in \mathbb{N}$, types T with type $Type_i$ are considered here to be higher-order types since elements t of type T are types. For example, $\mathbb{N}$ has type Set, which has type

$Type_0$. Traditionally we can think of $\mathbb{N}$ as a type and elements that inhabit it as values of that type. Dependent products with quantification over higher-order types are built with the rule *Prod-Type* of Figure 2.1. In this section we will see how polymorphism and type operators are permitted in Coq using the *Prod-Type* rule and higher-order types.

2.5.1 Polymorphism

Informally, a polymorphic function is a function with a type parameter. Working toward an example of such a function, consider a unary function *double* on natural numbers that returns the argument multiplied by two. Note that we do not define this function here as this requires concepts explained later, and the focus of this discussion is on the type of such a function. *double* has type $\mathbb{N} \to \mathbb{N}$. In fact, all unary functions on natural numbers can be specified with type $\mathbb{N} \to \mathbb{N}$. Then a function that iterates such unary functions on $\mathbb{N}$, call this *iter_nat*, will have type $(\mathbb{N} \to \mathbb{N}) \to \underline{\mathbb{N}} \to \mathbb{N} \to \mathbb{N}$, where the first argument is the function to iterate and the second argument (which is underlined) is the number of times to iterate the function argument. The definition of *iter_nat* does not make use of the fact that the unary functions are over $\mathbb{N}$ because it simply repeatedly applies the function the number of times specified by the second (underlined) argument. So the logic of *nat_iter* should be reusable to iterate unary functions over any type t of type *Set*. We want to define a function, say *iter*, that will accomplish this. *iter* will have type $\forall (t : Set), (t \to t) \to \mathbb{N} \to t \to t$. We can show that this is a valid type in CoC.

Let T be *Set*, then we can use the *Prod-Type* rule with $i = j = k = 0$ to build specifications of polymorphic functions:

$$\frac{\Gamma \vdash Set : Type_0 \quad \Gamma :: (t : Set) \vdash U : Type_0}{\Gamma \vdash \forall (t : Set), U : Type_0} \; Prod\text{-}Type$$

Using this rule, we prove below that $\forall t : Set, (t \to t) \to \mathbb{N} \to t \to t$ has type $Type_0$ in CoC. In the course of this proof we will also show that $t : Set \vdash (t \to t) \to \mathbb{N} \to t \to t : Set$ is a valid sequent. By the terminology of definition 2.6, this means we can *specify* a function for the n-th iterate of a unary function on some type t with sort *Set*, where n is the natural number argument to the function *iter*.

Claim: $\vdash \forall t : Set, (t \to t) \to \mathbb{N} \to t \to t : Type_0$ is provable.
Proof:

This proof will be presented from axioms and work towards the goal where most steps use some version of a *Prod* rule. We show the sequent in the above paragraph holds as justification that we can construct the type of specifications of polymorphic

functions. To finish the proof and show the claim above we will need to use the *Conv* rule toward the end, since we are building a term of type $Type_0$.

For any Γ with $t : Set \in \Gamma$, the *Var* rule is used to show that the sequent $\Gamma \vdash t : Set$ is provable. So the following sequents are valid:

$$[t : Set] \vdash t : Set \tag{2.1}$$
$$[t : Set; t : Set] \vdash t : Set \tag{2.2}$$
$$[t : Set; H_1 : t \to t; H_2 : \mathbb{N}] \vdash t : Set \tag{2.3}$$
$$[t : Set; H_1 : t \to t; H_2 : \mathbb{N}; t : Set] \vdash t : Set \tag{2.4}$$

By (2.1) and (2.2) above and the rule *Prod-Set*, we derive the sequent

$$[t : Set] \vdash (t \to t) : Set. \tag{2.5}$$

The type $\mathbb{N}$ is defined to have type *Set*, so

$$[t : Set; H_1 : t \to t] \vdash \mathbb{N} : Set \tag{2.6}$$

is also valid. By (2.3) and (2.4) above and the rule *Prod-Set*, we derive the sequent

$$[t : Set; H_1 : t \to t; H_2 : \mathbb{N}] \vdash t \to t : Set. \tag{2.7}$$

So we can use the *Prod-Set* rule with (2.6) and (2.7) to show that

$$[t : Set; H_1 : t \to t] \vdash \mathbb{N} \to t \to t : Set \tag{2.8}$$

is provable. Now *Prod-Set* applied to (2.5) and (2.8) gives us

$$[t : Set] \vdash (t \to t) \to \mathbb{N} \to t \to t : Set. \tag{2.9}$$

We have shown in (2.9) that we can construct the specification of a polymorphic function for iterating unary functions of type $t : Set$. Continuing the proof, the sequent

$$[t : Set] \vdash Type_0 : Type_1 \tag{2.10}$$

is valid by *Ax-Type*. By the definition of $\leq_{\beta\delta\iota\zeta}$, the sequent

$$[t : Set] \vdash Set \leq_{\beta\delta\iota\zeta} Type_0 \tag{2.11}$$

is also valid. Applying *Conv* to (2.10), (2.9), and (2.11) gives

$$[t : Set] \vdash (t \to t) \to \mathbb{N} \to t \to t : Type_0. \tag{2.12}$$

By the axiom *Ax-Set*, the sequent

$$\vdash Set : Type_0 \tag{2.13}$$

is valid. Finally, we use the rule *Prod-Type* a final time with (2.13) and (2.12) to show that

$$\vdash \forall(t : Set), (t \to t) \to \mathbb{N} \to t \to t : Type_0 \tag{2.14}$$

is derivable, as claimed.

∎

2.5.2 Type Operators

Informally, a *type operator* is a type built from other types. The *Prod-Type* rule is what also allows us to express type operators in CoC because its conclusion is a typing judgment for a dependent product with quantification over higher-order types.

Example 2.4. ***Logical Connectives***
Let T be *Prop* and $i = j = k = 0$, then we have a rule to build the type of logical connectives.

$$\frac{\Gamma \vdash Prop : Type_0 \quad \Gamma :: (t : Prop) \vdash U : Type_0}{\Gamma \vdash \forall t : Prop, U : Type_0}\ \textit{Prod-Type}$$

The infix binary connectives representing "or" and "and" can be declared as $\vee : Prop \to Prop \to Prop$ and $\wedge : Prop \to Prop \to Prop$, respectively. Note that we can only *declare* these types at this time, meaning we see here how to construct the types of these operators. This is necessary before we see how to define (and derive the type of) definitions. Inductive types will be discussed in Section 2.7.

Claim: $\vdash Prop \to Prop \to Prop : Type_0$ is provable.
Proof:

This proof is illustrated by the derivation tree below. First, we rewrite the consequent of the sequent in the form that more easily visually matches the conclusion of the *Prod-Type* rule. Recall that the arrow notation $A \to B$ is a simplified notation for $\forall(x : A), B$. So we can rewrite $Prop \to Prop \to Prop$ as $\forall(t_1 : Prop), \forall(t_2 : Prop), Prop$.

$$\dfrac{\dfrac{}{\vdash Prop : Type_0}\ * \quad \dfrac{\dfrac{}{[t_1 : Prop] \vdash Prop : Type_0}\ * \quad \dfrac{}{[t_1 : Prop; t_2 : Prop] \vdash Prop : Type_0}\ *}{[t_1 : Prop] \vdash \forall(t_2 : Prop), Prop : Type_0}\ \textit{Prod-Type}}{\vdash \forall(t_1 : Prop), \forall(t_2 : Prop), Prop : Type_0}\ \textit{Prod-Type}$$

All three leaves are marked with * and proven by clause 4 in the definition of the subtype relation in Figure 2.2.

∎

2.6 Interactive Proving in Coq

For the remainder of this chapter we are considering the Coq implementation of CIC and use of this system. Now when we talk about built-in language types, tactics, commands, or define types in the Coq syntax, we will use `teletype` font rather than *italicized* math font.

As described in Section 2.1, to prove a statement P where $\Gamma \vdash P$: `Prop` is provable, we construct (or find) a proof object t through a derivation of $\Gamma \vdash t : P$ (i.e. according to definition 2.5, show that P is inhabited by t in Γ). By definition 2.7, proof object t represents a proof of theorem P. As an alternative to "defining" the proof object t and allowing the type checker to verify that t is a proof term for P, Coq provides an interactive proof mode where *tactics* are used to interactively build t. These proofs start with the theorem statement P as the goal and work backward reducing the goal to subgoals at each step and eventually to axioms. Once all goals have been discharged, the system builds the proof term t. The names of some of these tactics will be mentioned throughout this document, so we collect descriptions of the relevant tactics at the end of this section.

2.6.1 Proof State

The interactive proof engine of Coq can be used to build a derivation in CoC in a bottom-up fashion, meaning we construct the proof tree for a sequent $\Gamma \vdash t : P$ beginning at the root. In fact, we are constructing both the proof tree and t, showing that P is inhabited.

Definition 2.9. ***Proof State***
Let Γ be the context $[H_1 : P_1; \ldots ; H_k : P_k]$ and let P be a formula that we want to show is inhabited in Γ by a proof object. The pair (Γ, P) is a *proof state*. We call P a *goal*. We say that a proof state (Γ, P) is *complete* when there exists a t such that $t : P \in \Gamma$ or $P \ \triangleright_{\beta\delta\iota\zeta} \ \top$. A proof state that is not complete is *incomplete*.

Visually we will write a proof state in a vertical form with the assumptions in the context above a horizontal line and the goal below. For example:

$$\frac{\begin{array}{c} H_1 : P_1 \\ \vdots \\ H_k : P_k \end{array}}{P}$$

Unlike in Coq, when we have multiple incomplete proof states corresponding to leaves in a partial derivation and goals $G_1, \ldots, G_j$ have the same context of assumptions, we will write them all below the horizontal line, separated by commas.

$$\frac{\begin{array}{l} H_1 : P_1 \\ \vdots \\ H_k : P_k \end{array}}{G_1, \ldots, G_j}$$

We also sometimes refer to the goal as a subgoal as a reminder that it was acquired from a previous goal and there may be other subgoals.

Example 2.5. ***Interactive Proof***
To illustrate the Coq interactive theorem proving system, we will prove the conjunction elimination rule $\dfrac{P \wedge Q}{P} \wedge_{e_1}$. Note that in Coq, conjunction is defined as an inductive type. Since we look at inductive types and inductive reasoning in Coq in Section 2.7 and we do not have to use any inductive reasoning in this proof, we elide the details of inductive type definitions here. The single rule for constructing conjunctions is

$$\forall (P\ Q : \texttt{Prop}), P \to Q \to P \wedge Q$$

which says for all propositions P and Q, if we have a proof of P (i.e. a term of type P) and if we have a proof of Q (i.e. a term of type Q), then we have a proof of $P \wedge Q$.

Claim: $\vdash \forall (P\ Q : \texttt{Prop}), P \wedge Q \to P$

Proof: We begin this proof at the root of the proof tree. Initially the context of assumptions is empty.

$$\frac{}{\forall (P\ Q : \texttt{Prop}), P \wedge Q \to P}$$

We use the `intros` tactic, which applies the *Lam* rule of Figure 2.1 in a backward direction as many times as possible, effectively moving the quantified variable declarations (including anything to the left of $\rightarrow$) in the goal to the context of assumptions of the proof state. Recall that a goal corresponds to the type on the right of a colon in the consequent of a CoC sequent. The `intros` tactic automatically solves the left premise of each application of the *Lam* rule (it involves only simple type checking), and presents the type on the right of the colon in the consequent of the last right premise as the new subgoal. At each step of a proof in Coq, the terms on the left of the colon are constructed internally and not displayed. Once a proof is completed, these terms are used to build the proof object for the theorem we started with, which can then be displayed at the user's request. Each application of the *Lam* rule in a backward direction introduces a new hypothesis into the context of assumptions of the proof state.

$$\frac{\begin{array}{c} P : \texttt{Prop} \\ Q : \texttt{Prop} \\ H : P \wedge Q \end{array}}{P}$$

Now we use the `inversion` tactic on H. The inversion tactic exploits the properties of injectivity and disjointedness of the constructors of an inductive type. Since we have not yet explained inductive types, it suffices here to say by the definition of conjunction and given that assumption H is a witness of the conjunction $P \wedge Q$, it must be the case that we can also assume both P and Q.

$$\frac{\begin{array}{c} P : \texttt{Prop} \\ Q : \texttt{Prop} \\ H : P \wedge Q \\ H_1 : P \\ H_2 : Q \end{array}}{P}$$

Now the goal matches H_1 and we finish this proof with `assumption`, which is a tactic that simply applies the *Var* rule of Figure 2.1.

∎

2.6.2 Some Coq Tactics, Tacticals, and Commands

A tactical is an operator that takes tactic arguments to build a tactic. Below are Coq tactics, tacticals, and commands used in the proofs in this thesis. These have been described using some terminology introduced earlier as well as informal descriptions. More information can be found in the Coq Reference Manual [24].

`intros`

introduces variables and assumptions from the goal to the context of assumptions; a meta-level backward reasoning step of implication introduction; optional arguments assign names to the variables and assumptions introduced, otherwise default names are used

`apply`

used either for backward reasoning, also known as *backchaining*, on the goal, or forward reasoning, also known as *forward chaining*, on assumptions in the context; to backchain on the goal `G` over some `P : t1 -> t2` where `G` matches `t2` we write `apply P` and the new goal is `t1`; to forward chain with some `H : t` with `t` matching `t1` in the context over the same `P` we write `apply P in H` and assumption `H` is now `t2` (see the proofs in Chapters 6 and 8 for examples of use)

`constructor`

a specialized form of `apply` which applies an appropriate constructor for the type of the goal without naming the constructor; constructors are the names of the clauses of an inductive definition as will be described in Section 2.7; proofs using this tactic can be seen in Chapters 6 and 8

`reflexivity`

solves a goal when it is an equality with both sides $\beta\delta\iota\zeta$-equivalent (see definition 2.8)

`simpl`

applies $\beta\iota$-reduction then expands constants from their definitions and again tries $\beta\iota$-reduction; by default this tactic is used on the goal but it can be used on an element `H : t` in the context by writing `simpl in H` to simplify `t`

`rewrite`

rewrites from an equality (replacing all occurrences in the proof state of one side of the equality with the other) that is either an assumption, a local definition, or a theorem; optionally use either `<-` or `->` to give the rewrite direction

`inversion`

all conditions derived for each constructor of the type of the argument are new

assumptions; for each constructor matched, the proof has one new subgoal with the premises of that clause as new assumptions in the context (see example 2.5)

`induction`
: applies the appropriate induction principle for the type we induct over; see Section 2.7 for a discussion on induction in Coq

`assumption`
: used when the goal matches an assumption to complete the proof of a goal

`auto`
: attempts to prove the goal automatically using results in a hints database

`Hint`
: a Coq command; using `Hint Resolve` *theorem_name* adds *theorem_name* to a list of hints used by `auto`

`try`
: a tactical that tries to apply the tactic given as an argument and if it fails does not cause an error

`;`
: applies tactics in sequence

Many of the tactics can be replaced with the same tactic name prefixed with the letter *e* (e.g. `eapply`). This provides placeholders of appropriate type that act as logical variables that can be filled in by unification. They are used where we would otherwise need to provide a witness in cases of application as backward reasoning on the goal.

2.7 Induction in Coq

CIC extends CoC by adding inductive type definitions. An inductive type is a type with *constructors* that may take arguments of that type, so it is self-referential. For example, the natural number type $\mathbb{N}$ is defined inductively in Coq as:

```
Inductive nat : Set :=
| Z : nat
| S : nat -> nat.
```

In Coq, we declare that we are defining an inductive type with the keyword `Inductive`. The name of the type is `nat` and its type is `Set`. This inductive type has two *constructors*, `Z` (to represent zero) and `S` (to represent the successor function). We can understand this type as saying that any natural number can be constructed either as

$$
\begin{aligned}
&\texttt{nat_ind} : \forall (P : \texttt{nat} \to \texttt{Prop}),\\
&\quad (*\texttt{Z}*) \qquad P\ \texttt{Z} \to\\
&\quad (*\texttt{S}*) \qquad (\forall (m : \texttt{nat}), P\ m \to P\ (\texttt{S}\ m)) \to\\
&\qquad\quad \forall (n : \texttt{nat}), P\ n
\end{aligned}
$$

Figure 2.3: Induction principle for type `nat`

zero or the successor of some other natural number and these are the only two ways to construct natural numbers.

From an inductive type, Coq automatically generates an *induction principle* whose target type is `Prop`. To prove a property of all elements of a type, proofs using these induction principles have one subcase for each constructor of the type. The induction principle for `nat` is in Figure 2.3, where P is the property to be proven of all natural numbers. We sometimes refer to P as the *induction property*.

Constructors that have recursive occurrences of the type being defined will have corresponding induction subcases with *induction hypotheses*. An example of such a constructor is `S` in the definition of `nat` because it requires a `nat` argument. Notice the corresponding induction subcase is to prove $\forall (m : \texttt{nat}), P\ m \to P\ (\texttt{S}\ m)$. The formula $P\ m$ is an induction hypothesis.

Now that we have an inductive type defined in Coq, namely `nat`, we can define functions by primitive recursion over this inductive type. This is done using the `match`...`with`...`end` construction. To define a recursive function over n of type `nat`, we need to consider the possible constructions of n. From the definition of `nat`, we see that either $n = \texttt{Z}$ or $n = \texttt{S}\,m$ where m has type `nat`. We need to decide what happens in either case. Using the Coq syntax, we can write a primitive recursive function over n as:

```
Fixpoint recursive_nat (n : nat) :=
  match n with
  | Z => f1
  | S m => f2
  end.
```

where `Fixpoint` is a keyword for defining recursive functions and `recursive_nat` is the function name. If `n` evaluates to `Z`, then the result is `f1`. If `n` evaluates to `S m'` for some `m'` of type `nat`, then the result is `f2` with `m` replaced by `m'` by ι-reduction.

Example 2.6. ***Proof by Induction***
We will see how to prove the statement $\forall (n : \texttt{nat}), n = n + \texttt{Z}$ by induction in Coq,

also pointing out the induction property used by the induction principle. This proof uses the definition of $+$, which is notation for `plus`, in reductions to irreducible terms.

Using the `match`...`with`...`end` construction described above, `plus n m` is defined recursively with cases on the structure of `n` as:

```
Fixpoint plus (n m : nat) :=
  match n with
  | Z => m
  | S n' => S (plus n' m)
  end.
```

We write $\mathtt{n} + \mathtt{m}$ as infix notation for `plus n m`

Claim: $\vdash \forall (n : \mathtt{nat}), n = n + \mathtt{Z}$

Proof:

This proof is completed bottom-up, so the initial proof state is the node at the root of the proof tree. The context is empty and the goal is the statement that we wish to prove.

$$\frac{}{\forall (n : \mathtt{nat}), n = n + \mathtt{Z}}$$

The tactic `induction` n is used to backchain with the induction principle for natural numbers in Figure 2.3. This proof has two subcases, one corresponding to each constructor of `nat`. These are to prove $P\ \mathtt{Z}$ and $\forall (m : \mathtt{nat}), P\ m \to P\ (\mathtt{S}\ m)$ where

$$P \coloneqq \lambda (n : \mathtt{nat})\ .\ n = n + \mathtt{Z}$$

is the induction property.

We will first prove $P\ \mathtt{Z}$ (usually called the “base case”):

$$\frac{}{\mathtt{Z} = \mathtt{Z} + \mathtt{Z}}$$

This is done by reducing $\mathtt{Z} + \mathtt{Z}$ to $\mathtt{Z}$ by the first branch in the definition of $+$ and then with `reflexivity`.

To complete this proof we need to show $\forall (m : \mathtt{nat}), P\ m \to P\ (\mathtt{S}\ m)$ (the “inductive step”):

$$\frac{}{\forall (m : \mathtt{nat}), m = m + \mathtt{Z} \to \mathtt{S}\ m = (\mathtt{S}\ m) + \mathtt{Z}}$$

We make introductions into the context with `intros`.

$$\begin{array}{l} m : \texttt{nat} \\ H : m = m + \texttt{Z} \\ \hline \texttt{S}\ m = (\texttt{S}\ m) + \texttt{Z} \end{array}$$

The right side of the goal equality can be reduced by `simpl`, using the second branch in the definition of $+$.

$$\begin{array}{l} m : \texttt{nat} \\ H : m = m + \texttt{Z} \\ \hline \texttt{S}\ m = \texttt{S}\ (m + \texttt{Z}) \end{array}$$

Now we can use `rewrite <- H` to replace $m + \texttt{Z}$ with m on the right side of the goal equality.

$$\begin{array}{l} m : \texttt{nat} \\ H : m = m + \texttt{Z} \\ \hline \texttt{S}\ m = \texttt{S}\ m \end{array}$$

The goal is an equality with both sides equal, so this proof is finished with `reflexivity`. ∎

2.7.1 Mutually Inductive Types

A type may be built using types that are already defined. When two types have dependencies on each other, they cannot both be defined before the other. In this case we define a mutually inductive type. An example of where this is useful is in defining two types `even` and `odd` which are unary relations to identify even and odd natural numbers, respectively. In Coq these can be defined as:

```
Inductive even : nat -> Prop :=
| e_Z : even Z
| e_S : forall (n : nat), odd n -> even (S n)
with odd : nat -> Prop :=
| o_S : forall (n : nat), even n -> odd (S n).
```

Intuitively this says that Z (meaning zero) is even and the successor of any odd number is even. Also, the successor of any even number is odd.

Coq automatically generates an induction principle for each of these types. For `even`, this is

$$\begin{aligned}
&\texttt{even_ind} : \forall (P : \texttt{nat} \to \texttt{Prop}),\\
&\quad (*\texttt{e_Z}*) \qquad (P\ \texttt{Z}) \to\\
&\quad (*\texttt{e_S}*) \qquad (\forall (n : \texttt{nat}), \texttt{odd}\ n \to P\ (S\ n)) \to\\
&\qquad\qquad \forall (n : \texttt{nat}), \texttt{even}\ n \to P\ n
\end{aligned}$$

where P is the induction property for even natural numbers. Notice that a proof using this induction principle will have one subcase for each constructor of `even`. Also, in the case corresponding to the constructor `e_S`, there is no induction hypothesis about the premise `odd` n. So for some types and some theorems to prove, the generated induction principle is insufficient.

The command `Scheme` may be used to generate induction principles over mutually inductive types. These induction principles will have subcases for every constructor in every type in the mutually inductive type. Continuing the example above, we can get the following mutual induction principle over `even`:

$$\begin{aligned}
&\texttt{even_mutind} : \forall (P_1\ P_2 : \texttt{nat} \to \texttt{Prop}),\\
&\quad (*\texttt{e_Z}*) \qquad (P_1\ \texttt{Z}) \to\\
&\quad (*\texttt{e_S}*) \qquad (\forall (n : \texttt{nat}), \texttt{odd}\ n \to P_2\ n \to P_1\ (S\ n)) \to\\
&\quad (*\texttt{o_S}*) \qquad (\forall (n : \texttt{nat}), \texttt{even}\ n \to P_1\ n \to P_2\ (S\ n)) \to\\
&\qquad\qquad \forall (n : \texttt{nat}), \texttt{even}\ n \to P_1\ n
\end{aligned}$$

This induction principle has more cases but provides more powerful assumptions in each inductive case. Notice that now the subcase corresponding to the constructor `e_S` also has the induction hypothesis $P_2\ n$ and we also have a subcase for the constructor `o_S`.

2.8 Conclusion

The CoC inference system implements a type checker, so it can be used both for proof verification and checking that a function satisfies its specification (i.e. it is a realization of the required type). It can also be used to construct a proof. By appropriately instantiating the *Prod* rule, the system is made more expressive as a functional language and theorem proving system while still maintaining many desirable properties including strong normalization and consistency.

Coq can be used to prove formalized statements. A large library of tactics are available to assist in proof development. Since Coq is an implementation of CIC, it is

possible to define inductive types and then prove statements by induction over these types using automatically generated induction principles.

The upcoming presentation will make use of all of the concepts just presented: the rich type system of CIC, interactive proofs, and inductive types, culminating in proofs by structural induction over mutually inductive dependent types.

Chapter 3

Hybrid

In this chapter each layer of Hybrid will be explored to provide more intuition on how it is constructed and used. This explanation will be driven by an analogy, for use as an aid to both memory and understanding of the system.

The orientation of the layers is as in Figure 3.1. We will first consider the top layer, the object logic, in Section 3.1 with an example to motivate what we are trying to accomplish. Next we will consider each layer bottom-up, beginning with the ambient logic in Section 3.2, then the higher-order abstract syntax layer in Section 3.3. Continuing up the stack we next come to the specification logic. Since much of the work presented later is on the implementation and metatheory of the specification logic required for our motivating example of Section 3.1, we will not see details of the specification logic here. Rather, Section 3.4 will illustrate the benefit a specification logic adds to Hybrid and reinforce its necessity. This will be followed by another look at the object logic in Section 3.5, but this time we will be focusing on implementation details with the rest of the system in place. To conclude this chapter, Section 3.6 will

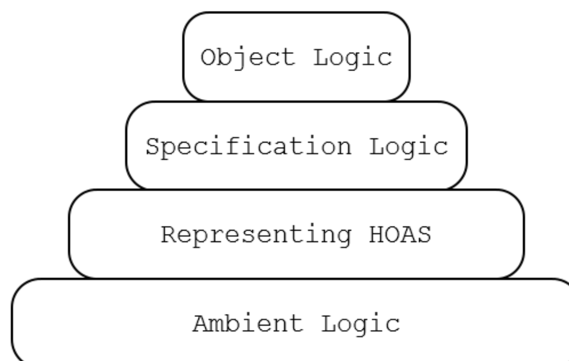


Figure 3.1: High-Level Hybrid Structure

compare Hybrid with alternative architectures for systems intended to reason about object logics using HOAS.

3.1 Object Logics

Suppose we wish to study flowers and create things with them. Then we need to be able to grow flowers.

Suppose we wish to prove something about a programming language or logic, the OL. This language will have rules expressing syntax and semantics that we need to encode in some proof assistant so that we can reason about it. It is also necessary to define the judgments of this language so that we can make claims about the OL.

Example 3.1. ***Object Logic: Equivalence of Named and Nameless λ-terms***
We consider one of the examples presented in [25]. Following the presentation there, we can define a syntax and rules expressing direct and de Bruijn representations of untyped λ-terms. By direct we mean the standard notation for λ-terms where abstractions reference a named variable that may be used in the body of the abstraction. De Bruijn indices [7] are a nameless representation of λ-terms where rather than using variable names, a natural number is used for occurrences of a variable.

Let n represent a natural number, x a variable, and e and d represent direct and de Bruijn representations, respectively. Then the following are grammars for these λ-terms:

$$e ::= x \mid \lambda x.e \mid e\ e$$
$$d ::= n \mid \lambda d \mid d\ d$$

A natural number n in the grammar for de Bruijn terms d serves as a pointer to the abstraction bounding that variable. This representation of λ-terms is more efficient for computation as we can avoid issues surrounding bound variable names. The λ-term $\lambda x.\lambda y.x\ y$ can be written using de Bruijn indices as $\lambda\ (\lambda\ (2\ 1))$. The number 2 refers to the outer binder (it is contained in two abstractions) and 1 refers to the inner binder.

An example property we might want to prove is that these two representations are equivalent (or seen another way, to construct equivalent λ-terms in these different forms). This logic has a judgment to say that λ-term e is equivalent to de Bruijn term d at depth n, written $e \equiv_n d$. There are three inference rules expressing equivalence of these two kinds of terms, one for each of application, abstraction, and variables, seen below.

$$\frac{\Gamma \vdash e_1 \equiv_n d_1 \quad \Gamma \vdash e_2 \equiv_n d_2}{\Gamma \vdash e_1\ e_2 \equiv_n d_1\ d_2}\ \mathit{hodb_app}$$

$$\frac{\Gamma, x \equiv_{n+k} k \vdash e \equiv_{n+1} d}{\Gamma \vdash \lambda x.e \equiv_n \lambda d} \; \textit{hodb_abs}$$

$$\frac{x \equiv_{n+k} k \in \Gamma}{\Gamma \vdash x \equiv_{n+k} k} \; \textit{hodb_var}$$

Applications in the two notations are considered equivalent under n abstractions if their corresponding components are. The rule *hodb_abs* is more complicated to understand due to an additional assumption in the context of the premise of the rule. Informally, this rule says if whenever assuming variable x is equivalent to index k at depth $n+k$ it can be shown that the bodies of the λ-terms e and d are equivalent at depth $n+1$, then we can conclude that the abstractions $\lambda x.e$ and λd are equivalent at depth n. As an illustration of how to use this system, we will see how to prove $\vdash \lambda x.\lambda y.x\ y \equiv_0 \lambda\ (\lambda\ (2\ 1))$ (i.e. these two λ-terms are equivalent under zero additional abstractions).

Claim: $\vdash \lambda x.\lambda y.x\ y \equiv_0 \lambda\ (\lambda\ (2\ 1))$
Proof:

Observe that by the *hodb_var* rule, both sequents below are provable.

$$x \equiv_2 2, y \equiv_2 1 \vdash x \equiv_2 2 \tag{3.1}$$

$$x \equiv_2 2, y \equiv_2 1 \vdash y \equiv_2 1 \tag{3.2}$$

This requires $n = 0$ and $k = 2$ in (3.1) and $n = k = 1$ in (3.2). Using the rule *hodb_app* with (3.1) and (3.2) we derive the sequent

$$x \equiv_2 2, y \equiv_2 1 \vdash x\ y \equiv_2 2\ 1 \tag{3.3}$$

Reviewing our claim, we are proving an equivalence of abstractions. The *hodb_abs* rule is used on (3.3) with $k = 1$.

$$x \equiv_2 2 \vdash \lambda y.x\ y \equiv_1 \lambda\ (2\ 1) \tag{3.4}$$

We apply *hodb_abs* again, this time with $k = 2$.

$$\vdash \lambda x.\lambda y.x\ y \equiv_0 \lambda\ (\lambda\ (2\ 1)) \tag{3.5}$$

We have derived the sequent claimed provable, so this proof is complete.

∎

Notice that the λ on the left of the equivalence in the conclusion of the rule *hodb_abs* is a binding operator. This observation will be important when we see how to represent untyped λ-terms using HOAS in Hybrid in Section 3.3 and then implement this OL in Section 3.5.

3.2 Ambient Logic

We can plant seeds in the ground and use the natural resources around us to reach our goal. The sun will provide energy and rain will give water.

The ambient logic (also known as the reasoning logic or the meta-meta-logic) is the layer of the system that everything else is defined in. It is an implementation of a logic and so has its own reasoning rules and allows us to define other reasoning systems within it. In our case, this is CIC and its implementation in Coq. This is the lowest reasoning level we consider carefully as part of our system; we will not be concerned with lower-level details of the implementation of CIC or its compilation. Chapter 2 covered all aspects of Coq, the ambient logic of Hybrid, that are necessary for understanding the contributions later in this thesis.

Existing theorem proving systems are an ideal tool to allow a language and its judgments to be encoded without building extra infrastructure. Hybrid is a Coq library (a collection of Coq files), so it is relatively easy to make modular updates to the system and to add new intermediate reasoning layers called specification logics, as will be explained in Section 3.4. Hybrid can also make use of the inductive and interactive reasoning tools of Coq as well as existing Coq libraries.

3.3 Representing Higher-Order Abstract Syntax in Hybrid

As our aspirations continue to grow, we find it difficult to scale up our flower production. When the rain doesn't fall as we require, we manually make up for the shortfall. The task of watering every plant every day is tedious. A dedicated plot of land with an organized arrangement and an irrigation system is a solution to this problem.

Many tedious computations are necessary for each encoding of an OL with binding structures. Examples include fresh name generation and capture-avoiding substitution. Since Hybrid is implemented in an ambient logic that is a typed λ-calculus, the technique of *higher-order abstract syntax* (HOAS) can be used for representing OL expressions. The idea is to use the binder of λ-calculus, function abstraction, to represent all OL binding operators. Using HOAS one can avoid implementing logic to reason about variable naming concepts, thus inheriting the meta-level solutions to these challenges. In addition, OL renaming and substitution are handled as meta-level α-conversion and β-reduction, respectively.

```
Inductive expr : Set :=
| CON : con -> expr
| VAR : var -> expr
| BND : bnd -> expr
| APP : expr -> expr -> expr
| ABS : expr -> expr.
```

Figure 3.2: Terms in Hybrid

At this level we have a type `expr` (see Figure 3.2) encoding a de Bruijn index version of the λ-calculus designed to be used to represent OL syntax. A parameter `con` is a placeholder for OL constants, to be defined for each OL. We define `var` and `bnd` to be the natural numbers. Hybrid expressions (`VAR` i) and (`BND` j) represent object-level free and bound variables, respectively. The constructor `APP` is used to build applications and `ABS` to build abstractions in de Bruijn notation.

Note that `con` is an implicit parameter in the environment it is defined in; uses outside of this environment must explicitly state this parameter (e.g. `expr con` instead of `expr`). A type to be given to this placeholder is defined for each OL. For example, the OL in Section 3.1 will have constants for application and abstraction for each kind of λ-term and a constant for variables in de Bruijn terms. These will be defined as an inductive type that is then used to instantiate the type `expr` for this particular OL. This example is implemented in Section 3.5.

Object-level binding operators are encoded in HOAS using the Hybrid operator `lambda` : (`expr con` $\rightarrow$ `expr con`) $\rightarrow$ `expr con` which is the meta-level binder defined in the Hybrid library. When using it to encode HOAS, the expanded definition is the underlying de Bruijn notation using only the constructors of `expr`. Although a Hybrid user never sees the expanded form and only works at the HOAS level. As an example, consider the untyped λ-term $(\lambda x.\lambda y.x\ y)$. We can represent this in Hybrid as (`lambda` (λx.(`lambda` ($\lambda y.x\ y$)))) which expands to `ABS` (`ABS` (`APP` (`BND` 1) (`BND` 0))). The `lambda` operator and the constructors of `expr` are used to encode OL syntax.

3.4 Specification Logic

Not all flowers will grow in the same conditions. Given any plot of land, there are many plants that will not grow there because they need specific nutrients in their soil. We can create different soil mixes depending on the needs of different classes of flowers.

There are OL judgments that we cannot encode as an inductive type in Coq. One

$$\frac{\Gamma, x : T \vdash E : T'}{\Gamma \vdash \lambda x \,.\, E : T \to T'}\ \textit{tp_abs}$$

Figure 3.3: Typing of λ-calculus Abstractions

example is a HOAS encoding of inference rules assigning simple types to λ-expressions. The standard rule for typing abstractions can be seen in Figure 3.3. Building on the example of Section 1.2, let `typ` be the type of OL types in the encoding in Coq. Let `arr` be a constant of type $\texttt{typ} \to \texttt{typ} \to \texttt{typ}$ representing arrow types. Recall `tm` is the type of OL terms. We want to define a typing predicate $\texttt{tp} : \texttt{tm} \to \texttt{typ} \to \texttt{Prop}$. Then the HOAS encoding of the rule for typing abstractions would be expressed as

$$\begin{aligned}&\forall (T\ T' : \texttt{typ})\ (E : \texttt{tm} \to \texttt{tm}),\\ &\qquad\qquad (\forall (x : \texttt{typ}), \underline{\texttt{tp}\ x\ T} \to \texttt{tp}\ (E\ x)\ T') \to \texttt{tp}\ (\texttt{lambda}\ E)\ (\texttt{arr}\ T\ T').\end{aligned}$$

Note that the `tp` predicate cannot be expressed inductively because of the (underlined) *negative occurrence* of the `tp` predicate in the above formula for the typing abstraction rule. Inductive types with negative recursive occurrences is not allowed by the Coq type system.

As a solution to the problem of needing to reason about judgments that violate this strict positivity requirement, Hybrid is a two-level system. By two-level we mean an intermediate specification level is introduced between the OL encoding and the meta-levels. The specification logic is less expressive than the ambient logic, the calculus of constructions, but it allows us to encode judgments with negative occurrences.

Hybrid is a Coq library and as mentioned earlier, this architectural decision makes quick prototyping of SLs possible. Another important benefit is that one can choose the simplest specification logic necessary for the present task, or possibly a combination of more than one depending on the OL to be encoded. Judgments that can be defined inductively do not need to be defined in a SL. This may simplify proofs of OL properties as the user can avoid using a more complicated logic than necessary.

The two levels of the OL and SL interact through a parameter of the SL,

$$\texttt{prog} : \texttt{atm} \to \texttt{oo} \to \texttt{Prop},$$

which is used to encode inference rules for OL judgments (and thus define provability at the OL level). There are two arguments to `prog`; the first is the (atomic) inference rule conclusion of type `atm` and the second a formula of type `oo` representing the premise(s) of the rule.

We use a for atoms with type `atm` and o for formulas of type `oo`, possibly with subscripts.

```
Inductive oo : Type :=
| atom : atm -> oo
| T : oo
| Conj : oo -> oo -> oo
| Imp : oo -> oo -> oo
| All : (expr con -> oo) -> oo
| Allx : (X -> oo) -> oo
| Some : (expr con -> oo) -> oo.
```

Figure 3.4: Type of SL Formulas

In this implementation, the type `atm` is a parameter of the SL and is instantiated with an inductive type whose constructors predicates expressing the judgments of a particular OL. For instance, the definition of `atm` for our above example might include a predicate `hodb` : (`expr con`) $\rightarrow$ `nat` $\rightarrow$ (`expr con`) $\rightarrow$ `atm` relating the higher-order and de Bruijn encodings at a given depth.

The type `oo` is the type of goals and clauses in the SL. The definition of `oo` for the SL defined later is in Figure 3.4. The constant `atom` coerces an atom (a predicate applied to its arguments) to an SL formula. For any α of type `atm`, we may refer to (`atom` α) as an atomic formula. The constructor `Conj` represents conjunction and `Imp` is used to build implications. Also note that in this implementation, we restrict the type of universal quantification to two types, (`expr con`) and `X`, where `X` is a parameter that can be instantiated with any primitive type; in our running example, `X` would become `nat` for the depth of binding in a de Bruijn term. We leave out disjunction. It is not difficult to extend our implementation to include disjunction and quantification (universal or existential) over other primitive types, but these have not been needed in reasoning about OLs.

We write $\langle\ a\ \rangle$, $(o_1\ \&\ o_2)$, and $(o_1 \longrightarrow o_2)$ as notation for (`atom` a), (`Conj` o_1 o_2), and (`Imp` o_1 o_2), respectively. Formulas quantified by `All` are written (`All` o) or (`All` $\lambda(x :$ `expr con`$)\ .\ o\ x$), where o has type `expr con` $\rightarrow$ `oo`. The latter is the η-long form with types included explicitly. The other quantifiers are treated similarly.

The type `oo` is an inductive type, so Coq will automatically generate the induction principle shown in Figure 3.5 as discussed in Section 2.7. We can use this induction principle to prove a statement of the form $\forall(o :$ `oo`$), P\ o$ for some $P :$ `oo` $\rightarrow$ `Prop`. This proof will have one subcase for each constructor of `oo`.

A Hybrid SL is defined as an inductive type in Coq to encode a sequent calculus. Each rule of the sequent calculus is represented by a constructor of the inductive type. The constructor name is the rule name and the type arrow is used for implication from premises to conclusion. The context of the sequent is defined to behave as a set of elements of type `oo`. We write Γ or c for contexts.

Since we explore the SL and proofs of its structural properties in detail later when

$$
\begin{array}{rl}
\texttt{oo_ind} : \forall (P : \texttt{oo} \to \texttt{Prop}), & \\
(*\texttt{atom}*) & (\forall (a : \texttt{atm}), P(\langle\, a \,\rangle)) \to \\
(*\texttt{T}*) & (P\ \texttt{T}) \to \\
(*\texttt{Conj}*) & (\forall (o_1 : \texttt{oo}), P\ o_1 \to \forall (o_2 : \texttt{oo}), P\ o_2 \to P\ (o_1 \& o_2)) \to \\
(*\texttt{Imp}*) & (\forall (o_1 : \texttt{oo}), P\ o_1 \to \forall (o_2 : \texttt{oo}), P\ o_2 \to P\ (o_1 \longrightarrow o_2)) \to \\
(*\texttt{All}*) & (\forall (o : \texttt{expr con} \to \texttt{oo}), (\forall (e : \texttt{expr con}), P\ (o\ e)) \to P\ (\texttt{All}\ o)) \to \\
(*\texttt{Allx}*) & (\forall (o : \texttt{X} \to \texttt{oo}), (\forall (x : \texttt{X}), P\ (o\ x)) \to P\ (\texttt{Allx}\ o)) \to \\
(*\texttt{Some}*) & (\forall (o : \texttt{expr con} \to \texttt{oo}), (\forall (e : \texttt{expr con}), P\ (o\ e)) \to P\ (\texttt{Some}\ o)) \to \\
\forall (o : \texttt{oo}), P\ o &
\end{array}
$$

Figure 3.5: Induction Principle for `oo`

describing the contributions of this research, we cut short the discussion here. For continuity in this chapter, some notation and the meaning of provability judgments of the SL are all we need now. We write $\Gamma \rhd o$ to denote an SL, where Γ has type `context` and o has type `oo`. The symbol $\rhd$ is used as the SL sequent arrow.

3.5 Example OL Implementation

Now we can see how to encode our example syntax and judgments in Hybrid. Let `tm` represent the type of direct λ-terms and `dtm` represent the type of de Bruijn terms. Since these are used to form OL expressions, `tm` and `dtm` are aliases for `expr con`. Before stating the implementation of the rules of the logic, we have to define the OL constants. For direct application and abstraction we have `hApp` : `tm` $\to$ `tm` $\to$ `tm` and `hAbs` : (`tm` $\to$ `tm`) $\to$ `tm`, respectively. Direct variables are encoded as meta-level variables. For de Bruijn application, abstraction, and variables we have `dApp` : `dtm` $\to$ `dtm` $\to$ `dtm`, `dAbs` : `dtm` $\to$ `dtm`, and `dVar` : `nat` $\to$ `dtm`, respectively.

In Figure 3.6 the constants of the OL are defined in the inductive type `con`. We also have the definitions of OL applications and abstractions for the direct and de Bruijn forms of λ-terms in terms of the OL constants and HOAS application and `lambda` operator. Note that in Coq, `fun` is notation for abstractions. When we write Coq code we use this notation but when writing pretty-printed versions of the code we will use λ-calculus abstraction notation. For example, we often write Coq abstractions `fun x => f x` as $\lambda x.f\ x$ because the latter is often more readable in our discussions. In Figure 3.6 we can see the use of HOAS in the definition of `hAbs` where we use the Hybrid `lambda` operator.

The atomic judgment discussed for this example (equivalence between the two

```
Inductive con : Set :=
| hAPP : con
| hABS : con
| dAPP : con
| dABS : con
| dVAR : nat -> con.

Definition hApp : tm -> tm -> tm :=
  fun (e1 : tm) =>
    fun (e2 : tm) =>
      APP (APP (CON hAPP) e1) e2.
Definition hAbs : (tm -> tm) -> tm :=
  fun (f : tm -> tm) =>
    APP (CON hABS) (lambda f).

Definition dApp : dtm -> dtm -> dtm :=
  fun (d1 : dtm) =>
    fun (d2 : dtm) =>
      APP (APP (CON dAPP) d1) d2.
Definition dAbs : dtm -> dtm :=
  fun (d : dtm) =>
    APP (CON cdABS) d.
Definition dVar : nat -> dtm :=
  fun (n : nat) =>
    (CON (dVAR n)).
```

Figure 3.6: Example OL: Encoding Syntax in Hybrid

```
Inductive prog : atm -> oo atm (expr con) X -> Prop :=
| hobd_app : forall (e1 e2 : tm) (n : nat) (d1 d2 : dtm),
   prog (hodb (hApp e1 e2) n (dApp d1 d2))
    (<<hodb e1 n d1>> & <<hodb e2 n d2>>)
| hodb_abs : forall (f : tm -> tm) (n : nat) (d : dtm),
   abstr f ->
   prog (hodb (hAbs f) n (dAbs d))
    (All (fun (x : tm) =>
     (Allx (fun (k : X) => <<hodb x (n + k) (dVar k)>>)) --->
       <<hodb (f x) (n + 1) d>>)).
```

Figure 3.7: Example OL: Encoding OL Inference Rules *hodb_app* and *hodb_abs* in Hybrid

representations of lambda terms) is part of the inductive type `atm` defined below.

```
Inductive atm : Set :=
| hodb : tm -> nat -> dtm -> atm.
```

The predicate `hodb` corresponds to the infix $\equiv_n$ relation in the rules in Section 3.1 (i.e. `hodb` $e\ n\ d$ is notation for $e \equiv_n d$). In the environment where the SL is defined, there are parameters `atm` for atomic judgments of the OL, `con` for OL constants, and `X` for another type we wish to universally quantify over. Now the type of SL formulas with all parameters filled in is `oo atm con X`. This is the type of SL formulas at the OL level.

The rules shown in Section 3.1 can now be defined in Hybrid using HOAS and a SL. More specifically, we can now define the inductive type `prog` as shown in Figure 3.7, where we see the HOAS encoding of the rules in Section 3.1. The inductive type `prog` has a constructor for each of the inference rules *hodb_app* and *hodb_abs*. As we will see, the *hodb_var* rule is not represented explicitly because it is taken care of at the level of the SL. The first argument to `prog` is an atomic OL inference rule conclusion and the second argument is a formula to encode the premises of the same OL inference rule. The Coq notation for $\langle\ a\ \rangle$ is `<<a>>`.

An example theorem for this OL is to prove that the judgment `hodb` is deterministic in its first and third arguments (and thus the relational definition of the rules represents a function). To do this we want to prove the two theorems below (where $=$ is equality in the ambient logic).

Proposition 3.5.1 (`hodb_det1`).

$$\forall(\Gamma : \texttt{context})(e : \texttt{tm})(d_1\ d_2 : \texttt{dtm})(n : \texttt{nat}),$$
$$\Gamma \rhd \langle\ \texttt{hodb}\ e\ n\ d_1\ \rangle \to \Gamma \rhd \langle\ \texttt{hodb}\ e\ n\ d_2\ \rangle \to d_1 = d_2.$$

Proposition 3.5.2 (`hodb_det3`).

$$\forall(\Gamma : \texttt{context})(e_1\ e_2 : \texttt{tm})(d : \texttt{dtm})(n : \texttt{nat}),$$
$$\Gamma \rhd \langle\ \texttt{hodb}\ e_1\ n\ d\ \rangle \to \Gamma \rhd \langle\ \texttt{hodb}\ e_2\ n\ d\ \rangle \to e_1 = e_2.$$

To prove these in Hybrid, we must first define a SL that is able to reason about this OL. Once we have defined the SL, using it and the encoding of the OL just described, we will be ready to prove the above propositions. As of this writing, these theorems are not proven in Hybrid.

3.6 Comparison to Other Architectures

Our approach to growing flowers is not the only solution. One alternative is to build a factory specializing in the production of flowers. This would give us full control over lighting, water, and soil composition; but the startup costs are high and modifications can be prohibitively expensive.

Other systems use HOAS for encoding and reasoning about OLs with binders but different choices are made in the implementation of these systems. We will briefly look at the features of the two most closely related systems, Abella [11] and Beluga [22], and compare these systems to Hybrid. These three systems, along with Twelf [23], are compared in detail using benchmark problems in [10].

One feature that sets Hybrid apart from these systems is that Hybrid is a library in an existing theorem proving system while Abella, Beluga, and Twelf are special-purpose theorem proving systems built for reasoning about OLs using HOAS. Using Coq means we can trust the proofs without having to develop extra infrastructure. These proofs can be independently checked because a proof term is a λ-term; a proof check is a type check in the Calculus of Constructions, a trusted and well studied theoretical foundation for our work. The trade-off is less control over the reasoning logic of Hybrid and more levels of encoding.

Abella Abella is an interactive proof environment using the special-purpose $\mathcal{G}$ logic as its reasoning logic. $\mathcal{G}$ is intuitionistic, predicative, higher-order, and has fixed-point definitions for atomic predicates. It also allows mathematical induction (over natural numbers). Infrastructure for reasoning using HOAS is built-in to this logic. Like Hybrid, it is a two-level logical framework. In contrast, since it is a special-purpose system for reasoning about OLs, only one SL is used by the system at a time; to use a different SL the system must be updated. Hybrid is a Coq library so multiple SLs can can be available for use by any OL.

Abella is a tactic-based interactive theorem prover. This is the same style used when using the interactive proof environment in Coq, but the crucial difference is that on completion of a Coq proof the system generates a proof term. This is an object that can be checked independent of the implementation of CIC or Coq. This means that rather than trusting the implementation of a language and the tactics, we are provided evidence on completion of the proof. Since Hybrid is implemented in Coq, we have access to proof terms once a theorem is proven. This is not the case in Abella.

An advantage to Abella is that is has the ∇-quantifier, a new specialized quantifier providing better direct reasoning about binding in OLs. This allows Abella to prove some properties about OLs that cannot be proven in Hybrid until we implement ∇.

Beluga Beluga is also a logical framework for reasoning about OLs using HOAS. The reasoning logic in this system is contextual LF; it supports reasoning over contexts. It is more specialized for reasoning with HOAS than Hybrid is. It implements a type theory instead of a logic.

In Beluga, some metatheory about contexts (e.g. the structural rules of weakening, contraction, and exchange in sequent calculi) is implicit. This means that it is built-in to the implementation rather than being axioms of a logic or proven to be admissible as rules. The benefit of this choice is it is not necessary to prove theses structural rules. The argument against this is that it requires more trust from the user. It is necessary to trust the implementation of the system rather than being able to see how the rules are defined to be axiomatic or proven to be admissible.

Chapter 4

Hereditary Harrop Formulas

The logic of hereditary Harrop formulas is foundational in the theory of logic programming languages. Although these formulas have their origins in encoding search behaviour and extending the power of logic programming languages in a semantically clear way, we make use of them in this thesis for their role in restricting the structure of proofs. Using such a restricted logic as a specification logic in Hybrid simplifies SL metatheory proofs and proofs about object logics.

Section 4.1 will introduce the language of higher-order hereditary Harrop formulas and an inference system for reasoning about them. Following this, in Section 4.2 we will see how to modify this logic to one with *focusing*, which helps to optimize proof search as will be described later.

4.1 Higher-Order Hereditary Harrop Formulas

The terms of the logic defined here are the terms of the simply-typed λ-calculus. Types are built from the primitive types and the (right-associative) function arrow $\to$ as usual. We introduce a type o for formulas. Logical connectives and quantifiers are introduced as constants with their corresponding types as in [5]. For example, conjunction has type $o \to o \to o$ and the quantifiers have type $(\tau \to o) \to o$, with some restrictions on τ described below. Predicates are function symbols whose target type is o. Following [15], the grammars below for G (goals) and D (clauses) define the formulas of the higher-order hereditary Harrop language.

Definition 4.1. ***Higher-Order Hereditary Harrop Formulas***
Formulas built from the grammar for G are called G-formulas and formulas build from the grammar for D are called D-formulas or *higher-order hereditary Harrop formulas*.

$$G ::= \top \mid A \mid G \,\&\, G \mid G \vee G \mid D \longrightarrow G \mid \forall_\tau x.G \mid \exists_\tau x.G$$
$$D ::= A \mid G \longrightarrow D \mid D \,\&\, D \mid \forall_\tau x.D$$

We use the metavariable A (possibly with subscripts) for atoms and write & for conjunction, $\longrightarrow$ for (right-associative) implication, and $\vee$ for disjunction. For universal and existential quantification, written as usual with symbols $\forall$ and $\exists$, we include the subscript τ to explicitly state the domain of quantification. This may be left out when it can be inferred from context. In goal formulas, we restrict τ to be a primitive type not containing o. In clauses, τ also cannot contain o, and is either primitive or has the form $\tau_1 \to \tau_2$ where both τ_1 and τ_2 are primitive types.

With the language of formulas defined, we can now consider an inference system for reasoning about these formulas. This is a sequent calculus with the same conventions as described for CoC in Section 2.1. A grammar for contexts of these sequents is below. Contexts here are lists of hereditary Harrop formulas.

$$\Gamma ::= [\,] \mid \Gamma, D$$

The rules for this logic are in Figure 4.1. We use the same naming conventions as in the grammars of this chapter and also use x for bound variables, c for fresh variables, and t for terms.

The *init* rule allows a branch of a proof to be completed when an atom A on the right of the sequent is in the context Γ. The only other way to finish a branch of a proof is with the axiom $\top$ when the consequent of a sequent is $\top$. The remaining rules are standard left and right introduction rules for formulas built from the grammars for G and D. In the rule $\vee_{R_i}$, $i \in \{1, 2\}$.

Notice that all sequents have only one formula on the right of the sequent (as was also the case in the rules of CoC in Chapter 2). A derivation tree built using a set of rules with this property is called **M**-proofs to say that it is a proof in minimal logic (it is also an **I**-proof since it will also hold in intuitionistic logic, see [16]).

This logic has both left and right rules for each logical connective. In the course of a proof, a proof writer may have multiple decisions on how to proceed. This nondeterminism is not desirable if our goal is to automate proof search, which is the case for logic programming.

Example 4.1. ***Hereditary Harrop Derivations***
Consider the sequent $\forall_\tau x, (A_1\ x) \& (A_2\ x) \vdash (A_1\ t) \& (A_2\ t)$ where $t : \tau$. Below are two derivation trees for this sequent:

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ init \qquad \frac{}{\Gamma \vdash \top}\ \top_R$$

$$\frac{\Gamma, D_1, D_2 \vdash G}{\Gamma, D_1 \& D_2 \vdash G}\ \&_L \qquad \frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \& G_2}\ \&_R$$

$$\frac{\Gamma, D_1 \vdash G \quad \Gamma, D_2 \vdash G}{\Gamma, D_1 \vee D_2 \vdash G}\ \vee_L \qquad \frac{\Gamma \vdash G_i}{\Gamma \vdash G_1 \vee G_2}\ \vee_{R_i}$$

$$\frac{\Gamma \vdash G_1 \quad \Gamma, D \vdash G_2}{\Gamma, G_1 \longrightarrow D \vdash G_2}\ \longrightarrow_L \qquad \frac{\Gamma, D \vdash G}{\Gamma \vdash D \longrightarrow G}\ \longrightarrow_R$$

$$\frac{\Gamma, D\{x/t\} \vdash G}{\Gamma, \forall_\tau x, D \vdash G}\ \forall_L \qquad \frac{\Gamma \vdash G\{x/c\}}{\Gamma \vdash \forall_\tau x, G}\ \forall_R$$

$$\frac{\Gamma, D\{x/c\} \vdash G}{\Gamma, \exists_\tau x, D \vdash G}\ \exists_L \qquad \frac{\Gamma \vdash G\{x/t\}}{\Gamma \vdash \exists_\tau x, G}\ \exists_R$$

Figure 4.1: The logic of higher-order hereditary Harrop formulas

$$\dfrac{\dfrac{\dfrac{\dfrac{A_1\ t \in A_1\ t, A_2\ t}{A_1\ t, A_2\ t \vdash A_1\ t}\,init}{(A_1\ t)\&(A_2\ t) \vdash A_1\ t}\,\&_L \qquad \dfrac{\dfrac{A_2\ t \in A_1\ t, A_2\ t}{A_1\ t, A_2\ t \vdash A_2\ t}\,init}{(A_1\ t)\&(A_2\ t) \vdash A_2\ t}\,\&_L}{(A_1\ t)\&(A_2\ t) \vdash (A_1\ t)\&(A_2\ t)}\,\&_R}{\forall_\tau x, (A_1\ x)\&(A_2\ x) \vdash (A_1\ t)\&(A_2\ t)}\,\forall_L$$

In this first derivation, we alternate between uses of left and right rules until all leaves that are sequents with the goal on the right contained in the context.

$$\dfrac{\dfrac{\dfrac{\dfrac{A_1\ t \in A_1\ t, A_2\ t}{A_1\ t, A_2\ t \vdash A_1\ t}\,init}{(A_1\ t)\&(A_2\ t) \vdash A_1\ t}\,\&_L}{\forall_\tau x, (A_1\ x)\&(A_2\ x) \vdash A_1\ t}\,\forall_L \qquad \dfrac{\dfrac{\dfrac{A_2\ t \in A_1\ t, A_2\ t}{A_1\ t, A_2\ t \vdash A_2\ t}\,init}{(A_1\ t)\&(A_2\ t) \vdash A_2\ t}\,\&_L}{\forall_\tau x, (A_1\ x)\&(A_2\ x) \vdash A_2\ t}\,\forall_L}{\forall_\tau x, (A_1\ x)\&(A_2\ x) \vdash (A_1\ t)\&(A_2\ t)}\,\&_R$$

In this second derivation, beginning at the root we first use as many right rules as necessary to only have atoms on the right side of sequents. Then we apply left rules until we finish the proof as above. This derivation is an example of a *uniform proof*.

Definition 4.2. ***Uniform Proof***
A *uniform proof* is an **I**-proof where every sequent in the derivation tree that is non-atomic on the right is derived from the right introduction rule (e.g. $\&_R$, $\forall_R$, etc.) of its top-level connective.

We can see that the first derivation above is *not* a uniform proof, because the rule $\forall_L$ is used to derive a sequent that does not have an atom on the right. If we wish to allow only uniform proofs, then this does not restrict what is provable by the logic of Figure 4.1. The set of uniform proofs using rules in Figure 4.1 in a restricted manner is sound and complete with respect to the set of proofs that can be built using the same rules without this restriction.

Uniform proofs can be generalized to the notion of *focusing* as presented in [13] and [4], where the logic presented above is viewed as the negative fragment of intuitionistic logic.

4.2 Focusing

A proof search strategy that reduces nondeterminism will make it easier to add automation to proof search. Here we describe a strategy that divides proof search into two stages by augmenting the inference system in a way that reduces the number of rule choices available at each step.

$$\frac{\Gamma; [D] \vdash A \quad D \in \Gamma}{\Gamma \vdash A}\ focus \qquad \frac{A \in \Gamma}{\Gamma \vdash A}\ init$$

$$\frac{}{\Gamma; [A] \vdash A}\ match \qquad \frac{}{\Gamma \vdash \top}\ \top_R$$

$$\frac{\Gamma, [D_i] \vdash A}{\Gamma, [D_1 \& D_2] \vdash A}\ \&_{L_i} \qquad \frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \& G_2}\ \&_R$$

$$\frac{\Gamma, [D_1] \vdash A \quad \Gamma, [D_2] \vdash A}{\Gamma, [D_1 \vee D_2] \vdash A}\ \vee_L \qquad \frac{\Gamma \vdash G_i}{\Gamma \vdash G_1 \vee G_2}\ \vee_{R_i}$$

$$\frac{\Gamma \vdash G \quad \Gamma, [D] \vdash A}{\Gamma, [G \longrightarrow D] \vdash A}\ \longrightarrow_L \qquad \frac{\Gamma, D \vdash G}{\Gamma \vdash D \longrightarrow G}\ \longrightarrow_R$$

$$\frac{\Gamma, [D\{x/t\}] \vdash A}{\Gamma, [\forall_\tau x, D] \vdash A}\ \forall_L \qquad \frac{\Gamma \vdash G\{x/c\}}{\Gamma \vdash \forall_\tau x, G}\ \forall_R$$

$$\frac{\Gamma, [D\{x/c\}] \vdash A}{\Gamma, [\exists_\tau x, D] \vdash A}\ \exists_L \qquad \frac{\Gamma \vdash G\{x/t\}}{\Gamma \vdash \exists_\tau x, G}\ \exists_R$$

Figure 4.2: The logic of higher-order hereditary Harrop formulas with focusing

The idea is, in a bottom-up proof, we apply the appropriate right-rule for the top-level connective of the consequent of the sequent until the consequent is atomic. At this point we will transition to using left rules on a formula from the context; we choose a single formula from the context and apply left rules on this formula until this "focused" formula is atomic. If it matches the atom on the right of the sequent, the branch is complete, otherwise the proof fails and we return to the point where a formula from the context was focused.

To accomplish this, the logic of Figure 4.1 is extended to that in Figure 4.2. A sequent $\Gamma; [D] \vdash A$ is called a *focused sequent* and we call rules with a focused sequent conclusion *focused rules*. The interpretation is that D is a formula under focus. Here we use the same sequent arrow $\vdash$ for both kinds of sequents. Notice that the consequent of the sequent for all focused rules is an atom. This is because,

as stated above, we first apply the right-rules until the right side of the sequent is atomic. Missing from the above description was a method to ensure the left rules are applied to a single element of the context until it too is atomic. This is accomplished with the *hhfocus* rule, which acts as a gateway from the right rules to the focused rules. The focused rules appear to be similar to the left rules in Figure 4.1, but now we are focused on a particular formula and are not able to alternate between the elements in the context that we apply left rules to.

An additional optimization is achieved by requiring the right branch of the $\longrightarrow_R$ rule (a focused sequent) to be fully explored before the left (unfocused sequent). This way, once we are applying left rules, we continue to do so until we have reached a leaf with sequent $\Gamma; [A'] \vdash A$. If $A = A'$, then the branch is completed with *match* and we work on the left (unfocused sequent) branch of the last application of the $\longrightarrow_R$ rule. Otherwise, the branch cannot be completed and a different choice of formula must be focused at the start of the current sequence of focused rule applications. We will illustrate this by showing how we can write a focused derivation of the sequent in example 4.1.

Example 4.2. ***Hereditary Harrop Focused Derivation***
This proof is very similar to the second derivation of example 4.1, but now we use focused rules where there we used left rules and make use of the *focus* and *match* rules. See below for the derivation tree of $\forall x, (A_1\ x)\&(A_2\ x) \vdash (A_1\ t)\&(A_2\ t)$ using the focused sequent calculus of Figure 4.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall_\tau x, (A_1\ x)\&(A_2\ x); [A_1\ t] \vdash A_1\ t}\ match}{\forall_\tau x, (A_1\ x)\&(A_2\ x); [(A_1\ t)\&(A_2\ t)] \vdash A_1\ t}\ \&_{L_1}}{\forall_\tau x, (A_1\ x)\&(A_2\ x); [\forall_\tau x, (A_1\ x)\&(A_2\ x)] \vdash A_1\ t}\ \forall_L}{\forall_\tau x, (A_1\ x)\&(A_2\ x) \vdash A_1\ t}\ focus \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\forall_\tau x, (A_1\ x)\&(A_2\ x); [A_2\ t] \vdash A_2\ t}\ match}{\forall_\tau x, (A_1\ x)\&(A_2\ x); [(A_1\ t)\&(A_2\ t)] \vdash A_2\ t}\ \&_{L_2}}{\forall_\tau x, (A_1\ x)\&(A_2\ x); [\forall_\tau x, (A_1\ x)\&(A_2\ x)] \vdash A_2\ t}\ \forall_L}{\forall_\tau x, (A_1\ x)\&(A_2\ x) \vdash A_2\ t}\ focus}{\forall_\tau x, (A_1\ x)\&(A_2\ x) \vdash (A_1\ t)\&(A_2\ t)}\ \&_R$$

Notice that even though we use the *focus* rule to select a formula from the context to focus and used the focused rules to manipulate it, we still retain the original formula in the context.

Another important observation is that proofs using this focused sequent calculus are forced to be uniform proofs, because we cannot freely choose between applying left or right rules; the search strategy forces us to use the rule introducing the top-level connective of the principal formula of the sequent. This logic is also sound and complete with respect to the logic of Figure 4.1.

In the next chapter we will present a specification logic that is a slight modification of the sequent calculus of Figure 4.2. We modify this logic is because there are rules that are unnecessary for our application of hereditary Harrop formulas and

focusing. There are also some implementation details built in to the rules presented later, as will be explained in Chapter 5.

Part II

Contributions

Chapter 5

Specification Logic

The first stage of the contributions outlined in this thesis is defining a specification logic to increase the reasoning power of Hybrid. The new specification logic (SL) for Hybrid is based on hereditary Harrop formulas using an intuitionistic logic with focusing as described in Chapter 4. We adopt a modified version of the rules very close to the style of the rules of the specification logic used in the higher-order version of Abella [25]. We do not include any rules for disjunction here because they have not been necessary for object logics in case studies of interest. The SL could easily be extended to add these rules and the proofs of SL metatheory would have the same structure, as will be seen in Chapter 7.

We note that unlike in all previous SLs for Hybrid there is no restriction on the implicational complexity (see [8]), because G-formulas in higher-order hereditary Harrop language allow D-formulas as the antecedent of implication as was seen in Section 4.1. In all previous SLs, only atomic formulas were allowed in place of the more general D-formulas allowed here.

The SL presented in this chapter is a sequent calculus implemented as an inductive type in Coq. Section 5.1 describes how contexts are defined for this SL. Section 5.2 presents the Coq implementation of the SL based on hereditary Harrop formulas and we see how to prove properties of this SL by structural induction in Section 5.3.

In Appendix A, we list notations that will be used in the rest of the thesis.

5.1 Contexts in Coq

The type `context` represents contexts of assumptions in sequents and is defined using the Coq `ensemble` library as `ensemble oo` since we want contexts to behave as sets with elements of type `oo`. In proofs of some context lemmas stated below we use the

`ensemble` extensional equality axiom:

$$\texttt{Extensionality_Ensembles} : \forall (E_1\ E_2 : \texttt{ensemble}), (\texttt{Same_set}\ E_1\ E_2) \rightarrow E_1 = E_2$$

where `Same_set` is defined in the `Ensemble` library. We use $o \in c$ as notation for `elem` $o\ c$ which means formula o is an element of context c. Context subset, written $\Gamma_1 \subseteq \Gamma_2$, is defined as $\forall (o : \texttt{oo}), o \in \Gamma_1 \rightarrow o \in \Gamma_2$.

We write (Γ, β) as notation for $(\texttt{context_cons}\ \Gamma\ \beta)$. We write write c or Γ to denote contexts when discussing formalized proofs.

The context lemmas below are proven as part of this work and are used in later proofs in this thesis. See the accompanying source code for the proofs. Note that all variables are externally quantified and each occurrence of β and Γ, possibly with subscripts, has type `oo` and `context`, respectively.

Lemma 5.1. ***elem_inv***

$$\frac{\beta_1 \in (\Gamma, \beta_2)}{(\beta_1 \in \Gamma) \vee (\beta_1 = \beta_2)}$$

Lemma 5.2. ***elem_sub***

$$\frac{\beta_1 \in \Gamma}{\beta_1 \in (\Gamma, \beta_2)}$$

Lemma 5.3. ***elem_self***

$$\frac{}{\beta \in (\Gamma, \beta)}$$

Lemma 5.4. ***elem_rep***

$$\frac{\beta_1 \in (\Gamma, \beta_2, \beta_2)}{\beta_1 \in (\Gamma, \beta_2)}$$

Lemma 5.5. ***context_swap***

$$\frac{}{(\Gamma, \beta_1, \beta_2) = (\Gamma, \beta_2, \beta_1)}$$

Lemma 5.6. ***context_sub_sup***

$$\frac{\Gamma_1 \subseteq \Gamma_2}{(\Gamma_1, \beta) \subseteq (\Gamma_2, \beta)}$$

5.2 Hereditary Harrop Specification Logic in Coq

The inference rules of the SL are implemented using two sequent judgments that distinguish between *goal-reduction rules* and *backchaining rules* which correspond to the right rules and left focused rules, respectively, of Figure 4.2 in Section 4.2. Figures 5.1 and 5.2 implement the inference rules of the SL (except for the disjunction rules, as mentioned in the introduction to this chapter). They are encoded in Coq as two mutually inductive types, one each for goal-reduction and backchaining sequents. The syntax used in the figures is a pretty-printed version of the Coq inductive types `grseq` and `bcseq`. Goal-reduction sequents have type $\texttt{grseq} : \texttt{context} \to \texttt{oo} \to \texttt{Prop}$, and we write $\Gamma \rhd \beta$ as notation for $\texttt{grseq}\ \Gamma\ \beta$. Backchaining sequents have type $\texttt{bcseq} : \texttt{context} \to \texttt{oo} \to \texttt{atm} \to \texttt{Prop}$ and we write $\Gamma, [\beta] \rhd \alpha$ as notation for $\texttt{bcseq}\ \Gamma\ \beta\ \alpha$, understanding β to be the focused formula from Γ. The symbol $\forall$ is used for universal quantification in Coq, rather than universal quantification in SL formulas. When we see $\forall$ in the premises of rules, this is to make it clear that it is only over the premise of the rule.

The rule names in the figures are the constructor names in the inductive definitions in the Coq files. The premises and conclusion of a rule are the argument types and the target type, respectively, of one clause in the definition. Quantification at the outer level is implicit and, as noted, inner quantification is written explicitly in the premises. For example, the linear format of the *g_dyn* rule from Figure 5.1 with all quantifiers explicit is

$$\forall(\Gamma : \texttt{context})(D : \texttt{oo})(A : \texttt{atm}), D \in \Gamma \to \Gamma, [D] \rhd A \to \Gamma \rhd \langle\ A\ \rangle$$

This is the type of the *g_dyn* constructor in the inductive definition of `grseq` (see the definition of `grseq` in the Coq files).

The notation $\langle\ A\ \rangle$ is to say that $A : \texttt{atm}$ is coerced to have type `oo`, where `oo` is the implemented type of formulas (see Figure 3.4), referred to as o in Chapter 4. The constants `Some` and `All` are used for existential and universal quantification in SL formulas, respectively, over the type `expr con` which is the type for OL expressions. `Allx` is a constant for universal quantification over a type `X` of type `Set`. We juxtapose appropriate terms to denote application since Coq will reduce the expression, rather than explicitly writing the substitution (for example, compare rule $\forall_R$ in Figure 4.2 to rule *g_allx* in Figure 5.1). A final implementation byproduct is the predicate `proper` appearing in the premise of some rules. This is used in the Hybrid library to rule out terms of type `expr` that have dangling indices (see [8]).

In the sequents for this SL there is also an implicit fixed context Δ, called the *static program clauses*, containing closed clauses (D-formulas) of the form

$$\forall_{\tau_1} \ldots \forall_{\tau_n}.G \longrightarrow A$$

$$\frac{A :- G \quad \Gamma \rhd G}{\Gamma \rhd \langle\, A \,\rangle}\ g_prog \qquad \frac{D \in \Gamma \quad \Gamma, [D] \rhd A}{\Gamma \rhd \langle\, A \,\rangle}\ g_dyn \qquad \frac{}{\Gamma \rhd \mathtt{T}}\ g_tt$$

$$\frac{\Gamma \rhd G_1 \quad \Gamma \rhd G_2}{\Gamma \rhd G_1 \,\&\, G_2}\ g_and \qquad \frac{\Gamma\,,\, D \rhd G}{\Gamma \rhd D \longrightarrow G}\ g_imp \qquad \frac{\mathtt{proper}\ E \quad \Gamma \rhd G\,E}{\Gamma \rhd \mathtt{Some}\ G}\ g_some$$

$$\frac{\forall (E : \mathtt{expr\ \ con}), (\mathtt{proper}\ E \to \Gamma \rhd G\,E)}{\Gamma \rhd \mathtt{All}\ G}\ g_all \qquad \frac{\forall (E : \mathtt{X}), (\Gamma \rhd G\,E)}{\Gamma \rhd \mathtt{Allx}\ G}\ g_allx$$

Figure 5.1: Goal-Reduction Rules, $\mathtt{grseq} : \mathtt{context} \to \mathtt{oo} \to \mathtt{Prop}$

with $n \geq 0$. Any set of D-formulas can be transformed to an equivalent one that all have this form. These clauses represent the inference rules of an OL. We do not explicitly mention Δ in the rules for this SL because no rules modify it.

The goal-reduction rules of Figure 5.1 are implemented version of the right introduction rules of this sequent calculus as seen in Figure 4.2. The rules *g_prog* and *g_dyn* are the only goal-reduction rules with an atomic principal formula.

The rule *g_prog* is used to backchain over the static program clauses Δ, which are defined for each new OL as an inductive type called `prog` of type $\mathtt{atm} \to \mathtt{oo} \to \mathtt{Prop}$, and represent the inference rules of the OL (this is discussed further in Section 3.4). The rule *g_prog* is the interface between the SL and OL layers and we say that the SL is parametric in OL provability. We write $A :- G$ for ($\mathtt{prog}\ A\ G$) to suggest backward implication. Recall that clauses in Δ may have outermost universal quantification. The premise $A :- G$ actually represents an instance of a clause in Δ.

The rule *g_dyn* allows backchaining over dynamic assumptions (i.e. a formula from Γ) and is the implemented version of the *focus* rule of Figure 4.2. To use this rule to prove $\Gamma \rhd \langle\, A \,\rangle$, we need to show $D \in \Gamma$ and $\Gamma, [D] \rhd A$. Formula D is chosen from, or shown to be in, the dynamic context Γ and we use the backchaining rules of Figure 5.2 to show $\Gamma, [D] \rhd A$ (where D is the focused formula).

The backchaining rules of Figure 5.2 are the implemented version of the standard focused left rules for conjunction, implication, and universal and existential quantification seen in Figure 4.2. Considered bottom up, they provide backchaining over the focused formula. In using the backchaining rules, each branch is either completed by *b_match* where the focused formula is an atomic formula identical to the goal of the sequent, or *b_imp* is used resulting in one branch switching back to using goal-reduction rules.

$$\frac{}{\Gamma, [\langle\, A\,\rangle] \rhd A}\ b_match \qquad \frac{\Gamma, [D_1] \rhd A}{\Gamma, [D_1 \,\&\, D_2] \rhd A}\ b_and_1 \qquad \frac{\Gamma, [D_2] \rhd A}{\Gamma, [D_1 \,\&\, D_2] \rhd A}\ b_and_2$$

$$\frac{\Gamma \rhd G \quad \Gamma, [D] \rhd A}{\Gamma, [G \longrightarrow D] \rhd A}\ b_imp \qquad \frac{\texttt{proper}\ E \quad \Gamma, [D\,E] \rhd A}{\Gamma, [\texttt{All}\ D] \rhd A}\ b_all \qquad \frac{\Gamma, [D\,E] \rhd A}{\Gamma, [\texttt{Allx}\ D] \rhd A}\ b_allx$$

$$\frac{\forall (E : \texttt{expr con}), (\texttt{proper}\ E \to \Gamma, [D\,E] \rhd A)}{\Gamma, [\texttt{Some}\ D] \rhd A}\ b_some$$

Figure 5.2: Backchaining Rules, `bcseq` : `context` → `oo` → `atm` → `Prop`

5.3 Mutual Structural Induction

The theorem statements in this thesis all have the form

$$\begin{aligned} &(\forall\ (c : \texttt{context})\ (o : \texttt{oo}), \\ &\qquad\qquad (c \rhd o) \to (P_1\ c\ o))\ \wedge \\ &(\forall\ (c : \texttt{context})\ (o : \texttt{oo})\ (a : \texttt{atm}), \\ &\qquad\qquad (c, [o] \rhd a) \to (P_2\ c\ o\ a)) \end{aligned}$$

where P_1 : `context` → `oo` → `Prop` and P_2 : `context` → `oo` → `atm` → `Prop` are predicates extracted from the statement to be proven. The `Scheme` command provides an induction principle to allow us to prove statements of the above form by mutual structural induction.

To prove such a statement by mutual structural induction over $c \rhd o$ and $c, [o] \rhd a$, 15 subcases must be proven, one corresponding to each inference rule of the SL. The proof state of each subcase of this induction is constructed from an inference rule of the system. We can see the sequent mutual induction principle in Figure 5.3, where each antecedent (clause of the induction principle defining the cases) corresponds to a rule of the SL and a subcase for an induction using this technique. After backchaining over the induction principle, the 15 subcases are generated. As an aside, externally quantified variables in each antecedent can be introduced to the context of assumptions of the proof state and are then considered *signature variables*. For example, the subcase for *g_prog* will have signature variables c : `context`, o : `oo`, and a : `atm`.

This induction principle is automatically generated following the description shown below, with examples from the figure given in each point.

$$
\begin{array}{rl}
\multicolumn{2}{l}{\texttt{seq_mutind} : \forall (P_1 : \texttt{context} \to \texttt{oo} \to \texttt{Prop})(P_2 : \texttt{context} \to \texttt{oo} \to \texttt{atm} \to \texttt{Prop}),} \\
(*g_prog*) & (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), \\
& \quad a :- o \to c \rhd o \to P_1\ c\ o \to P_1\ c\ \langle\ a\ \rangle) \to \\
(*g_dyn*) & (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), \\
& \quad o \in c \to c, [o] \rhd a \to P_2\ c\ o\ a \to P_1\ c\ \langle\ a\ \rangle) \to \\
(*g_tt*) & (\forall (c : \texttt{context}), P_1\ c\ \texttt{T}) \to \\
(*g_and*) & (\forall (c : \texttt{context})(o_1\ o_2 : \texttt{oo}), \\
& \quad c \rhd o_1 \to P_1\ c\ o_1 \to c \rhd o_2 \to P_1\ c\ o_2 \to P_1\ c\ (o_1 \& o_2)) \to \\
(*g_imp*) & (\forall (c : \texttt{context})(o_1\ o_2 : \texttt{oo}), \\
& \quad c, o_1 \rhd o_2 \to P_1\ (c, o_1)\ o_2 \to P_1\ c\ (o_1 \longrightarrow o_2)) \to \\
(*g_all*) & (\forall (c : \texttt{context})(o : \texttt{expr con} \to \texttt{oo}), \\
& \quad (\forall (e : \texttt{expr con}), \texttt{proper}\ e \to c \rhd o\ e) \to \\
& \quad (\forall (e : \texttt{expr con}), \texttt{proper}\ e \to P_1\ c\ (o\ e)) \to P_1\ c\ (\texttt{All}\ o)) \to \\
(*g_allx*) & (\forall (c : \texttt{context})(o : \texttt{X} \to \texttt{oo}), \\
& \quad (\forall (e : \texttt{X}), c \rhd o\ e) \to (\forall (e : \texttt{X}), P_1\ c\ (o\ e)) \to P_1\ c\ (\texttt{Allx}\ o)) \to \\
(*g_some*) & (\forall (c : \texttt{context})(o : \texttt{expr con} \to \texttt{oo})(e : \texttt{expr con}), \\
& \quad \texttt{proper}\ e \to c \rhd o\ e \to P_1\ c\ (o\ e) \to P_1\ c\ (\texttt{Some}\ o)) \to \\
(*b_match*) & (\forall (c : \texttt{context})(a : \texttt{atm})), c, [\langle\ a\ \rangle] \rhd a) \\
(*b_and_1*) & (\forall (c : \texttt{context})(o_1\ o_2 : \texttt{oo})(a : \texttt{atm}), \\
& \quad c, [o_1] \rhd a \to P_2\ c\ o_1\ a \to P_2\ c\ (o_1 \& o_2)\ a) \to \\
(*b_and_2*) & (\forall (c : \texttt{context})(o_1\ o_2 : \texttt{oo})(a : \texttt{atm}), \\
& \quad c, [o_2] \rhd a \to P_2\ c\ o_2\ a \to P_2\ c\ (o_1 \& o_2)\ a) \to \\
(*b_imp*) & (\forall (c : \texttt{context})(o_1\ o_2 : \texttt{oo})(a : \texttt{atm}), \\
& \quad c \rhd o_1 \to P_1\ c\ o_1 \to c, [o_2] \rhd a \to P_2\ c\ o_2\ a \to \\
& \quad P_2\ c\ (o_1 \longrightarrow o_2)\ a) \to \\
(*b_all*) & (\forall (c : \texttt{context})(o : \texttt{expr con} \to \texttt{oo})(a : \texttt{atm})(e : \texttt{expr con}), \\
& \quad \texttt{proper}\ e \to c, [o\ e] \rhd a \to P_2\ c\ (o\ e)\ a \to P_2\ c\ (\texttt{All}\ o)\ a) \to \\
(*b_allx*) & (\forall (c : \texttt{context})(o : \texttt{X} \to \texttt{oo})(a : \texttt{atm})(e : \texttt{X}), \\
& \quad c, [o\ e] \rhd a) \to P_2\ c\ (o\ e)\ a \to P_2\ c\ (\texttt{Allx}\ o)\ a \to \\
(*b_some*) & (\forall (c : \texttt{context})(o : \texttt{expr con} \to \texttt{oo})(a : \texttt{atm}), \\
& \quad (\forall (e : \texttt{expr con}), \texttt{proper}\ e \to c, [o\ e] \rhd a) \to \\
& \quad (\forall (e : \texttt{expr con}), \texttt{proper}\ e \to P_2\ c\ (o\ e)\ a) \to P_2\ c\ (\texttt{Some}\ o)\ a) \to \\
& (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \to P_1\ c\ o) \wedge \\
& (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \to P_2\ c\ o\ a)
\end{array}
$$

Figure 5.3: SL Sequent Mutual Induction Principle

- Non-sequent premises are assumptions of the induction subcase. For example, $o \in c$ from the *g_dyn* rule.

- For every rule premise that is a goal-reduction sequent (with possible local quantifiers) of the form $\forall (x_1 : T_1) \cdots (x_n : T_n), \Gamma \rhd \beta$ where $n \geq 0$, the induction subcase has assumptions $(\forall (x_1 : T_1) \cdots (x_n : T_n), \Gamma \rhd \beta)$ and $(\forall (x_1 : T_1) \cdots (x_n : T_n), P_1\ \Gamma\ \beta)$. For example, $\forall (e : \texttt{expr con}), \texttt{proper}\ e \to c \rhd o\ e$ and $\forall (e : \texttt{expr con}), \texttt{proper}\ e \to P_1\ c\ (o\ e)$ from the *g_all* rule with $n = 2$ and unabbreviated prefix $\forall (e : \texttt{expr con})(H : \texttt{proper}\ e)$.

- For every rule premise that is a backchaining sequent (with possible local quantifiers) of the form $\forall (x_1 : T_1) \cdots (x_n : T_n), \Gamma, [\beta] \rhd \alpha$ where $n \geq 0$, the induction subcase has assumptions $(\forall (x_1 : T_1) \cdots (x_n : T_n), \Gamma, [\beta] \rhd \alpha)$ and $(\forall (x_1 : T_1) \cdots (x_n : T_n), P_2\ \Gamma\ \beta\ \alpha)$. For example, $c, [o_2] \rhd a$ and $(P_2\ c\ o_2\ a)$ from the *b_imp* rule.

- If the rule conclusion is a goal-reduction sequent of the form $\Gamma \rhd \beta$, then the subcase goal is $P_1\ \Gamma\ \beta$. For example, $(P_1\ c\ \langle\ a\ \rangle)$ from the *g_dyn* rule.

- If the rule conclusion is a backchaining sequent of the form $\Gamma, [\beta] \rhd \alpha$, then the subcase goal is $P_2\ \Gamma\ \beta\ \alpha$. For example, $(P_2\ c\ (o_1 \longrightarrow o_2)\ a)$ from the *b_imp* rule.

Assumptions in the second and third bullets above that contain P_1 or P_2 are induction hypotheses. Implicit in the last two points is the possible introduction of more assumptions, in the case when P_1 and P_2 are dependent products themselves (i.e. contain quantification and/or implication). As a trivial example, if $P_1\ \Gamma\ \beta$ is $\forall (\delta : \texttt{oo}), \beta \in \Gamma \to \beta \in (\Gamma, \delta)$, then we can introduce δ into the context as a new signature variable and $\beta \in \Gamma$ as a new assumption. We will refer to assumptions introduced this way as *induction assumptions* in future proofs, since they are from a predicate that is used to construct induction hypotheses.

In describing proofs, we will follow the Coq style as introduced in Section 2.6.1 and write the proof state in a vertical format with the assumptions above a horizontal line and the goal below it. For example, the *g_dyn* subcase requires a proof of

$$\begin{aligned} &\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), \\ &\qquad o \in c \to c, [o] \rhd a \to P_2\ c\ o\ a \to P_1\ c\ \langle\ a\ \rangle \end{aligned}$$

This can be seen in lines 4 and 5 in the induction principle in Figure 5.3. After introductions, the proof state will have the following form:

$$
\frac{\begin{array}{r@{\,:\,}l} c & \texttt{context} \\ o & \texttt{oo} \\ a & \texttt{atm} \\ H_1 & o \in c \\ H_2 & c, [o] \rhd a \\ IH & P_2\ c\ o\ a \end{array}}{P_1\ c\ \langle\ a\ \rangle}
$$

As in Coq, we provide hypothesis names so that we can refer to them as needed. Also, we often omit the type declarations of signature variables. In this case we could have omitted $c : \texttt{context}$, $o : \texttt{oo}$, and $a : \texttt{atm}$ because they can be easily inferred from context (they must have these types for $c, [o] \rhd a$ to be well-typed).

Chapter 6

Specification Logic Metatheory

Proving admissibility of structural rules of a specification logic (SL) frees us from defining them as axiomatic and having to make external justifications for such axioms. We prove admissibility of the structural rules of contraction, weakening, exchange, and cut for both goal-reduction and backchaining sequents. Once proven at the specification level, they can be reused for any OL using this SL. Cut admissibility is particularly useful and considerably more challenging to prove than the other structural rules. It establishes consistency and also provides justification for substituting a formula for an assumption in a context of assumptions. It can greatly simplify reasoning about OLs in systems that provide HOAS.

We can prove properties of this logic using the mutual structural induction principle over the rules of the SL from Figure 5.3 when the theorem (or goal statement) is the same form as the conclusion of the induction principle. Backchaining over the induction principle, we will have fifteen subcases; one subcase corresponding to each rule of the SL. Many of these cases have similar proofs. We will look at a few cases that are interesting for the following reasons:

g_dyn
: This rule has a goal-reduction sequent conclusion, a non-sequent premise depending on the context of the conclusion and a backchaining sequent premise.

g_imp
: This rule has a goal-reduction sequent conclusion and a sequent premise with a context different from that of the conclusion.

b_imp
: This rule has a backchaining sequent conclusion and both a goal-reduction and backchaining sequent premise.

6.1 Structural Rules

For our SL we prove the standard structural rules of weakening, contraction, and exchange for both goal-reduction and backchaining sequents:

Theorem 6.1. ***gr_weakening***

$$\frac{\Gamma \rhd \beta_2}{\Gamma\,,\beta_1 \rhd \beta_2}$$

Theorem 6.2. ***bc_weakening***

$$\frac{\Gamma, [\beta_2] \rhd \alpha}{\Gamma\,,\beta_1, [\beta_2] \rhd \alpha}$$

Theorem 6.3. ***gr_contraction***

$$\frac{\Gamma\,,\ \beta_1\,,\beta_1 \rhd \beta_2}{\Gamma\,,\beta_1 \rhd \beta_2}$$

Theorem 6.4. ***bc_contraction***

$$\frac{\Gamma\,,\ \beta_1\,,\ \beta_1, [\beta_2] \rhd \alpha}{\Gamma\,,\ \beta_1, [\beta_2] \rhd \alpha}$$

Theorem 6.5. ***gr_exchange***

$$\frac{\Gamma\,,\ \beta_2\,,\ \beta_1 \rhd \beta_3}{\Gamma\,,\beta_1\,,\ \beta_2 \rhd \beta_3}$$

Theorem 6.6. ***bc_exchange***

$$\frac{\Gamma\,,\ \beta_2\,,\ \beta_1, [\beta_3] \rhd \alpha}{\Gamma\,,\ \beta_1\,,\ \beta_2, [\beta_3] \rhd \alpha}$$

These are all corollaries of a general theorem:

Theorem 6.7. ***monotone***

$$\frac{\Gamma \subseteq c' \quad \Gamma \rhd \beta}{c' \rhd \beta} \text{ and } \frac{\Gamma \subseteq c' \quad \Gamma, [\beta] \rhd \alpha}{c', [\beta] \rhd \alpha}$$

Proof:

Theorem 6.7 is proven by mutual structural induction over the premises $\Gamma \rhd \beta$ and $\Gamma, [\beta] \rhd \alpha$. Defining P_1 and P_2 as

$$\begin{aligned} P_1 :=& \lambda\ (c : \texttt{context})(o : \texttt{oo})\ . \\ & \quad \forall\ (c' : \texttt{context}), c \subseteq c' \to c' \rhd o \\ P_2 :=& \lambda\ (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm})\ . \\ & \quad \forall\ (c' : \texttt{context}), c \subseteq c' \to c', [o] \rhd a \end{aligned}$$

we are proving

$$\begin{aligned} &(\forall\ (c : \texttt{context})\ (o : \texttt{oo}), \\ &\qquad (c \rhd o) \to (P_1\ c\ o)) \ \wedge \\ &(\forall\ (c : \texttt{context})\ (o : \texttt{oo})\ (a : \texttt{atm}), \\ &\qquad (c, [o] \rhd a) \to (P_2\ c\ o\ a)) \end{aligned}$$

which has the form discussed in Section 5.3, so the mutual structural induction principle may be used. Here we will show the cases for the rules *g_dyn*, *g_imp*, and *b_imp*. The antecedent of the induction principle for each subcase gives the initial subgoals.

Case $\dfrac{D \in \Gamma \quad \Gamma, [D] \rhd A}{\Gamma \rhd \langle\ A\ \rangle}$ ***g_dyn*:**

This rule has one non-sequent premise and one backchaining sequent premise. So there will be one induction hypothesis from the backchaining sequent premise. From the induction principle in Figure 5.3 we need to prove

$$\begin{aligned} &\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), \\ &\qquad o \in c \to c, [o] \rhd a \to P_2\ c\ o\ a \to P_1\ c\ \langle\ a\ \rangle \end{aligned}$$

After introductions the proof state is

$$\frac{\begin{array}{r} H_1 : o \in c \\ Hb_1 : c, [o] \rhd a \\ IHb_1 : P_2\ c\ o\ a \end{array}}{P_1\ c\ \langle\ a\ \rangle}$$

Unfolding P_1 and P_2 as defined for this theorem, we have

$$\frac{\begin{array}{rl} H_1 : & o \in c \\ Hb_1 : & c, [o] \rhd a \\ IHb_1 : & \forall\ (c' : \texttt{context}), c \subseteq c' \to c', [o] \rhd a \end{array}}{\forall\ (c' : \texttt{context}), c \subseteq c' \to c' \rhd \langle\ a\ \rangle}$$

Next we make introductions from the goal.

$$\frac{\begin{array}{rl} H_1 : & o \in c \\ Hb_1 : & c, [o] \rhd a \\ IHb_1 : & \forall\ (c' : \texttt{context}), c \subseteq c' \to c', [o] \rhd a \\ c' : & \texttt{context} \\ P_1 : & c \subseteq c' \end{array}}{c' \rhd \langle\ a\ \rangle}$$

Now the goal is a goal-reduction sequent with an atomic formula. We can backchain with the rule *g_dyn* and will get two new subgoals from the premises of this rule.

$$\frac{\begin{array}{rl} H_1 : & o \in c \\ Hb_1 : & c, [o] \rhd a \\ IHb_1 : & \forall\ (c' : \texttt{context}), c \subseteq c' \to c', [o] \rhd a \\ c' : & \texttt{context} \\ P_1 : & c \subseteq c' \end{array}}{(o \in c'), (c', [o] \rhd a)}$$

To prove the second subgoal we use induction hypothesis IHb_1 to get the new subgoal $c \subseteq c'$ which is provable by induction assumption P_1. To prove the first, we need to

unfold the definition of subset in P_1.

$$\begin{array}{rl} H_1 : & o \in c \\ Hb_1 : & c, [o] \rhd a \\ IHb_1 : & \forall\ (c' : \texttt{context}), c \subseteq c' \to c', [o] \rhd a \\ c' : & \texttt{context} \\ P_1 : & \forall (o : \texttt{oo}), o \in c \to o \in c' \\ \hline & o \in c' \end{array}$$

Backchaining over P_1 we get the new subgoal $o \in c$ which is provable by assumption H_1. The proof for this case is complete.

Case $\dfrac{\Gamma\,,\, D \rhd G}{\Gamma \rhd D \longrightarrow G}$ ***g_imp*:**

This rule has one goal-reduction sequent premise which gives one induction hypothesis. From the induction principle the goal is

$$\forall (c : \texttt{context})(o_1\ o_2 : \texttt{oo}), c, o_1 \rhd o_2 \to P_1\ (c, o_1)\ o_2 \to P_1\ c\ (o_1 \longrightarrow o_2)$$

After introductions we are proving

$$\begin{array}{rl} Hg_1 : & c, o_1 \rhd o_2 \\ IHg_1 : & P_1\ (c, o_1)\ o_2 \\ \hline & P_1\ c\ (o_1 \longrightarrow o_2) \end{array}$$

Unfolding P_1 as defined for this theorem, we have

$$\begin{array}{rl} Hg_1 : & c, o_1 \rhd o_2 \\ IHg_1 : & \forall (c' : \texttt{context}), (c, o_1) \subseteq c' \to c' \rhd o_2 \\ \hline & \forall (c' : \texttt{context}), c \subseteq c' \to c' \rhd o_1 \longrightarrow o_2 \end{array}$$

Next we make introductions from the goal.

$$\begin{array}{rl} Hg_1 : & c, o_1 \rhd o_2 \\ IHg_1 : & \forall (c' : \texttt{context}), (c, o_1) \subseteq c' \to c' \rhd o_2 \\ c' : & \texttt{context} \\ P_1 : & c \subseteq c' \\ \hline & c' \rhd o_1 \longrightarrow o_2 \end{array}$$

The rule *g_imp* is the only rule of the SL that we can backchain over with the current goal.

$$\begin{array}{r l} Hg_1 & : c, o_1 \rhd o_2 \\ IHg_1 & : \forall(c' : \texttt{context}), (c, o_1) \subseteq c' \to c' \rhd o_2 \\ c' & : \texttt{context} \\ P_1 & : c \subseteq c' \\ \hline & c', o_1 \rhd o_2 \end{array}$$

Now we use the induction hypothesis IHg_1. This step of backward reasoning gives the new subgoal $c, o_1 \subseteq c', o_1$. Next backchain with the context lemma `context_sub_sup` (Lemma 5.6) and we have to prove $c \subseteq c'$ which is provable by the induction assumption P_1. The proof for this case is complete.

Case $\dfrac{\Gamma \rhd G \quad \Gamma, [D] \rhd A}{\Gamma, [G \longrightarrow D] \rhd A}$ ***b_imp*:**

This rule has one goal-reduction sequent premise and one backchaining sequent premise. So there will be one induction hypothesis from each sequent premise. From the induction principle we need to prove

$$\begin{array}{l} \forall(c : \texttt{context})(o_1\ o_2 : \texttt{oo})(a : \texttt{atm}), \\ \quad c \rhd o_1 \to P_1\ c\ o_1 \to c, [o_2] \rhd a \to P_2\ c\ o_2\ a \to P_2\ c\ (o_1 \longrightarrow o_2)\ a \end{array}$$

After introductions the proof state is

$$\begin{array}{r l} Hg_1 & : c \rhd o_1 \\ IHg_1 & : P_1\ c\ o_1 \\ Hb_1 & : c, [o_2] \rhd a \\ IHb_1 & : P_2\ c\ o_2\ a \\ \hline & P_2\ c\ (o_1 \longrightarrow o_2)\ a \end{array}$$

We unfold uses of P_1 and P_2.

$$\begin{array}{r l} Hg_1 & : c \rhd o_1 \\ IHg_1 & : \forall(c' : \texttt{context}), c \subseteq c' \to c' \rhd o_1 \\ Hb_1 & : c, [o_2] \rhd a \\ IHb_1 & : \forall(c' : \texttt{context}), c \subseteq c' \to c', [o_2] \rhd a \\ \hline & \forall(c' : \texttt{context}), c \subseteq c' \to c, [o_1 \longrightarrow o_2] \rhd a \end{array}$$

Next we can make introductions from the goal.

$$\begin{array}{l} Hg_1 : c \rhd o_1 \\ IHg_1 : \forall (c' : \texttt{context}), c \subseteq c' \to c' \rhd o_1 \\ Hb_1 : c, [o_2] \rhd a \\ IHb_1 : \forall (c' : \texttt{context}), c \subseteq c' \to c', [o_2] \rhd a \\ IP_1 : c \subseteq c' \\ \hline c', [o_1 \longrightarrow o_2] \rhd a \end{array}$$

The only SL rule whose conclusion matches the goal is *b_imp* so we backchain with this rule to get two new subgoals.

$$\begin{array}{l} Hg_1 : c \rhd o_1 \\ IHg_1 : \forall (c' : \texttt{context}), c \subseteq c' \to c' \rhd o_1 \\ Hb_1 : c, [o_2] \rhd a \\ IHb_1 : \forall (c' : \texttt{context}), c \subseteq c' \to c', [o_2] \rhd a \\ IP_1 : c \subseteq c' \\ \hline (c' \rhd o_1), (c', [o_2] \rhd a) \end{array}$$

We backchain over the appropriate induction hypothesis for each of these subgoals, and in both cases get the subgoal $c \subseteq c'$, provable by induction assumption IP_1. The proof of this subcase is complete.

6.2 Cut Admissibility

The cut rule is shown to be admissible in this specification logic by proving the following:

Theorem 6.8. ***cut_admissible***

$$\frac{\Gamma, \delta \rhd \beta \quad \Gamma \rhd \delta}{\Gamma \rhd \beta} \text{ and } \frac{\Gamma, \delta, [\beta] \rhd \alpha \quad \Gamma \rhd \delta}{\Gamma, [\beta] \rhd \alpha}$$

Since our specification logic makes use of two kinds of sequents, we prove two cut rules. These correspond to the two conjuncts above, where the first is for goal-reduction sequents and the second is for backchaining sequents.

Proof: (**Outline**)

This proof will be a nested induction, first over the cut formula δ, then over the sequent premises with δ in their contexts. Since there are seven rules for constructing formulas and 15 SL rules, this will result in 105 subcases. These can be partitioned into five classes with the same proof structure, four of which we briefly illustrate presently.

The cases for the axioms *g_tt* and *b_match* are proven by one use of `constructor` (7 formulas * 2 rules = 14 subcases).

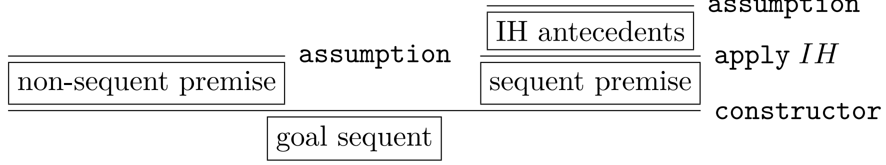
$$\dfrac{}{\boxed{\text{goal sequent}}}\ \texttt{constructor}$$

Cases for rules with only sequent premises, including those with inner quantification, with the same context as the conclusion have the same proof structure. Note that by *same context*, we include rules modifying the focused formula. The rules in this class are *g_and*, *g_all*, *g_allx*, b_and_1, b_and_2, *b_imp*, *b_allx*, and *b_some* (7 formulas * 8 rules = 56 subcases). We apply `constructor` to the goal sequent which, after any introductions, will give a sequent subgoal for each sequent premise of the rule. To each of the new subgoals we apply the appropriate induction hypothesis, giving new subgoals for each antecedent of each induction hypothesis used. Now all goals can be proven by assumption (hypotheses from the induction principle and induction assumptions).

$$\dfrac{\dfrac{\dfrac{}{\left\{\boxed{\text{IH antecedents}}\right\}}\ \texttt{assumption}}{\left\{\boxed{\text{rule sequent premise(s)}}\right\}}\ \texttt{apply}\ IH}{\boxed{\text{goal sequent}}}\ \texttt{constructor}$$

Only one rule modifies the context of the sequent, *g_imp* (7 formulas * 1 rule = 7 subcases). The proof of the subcase for this rule is similar to above, but requires the use of another structural rule, `gr_weakening` (Theorem 6.1), before the sequent subgoal will match the sequent assumption introduced from the goal.

The remaining four rules have both a non-sequent premise and a sequent premise. Of these, the subcases for *g_prog*, *g_some*, and *b_all* have a similar proof structure; apply `constructor` to the goal so that the non-sequent premise is provable by assumption, then prove the branch for the sequent premise as above (7 formulas * 3 rules = 21 subcases).



The proof of the subcase for g_dyn is more complicated due to the form of the non-sequent premise, $D \in \Gamma$, which depends on the context in the goal sequent, $\Gamma \rhd \langle\, A \,\rangle$. We need more details to analyse the subcases for this rule further.

So 98 of 105 subcases are proven following this outline.

(end outline)

∎

The cut admissibility theorem stated above is a simple corollary of the following theorem (with explicit quantification):

$$\begin{aligned} &\forall(\delta : \texttt{oo}),\ (\forall(c : \texttt{context})(o : \texttt{oo}),\ c \rhd o \to \\ &\qquad \forall(c' : \texttt{context}), c = c', \delta \to c' \rhd \delta \to c' \rhd o) \wedge \\ &\quad (\forall(c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}),\ c, [o] \rhd a \to \\ &\qquad \forall(c' : \texttt{context}), c = c', \delta \to c' \rhd \delta \to c', [o] \rhd a) \end{aligned}$$

Proof:

We begin with an induction over δ, so we are proving $\forall(\delta : \texttt{oo}), P\ \delta$ with P defined as

$$\begin{aligned} P : \texttt{oo} \to \texttt{Prop} := \lambda(\delta : \texttt{oo})\ .\ &(\forall(c : \texttt{context})(o : \texttt{oo}), \\ &\qquad c \rhd o \to P_1\ c\ o) \wedge \\ &(\forall(c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), \\ &\qquad c, [o] \rhd a \to P_2\ c\ o\ a) \end{aligned}$$

where

$$\begin{aligned} &P_1 : \texttt{context} \to \texttt{oo} \to \texttt{Prop} := \lambda(c : \texttt{context})(o : \texttt{oo})\ . \\ &\qquad \forall(c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to c' \rhd o \\ &P_2 : \texttt{context} \to \texttt{oo} \to \texttt{atm} \to \texttt{Prop} := \lambda(c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm})\ . \\ &\qquad \forall(c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to c', [o] \rhd a \end{aligned}$$

P, P_1, and P_2 will provide the induction hypotheses used in this proof. Next is a nested induction, which is a mutual structural induction over $c \rhd o$ and $c, [o] \rhd a$ using P_1 and P_2 as above.

In the proof presentation here we will only look at cases for the rule *g_dyn*. Later we will see a generalization of the SL and a proof that captures the remaining 98 cases, as well as the proof of `monotone` (Theorem 6.7) seen above. Since in the proof of `monotone` we have already seen how to prove a few concrete cases in detail using the mutual structural induction principle, it would be tedious to continue to work through more subcases in the same way.

6.2.1 Subcase for *g_dyn*: Alternate Proof Attempt

Before proving this subcase for the nested induction, suppose that rather than an outer induction over the cut formula δ we had simply introduced this variable into the context of the proof state and begun the proof as a mutual structural induction over the sequent premises with δ in their context. Then we can wait until it is necessary to have an induction over the cut formula.

The subcase of the induction principle for *g_dyn* from Figure 5.3 requires a proof of

$$\begin{aligned}&\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}),\\ &\qquad o \in c \to c, [o] \rhd a \to P_2\ c\ o\ a \to P_1\ c\ \langle\ a\ \rangle\end{aligned}$$

After introductions and unfolding P_1 and P_2 as defined for this theorem, the proof state is

$$\begin{array}{rl} H_1 : & o \in c \\ Hb_1 : & c, [o] \rhd a \\ IHb_1 : & \forall (c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to c', [o] \rhd a \\ \hline & \forall (c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to c' \rhd \langle\ a\ \rangle \end{array}$$

Next we make introductions from the goal.

$$\begin{array}{rl} H_1 : & o \in c \\ Hb_1 : & c, [o] \rhd a \\ IHb_1 : & \forall (c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to c', [o] \rhd a \\ c' : & \texttt{context} \\ IP_1 : & c = (c', \delta) \\ IP_2 : & c' \rhd \delta \\ \hline & c' \rhd \langle\ a\ \rangle \end{array}$$

Next we substitute (c', δ) for c using IP_1 and rename c' to Γ_0 in IHb_1 to distinguish the bound variable from the free variable c'. Now ignore IP_1.

$$\begin{array}{r@{\ }l} H_1 : & o \in c', \delta \\ Hb_1 : & c', \delta, [o] \rhd a \\ IHb_1 : & \forall(\Gamma_0 : \texttt{context}), (c', \delta) = (\Gamma_0, \delta) \to \Gamma_0 \rhd \delta \to \Gamma_0, [o] \rhd a \\ c' : & \texttt{context} \\ IP_2 : & c' \rhd \delta \\ \hline & c' \rhd \langle\ a\ \rangle \end{array}$$

We can get a new premise $P_3 : c', [o] \rhd a$ by specializing IHb_1 with c', a reflexivity lemma and IP_2. Now ignore IHb_1 which is no longer needed and Hb_1 which we can get from `bc_weakening` (Theorem 6.2) and P_3.

$$\begin{array}{r@{\ }l} H_1 : & o \in c', \delta \\ c' : & \texttt{context} \\ IP_2 : & c' \rhd \delta \\ P_3 : & c', [o] \rhd a \\ \hline & c' \rhd \langle\ a\ \rangle \end{array}$$

We can apply the context lemma `elem_inv` to H_1 to get the premise $(o \in c') \vee (o = \delta)$. Applying `inversion` to this, we have two new subgoals with diverging sets of assumptions. In the second we substitute δ for o by H_1 in that proof state.

$$\begin{array}{r@{\ }l@{\qquad\qquad}r@{\ }l} H_1 : & o \in c' & H_1 : & o = \delta \\ c' : & \texttt{context} & c' : & \texttt{context} \\ IP_2 : & c' \rhd \delta & IP_2 : & c' \rhd \delta \\ P_3 : & c', [o] \rhd a & P_3 : & c', [\delta] \rhd a \\ \hline & c' \rhd \langle\ a\ \rangle & & c' \rhd \langle\ a\ \rangle \end{array}$$

The left subgoal is provable by first applying *g_dyn* to get subgoals $o \in c'$ and $c', [o] \rhd a$, both proven by assumption.

The proof on the right will be continued with an induction over δ. The property to prove is

$$\begin{aligned} P_0 : \texttt{oo} \to \texttt{Prop} := \lambda(\delta : \texttt{oo})\ . \\ \forall(c' : \texttt{context})(a : \texttt{atm}), \\ c' \rhd \delta \to c', [\delta] \rhd a \to c' \rhd \langle\ a\ \rangle \end{aligned}$$

We will now look at a specific subcase of this induction.

Subcase $\delta = o_1 \longrightarrow o_2$:

In this case we prove the appropriate antecedent of the induction principle for induction over δ (see Figure 3.5), shown below.

$$\forall (o_1\ o_2 : \texttt{oo}), P_0\ o_1 \to P_0\ o_2 \to P_0\ (o_1 \longrightarrow o_2)$$

The expanded proof state after premise introductions is:

$$\begin{array}{l}
IH_1 : \forall (c' : \texttt{context})(a : \texttt{atm}), c' \rhd o_1 \to c', [o_1] \rhd a \to c' \rhd \langle\ a\ \rangle \\
IH_2 : \forall (c' : \texttt{context})(a : \texttt{atm}), c' \rhd o_2 \to c', [o_2] \rhd a \to c' \rhd \langle\ a\ \rangle \\
c' : \texttt{context} \\
IP_2 : c' \rhd (o_1 \longrightarrow o_2) \\
P_3 : c', [o_1 \longrightarrow o_2] \rhd a \\
\hline
c' \rhd \langle\ a\ \rangle
\end{array}$$

We can apply `inversion` to the premises IP_2 and P_3 to get new assumptions in the context.

$$\begin{array}{l}
IH_1 : \forall (c' : \texttt{context})(a : \texttt{atm}), c' \rhd o_1 \to c', [o_1] \rhd a \to c' \rhd \langle\ a\ \rangle \\
IH_2 : \forall (c' : \texttt{context})(a : \texttt{atm}), c' \rhd o_2 \to c', [o_2] \rhd a \to c' \rhd \langle\ a\ \rangle \\
c' : \texttt{context} \\
IP_2 : c', o_1 \rhd o_2 \\
P_{3_1} : c', [o_2] \rhd a \\
P_{3_2} : c' \rhd o_1 \\
\hline
c' \rhd \langle\ a\ \rangle
\end{array}$$

IH_1 is not useful here, since we have no way to prove sequents with o_1 focused. Applying IH_2 and ignoring induction hypotheses, we have:

$$\begin{array}{l}
c' : \texttt{context} \\
IP_2 : c', o_1 \rhd o_2 \\
P_{3_1} : c', [o_2] \rhd a \\
P_{3_2} : c' \rhd o_1 \\
\hline
(c', o_2, [o_2] \rhd a), (c' \rhd o_2), (c', [o_2] \rhd a)
\end{array}$$

The first subgoal is proven using `bc_weakening` (Theorem 6.2) and assumption P_{3_1}, and the third subgoal by P_{3_1}.

On trying to prove the second subgoal, we should reflect on two things. First, proving $c' \rhd o_2$ from the assumptions IP_2 and P_{3_2} would be a use of the goal-reduction cut rule. Second, we are proving the subcase corresponding to the *g_dyn* rule and the only sequent premise of this rule is a backchaining sequent; we only get the backchaining part of the cut rule in the induction hypothesis. To illustrate this, recall that for this subcase we have $c, [o] \rhd a$ and the induction hypothesis $P_2\ c\ o\ a$ in the context of assumptions. The induction hypothesis expands to

$$\forall (c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to c', [o] \rhd a$$

Combining these assumptions we have

$$c', \delta, [o] \rhd a \to c' \rhd \delta \to c', [o] \rhd a$$

which is the conjunct of the cut rule for backchaining sequents. Combining the above observations, we see that this branch cannot be continued any further.

6.2.2 Subcase for *g_dyn*: Original Proof Structure

Convinced of the necessity of our original proof structure, now we will move on with our proof of the cut rule by nested inductions, first on the cut formula δ then over the sequent premises with δ in the context. Below is a proof of the *g_dyn* subcase where $\delta = o_1 \longrightarrow o_2$. The *g_dyn* subcases for other formula constructions follow similarly.

Case $\delta = o_1 \ \longrightarrow \ o_2$ **:**

From Figure 3.5, the antecedent of the `oo` induction principle for this case is

$$\forall (o_1\ o_2 : \texttt{oo}), P\ o_1 \to P\ o_2 \to P\ (o_1 \longrightarrow o_2)$$

where $P\ o_1$ and $P\ o_2$ are induction hypotheses and P is as defined at the start of this proof. Expanding the goal (we will wait to expand the premises), the proof state is

$$\begin{array}{l} IH_1 : P\ o_1 \\ IH_2 : P\ o_2 \\ \hline (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \to \forall (c' : \texttt{context}), \\ \qquad c = (c', (o_1 \longrightarrow o_2)) \to c' \rhd (o_1 \longrightarrow o_2) \to c' \rhd o) \wedge \\ (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \to \forall (c' : \texttt{context}), \\ \qquad c = (c', (o_1 \longrightarrow o_2)) \to c' \rhd (o_1 \longrightarrow o_2) \to c', [o] \rhd a) \end{array}$$

Next we have the mutual induction over sequents. As stated above, we will only show the subcase for the *g_dyn* rule.

Subcase $\dfrac{D \in \Gamma \quad \Gamma, [D] \rhd A}{\Gamma \rhd \langle\, A \,\rangle}$ *g_dyn* **:**

The goal for this subcase is

$$\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), o \in c \to c, [o] \rhd a \to P_2\ c\ o\ a \to P_1\ c\ \langle\, a \,\rangle$$

After introductions, the proof state is

$$\begin{aligned}
IH_1 &: P\ o_1 \\
IH_2 &: P\ o_2 \\
H_1 &: o \in c \\
Hb_1 &: c, [o] \rhd a \\
IHb_1 &: \forall (c' : \texttt{context}), c = (c', o_1 \longrightarrow o_2) \to c' \rhd (o_1 \longrightarrow o_2) \to c', [o] \rhd a \\
c' &: \texttt{context} \\
IP_1 &: c = c', o_1 \longrightarrow o_2 \\
IP_2 &: c' \rhd o_1 \longrightarrow o_2
\end{aligned}$$

$$c' \rhd \langle\, a \,\rangle$$

Next substitute $(c', o_1 \longrightarrow o_2)$ for c using IP_1 and rename c' to Γ_0 in IHb_1 to distinguish the bound variable from the free variable c'. Now ignore IP_1.

$$\begin{aligned}
IH_1 &: P\ o_1 \\
IH_2 &: P\ o_2 \\
H_1 &: o \in (c', o_1 \longrightarrow o_2) \\
Hb_1 &: c', o_1 \longrightarrow o_2, [o] \rhd a \\
IHb_1 &: \forall (\Gamma_0 : \texttt{context}), \\
&\qquad (c', o_1 \longrightarrow o_2) = (\Gamma_0, o_1 \longrightarrow o_2) \to \Gamma_0 \rhd (o_1 \longrightarrow o_2) \to \Gamma_0, [o] \rhd a \\
c' &: \texttt{context} \\
IP_2 &: c' \rhd o_1 \longrightarrow o_2
\end{aligned}$$

$$c' \rhd \langle\, a \,\rangle$$

We can specialize IHb_1 with c', a reflexivity lemma and IP_2 to get the new premise P_3 : $c', [o] \rhd a$ and apply `elem_inv` (Lemma 5.1) to H_1 to get $(o \in c') \vee (o = o_1 \longrightarrow o_2)$. Now ignore IHb_1 and Hb_1 (we can get the latter from assumption P_3 and `bc_weakening`, Theorem 6.2).

$$\dfrac{\begin{array}{r l} IH_1 : & P\ o_1 \\ IH_2 : & P\ o_2 \\ H_1 : & (o \in c') \vee (o = o_1 \longrightarrow o_2) \\ IP_2 : & c' \rhd o_1 \longrightarrow o_2 \\ P_3 : & c', [o] \rhd a \end{array}}{c' \rhd \langle\ a\ \rangle}$$

Inverting H_1, we get two new subgoals with different sets of assumptions. In the second we substitute $o_1 \longrightarrow o_2$ for o using H_1 in that proof state.

$$\dfrac{\begin{array}{r l} IH_1 : & P\ o_1 \\ IH_2 : & P\ o_2 \\ H_1 : & o \in c' \\ IP_2 : & c' \rhd o_1 \longrightarrow o_2 \\ P_3 : & c', [o] \rhd a \end{array}}{c' \rhd \langle\ a\ \rangle} \qquad \dfrac{\begin{array}{r l} IH_1 : & P\ o_1 \\ IH_2 : & P\ o_2 \\ H_1 : & o = o_1 \longrightarrow o_2 \\ IP_2 : & c' \rhd o_1 \longrightarrow o_2 \\ P_3 : & c', [o_1 \longrightarrow o_2] \rhd a \end{array}}{c' \rhd \langle\ a\ \rangle}$$

To prove the first, we apply *g_dyn* to the goal, then need to prove $o \in c'$ and $c', [o] \rhd a$ which are both provable by assumption.

For the second (right) subgoal, it will be necessary to apply `inversion` to some assumptions to get structurally simpler assumptions, before being able to apply the induction hypotheses IH_1 and IH_2. Inverting IP_2 and P_3, and unfolding P, we have:

$$\dfrac{\begin{array}{l l} IH_1 : & (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \rightarrow \\ & \qquad \forall (c' : \texttt{context}), c = (c', o_1) \rightarrow c' \rhd o_1 \rightarrow c' \rhd o) \wedge \\ & (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \rightarrow \\ & \qquad \forall (c' : \texttt{context}), c = (c', o_1) \rightarrow c' \rhd o_1 \rightarrow c', [o] \rhd a) \\ IH_2 : & (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \rightarrow \\ & \qquad \forall (c' : \texttt{context}), c = (c', o_2) \rightarrow c' \rhd o_2 \rightarrow c' \rhd o) \wedge \\ & (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \rightarrow \\ & \qquad \forall (c' : \texttt{context}), c = (c', o_2) \rightarrow c' \rhd o_2 \rightarrow c', [o] \rhd a) \\ IP_2 : & c', o_1 \rhd o_2 \\ P_{3_1} : & c' \rhd o_1 \\ P_{3_2} : & c', [o_2] \rhd a \end{array}}{c' \rhd \langle\ a\ \rangle}$$

Backchaining on the first conjunct of IH_2, instantiating c with (c', o_2), gives three new subgoals.

$$\begin{array}{l}
IH_1 : (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_1) \rightarrow c' \rhd o_1 \rightarrow c' \rhd o) \wedge \\
\qquad (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_1) \rightarrow c' \rhd o_1 \rightarrow c', [o] \rhd a) \\
IH_2 : (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_2) \rightarrow c' \rhd o_2 \rightarrow c' \rhd o) \wedge \\
\qquad (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_2) \rightarrow c' \rhd o_2 \rightarrow c', [o] \rhd a) \\
P_{3_1} : c' \rhd o_1 \\
P_{3_2} : c', [o_2] \rhd a \\
IP_2 : c', o_1 \rhd o_2 \\
\hline
\quad (c', o_2 \rhd \langle\ a\ \rangle), (c', o_2 = c', o_2), (c' \rhd o_2)
\end{array}$$

For the first, apply *g_dyn*, then we need to prove $o_2 \in (c', o_2)$ (proven by `elem_self`, Lemma 5.3) and $c', o_2, [o_2] \rhd a$ (proven by `bc_weakening`, Theorem 6.2, and assumption P_{3_2}). The second is proven by `reflexivity`. For the third, we backchain on the first conjunct of IH_1, instantiating c with (c', o_1), and get three new subgoals.

$$\begin{array}{l}
IH_1 : (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_1) \rightarrow c' \rhd o_1 \rightarrow c' \rhd o) \wedge \\
\qquad (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_1) \rightarrow c' \rhd o_1 \rightarrow c', [o] \rhd a) \\
IH_2 : (\forall (c : \texttt{context})(o : \texttt{oo}), c \rhd o \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_2) \rightarrow c' \rhd o_2 \rightarrow c' \rhd o) \wedge \\
\qquad (\forall (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c, [o] \rhd a \rightarrow \\
\qquad\qquad \forall (c' : \texttt{context}), c = (c', o_2) \rightarrow c' \rhd o_2 \rightarrow c', [o] \rhd a) \\
P_{3_1} : c' \rhd o_1 \\
P_{3_2} : c', [o_2] \rhd a \\
IP_2 : c', o_1 \rhd o_2 \\
\hline
\quad (c', o_1 \rhd o_2), (c', o_1 = c', o_1), (c' \rhd o_1)
\end{array}$$

The sequent subgoals are proven by `assumption` and the other by `reflexivity`.

```
Proof.
Hint Resolve context_sub_sup.
eapply seq_mutind; intros;
try (econstructor; eauto; eassumption).
Qed.
```

Figure 6.1: Coq proof of `monotone` (Theorem 6.7)

```
Proof.
Hint Resolve gr_weakening context_swap.
induction delta; eapply seq_mutind; intros;
subst; try (econstructor; eauto; eassumption).
...
```

Figure 6.2: Coq proof of 98/105 cases of `cut_admissible` (Theorem 6.8)

The *g_dyn* subcases for the remaining six constructors of `oo` follow a similar argument requiring `inversion` on hypotheses and induction hypothesis specialization.

From this presentation we can see that working through the details for every case can be a tedious and repetitive task. We later see a generalization that helps us to understand what subcases have the same structure and separate out the challenging cases. This understanding leads us to a condensed automated Coq proof for `monotone` (Theorem 6.7, see Figure 6.1) and proofs of 98 of 105 subcases in the proof of `cut_admissible` (Theorem 6.8, see Figure 6.2 where `delta` is the cut formula in the implementation, in place of δ).

Chapter 7

Generalized Specification Logic

All non-axiomatic rules of the SL have some number of premises that are either non-sequent predicates, goal-reduction sequents, or backchaining sequents. Also, all rule conclusions are sequents; this is necessary to encode these rules in inductive types `grseq` and `bcseq`. With this observation, we can generalize the rules of the SL inference system to aid our understanding and presentation of proofs presented in Chapter 6

Here we present generalized specification logic rules to reduce the number of induction cases and allow us to partition cases of proofs about the original SL based on rule structure. Our goal is to gain insight into the high-level structure of such inductive proofs, providing the proof writer and reader with the ability to understand where the difficult cases are and how similar cases can be handled in a general way.

All rules have one of the following forms:

$$\frac{\begin{array}{c}\overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\ \forall\overline{(x_{n,s_n} : R_{n,s_n})},(c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\ \forall\overline{(y_{p,t_p} : S_{p,t_p})},(c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p})\end{array}}{c \rhd o}\ gr_rule$$

$$\frac{\begin{array}{c}\overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\ \forall\overline{(x_{n,s_n} : R_{n,s_n})},(c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\ \forall\overline{(y_{p,t_p} : S_{p,t_p})},(c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p})\end{array}}{c, [o] \rhd a}\ bc_rule$$

where m, n, p represent the (possibly zero) number of non-sequent premises, goal-reduction sequent premises, and backchaining sequent premises, respectively. Note that for all rules in our implemented SL, $0 \leq m \leq 1$, $0 \leq n \leq 2$, and $0 \leq p \leq 1$.

We call this collection of inference rules consisting of *gr_rule* and *bc_rule* the generalized specification logic (GSL). This is *not* implemented in Coq as the previously described SL is; but rather all rules of the SL can be instantiated from the two rules of the GSL (as will be seen in Section 7.1). The GSL allows us to investigate the SL without needing to consider each of the 15 rules of the SL separately. This makes it possible to more efficiently study and explain the metatheory of the SL.

Much of the notation used in these rules requires further explanation. A horizontal bar above an element with some subscript index, say z, means we have a collection of such items indexed from 1 to z. For example, the "premise" $\overline{Q_m}\langle\!\langle c, o\rangle\!\rangle$ represents the m premises $Q_1\langle\!\langle c, o\rangle\!\rangle, \ldots, Q_m\langle\!\langle c, o\rangle\!\rangle$. The premises with sequents can possibly have local quantification. For $i = 1, \ldots, n$, $\overline{(x_{i,s_i} : R_{i,s_i})}$ represents the prefix $(x_{i,1} : R_{i,1}) \cdots (x_{i,s_i} : R_{i,s_i})$.

The notation $\langle\!\langle\cdot\rangle\!\rangle$ is used to list arguments from the conclusion that may be used in instances of terms where it occurs. We wish to show how elements of the rule conclusion propagate through a proof.

Given types $T_0, T_1, \ldots, T_z$, when we write $F\langle\!\langle a_1 : T_1, \ldots, a_z : T_z\rangle\!\rangle : T_0$, we mean a term of type T_0 that may contain any (sub)terms appearing in conclusion terms $a_1, \ldots, a_z$. For example, given $\gamma_1\langle\!\langle D \longrightarrow G : \texttt{oo}\rangle\!\rangle : \texttt{context}$, we may "instantiate" this expression to $\{D\}$. We often omit types and use definitional notation, e.g., in this case we may write $\gamma_1\langle\!\langle D \longrightarrow G\rangle\!\rangle \coloneqq \{D\}$.

We infer the following typing judgments from the GSL rules:

- For $i = 1, \ldots, m$, the definition of Q_i may use the context and formula of the conclusion, so with full typing information, $Q_i\langle\!\langle c : \texttt{context}, o : \texttt{oo}\rangle\!\rangle : \texttt{Prop}$
- For $j = 1, \ldots, n$, SL context γ_j may use the formula of the conclusion and SL formula F_j may use the formula of the conclusion and locally quantified variables. So with full typing information, $\gamma_j\langle\!\langle o : \texttt{oo}\rangle\!\rangle : \texttt{context}$ and $F_j\langle\!\langle o : \texttt{oo}, x_{j,1} : R_{j,1}, \ldots, x_{j,s_j} : R_{j,s_j}\rangle\!\rangle : \texttt{oo}$
- For $k = 1, \ldots, p$, SL context γ'_k may use the formula of the conclusion and SL formula F'_k may use the formula of the conclusion and locally quantified variables. So with full typing information $\gamma'_k\langle\!\langle o : \texttt{oo}\rangle\!\rangle : \texttt{context}$ and $F'_k\langle\!\langle o : \texttt{oo}, y_{k,1} : S_{k,1}, \ldots, y_{k,t_k} : S_{k,t_k}\rangle\!\rangle : \texttt{oo}$

7.1 SL Rules from GSL Rules

The rules of the GSL can be instantiated to obtain the SL by specifying the values of the variables in the GSL rules. We first fill in m, n, and p. Then for $i = 1, \ldots, m$, we specify Q_i. For $j = 1, \ldots, n$, we specify s_j, γ_j, F_j, $x_{j,1}$, ..., x_{j,s_j}, and $R_{j,1}$, ..., R_{j,s_j}. For $k = 1, \ldots, p$, we specify γ'_k, F'_k, $y_{k,1}$, ..., y_{k,t_k}, and $S_{k,1}$, ..., S_{k,t_k}. Below are instantiations for all rules of the SL.

Rule	m	n	p	c	o
$\dfrac{A :- G \quad \Gamma \rhd G}{\Gamma \rhd \langle\, A \,\rangle}$ *g_prog*	1	1	0	Γ	$\langle\, A \,\rangle$
$s_1 := 0$ $Q_1⦅\Gamma, \langle\, A \,\rangle⦆ := A :- G \quad \gamma_1⦅\langle\, A \,\rangle⦆ := \emptyset \quad F_1⦅\langle\, A \,\rangle⦆ := G$					
$\dfrac{D \in \Gamma \quad \Gamma, [D] \rhd A}{\Gamma \rhd \langle\, A \,\rangle}$ *g_dyn*	1	0	1	Γ	$\langle\, A \,\rangle$
$t_1 := 0 \quad a_1 := A$ $Q_1⦅\Gamma, \langle\, A \,\rangle⦆ := D \in \Gamma \quad \gamma'_1⦅\langle\, A \,\rangle⦆ := \emptyset \quad F'_1⦅\langle\, A \,\rangle⦆ := D$					
$\dfrac{}{\Gamma \rhd \mathtt{T}}$ *g_tt*	0	0	0	Γ	$\mathtt{T}$
$\dfrac{\Gamma \rhd G_1 \quad \Gamma \rhd G_2}{\Gamma \rhd G_1 \,\&\, G_2}$ *g_and*	0	2	0	Γ	$G_1\&G_2$
$s_1 := 0 \quad s_2 := 0$ $\gamma_1⦅G_1\&G_2⦆ := \emptyset \quad F_1⦅G_1\&G_2⦆ := G_1$ $\gamma_2⦅G_1\&G_2⦆ := \emptyset \quad F_2⦅G_1\&G_2⦆ := G_2$					
$\dfrac{\Gamma\,,\, D \rhd G}{\Gamma \rhd D \longrightarrow G}$ *g_imp*	0	1	0	Γ	$D \longrightarrow G$
$s_1 := 0 \quad \gamma_1⦅D \longrightarrow G⦆ := \{D\} \quad F_1⦅D \longrightarrow G⦆ := G$					
$\dfrac{\forall (E : \mathtt{expr\ \ con}), (\mathtt{proper}\ E \rightarrow \Gamma \rhd G\, E)}{\Gamma \rhd \mathtt{All}\ G}$ *g_all*	0	1	0	Γ	$\mathtt{All}\ G$
$s_1 := 2 \quad x_{1,1} := E \quad R_{1,1} := \mathtt{expr\ \ con} \quad x_{1,2} := H \quad R_{1,2} := \mathtt{proper}\ E$ $\gamma_1⦅\mathtt{All}\ G⦆ := \emptyset \quad F_1⦅\mathtt{All}\ G, E, H⦆ := G\ E$					
$\dfrac{\forall (E : \mathtt{X}), (\Gamma \rhd G\, E)}{\Gamma \rhd \mathtt{Allx}\ G}$ *g_allx*	0	1	0	Γ	$\mathtt{Allx}\ G$
$s_1 := 1 \quad x_{1,1} := E \quad R_{1,1} := \mathtt{X}$ $\gamma_1⦅\mathtt{Allx}\ G⦆ := \emptyset \quad F_1⦅\mathtt{Allx}\ G, E, H⦆ := G\ E$					
$\dfrac{\mathtt{proper}\ E \quad \Gamma \rhd G\, E}{\Gamma \rhd \mathtt{Some}\ G}$ *g_some*	1	1	0	Γ	$\mathtt{Some}\ G$
$s_1 := 0$ $Q_1⦅\Gamma, \mathtt{Some}\ G⦆ := \mathtt{proper}\ E \quad \gamma_1⦅\mathtt{Some}\ G⦆ := \emptyset \quad F_1⦅\mathtt{Some}\ G⦆ := G\ E$					

Rule	m	n	p	c	o
$\dfrac{}{\Gamma, [\langle\ A\ \rangle] \rhd A}\ b_match$	0	0	0	Γ	$\langle\ A\ \rangle$
$\dfrac{\Gamma, [D_1] \rhd A}{\Gamma, [D_1 \,\&\, D_2] \rhd A}\ b_and_1$	0	0	1	Γ	$D_1 \& D_2$
$t_1 := 0 \quad a_1 := A \quad \gamma'_1\langle\!\langle D_1 \& D_2 \rangle\!\rangle := \emptyset \quad F'_1\langle\!\langle D_1 \& D_2 \rangle\!\rangle := D_1$					
$\dfrac{\Gamma, [D_2] \rhd A}{\Gamma, [D_1 \,\&\, D_2] \rhd A}\ b_and_2$	0	0	1	Γ	$D_1 \& D_2$
$t_1 := 0 \quad a_1 := A \quad \gamma'_1\langle\!\langle D_1 \& D_2 \rangle\!\rangle := \emptyset \quad F'_1\langle\!\langle D_1 \& D_2 \rangle\!\rangle := D_2$					
$\dfrac{\Gamma \rhd G \quad \Gamma, [D] \rhd A}{\Gamma, [G \longrightarrow D] \rhd A}\ b_imp$	0	1	1	Γ	$G \longrightarrow D$
$s_1 := 0 \quad t_1 := 0 \quad a_1 := A$ $\gamma_1\langle\!\langle G \longrightarrow D \rangle\!\rangle := \emptyset \quad F_1\langle\!\langle G \longrightarrow D \rangle\!\rangle := G$ $\gamma'_1\langle\!\langle G \longrightarrow D \rangle\!\rangle := \emptyset \quad F'_1\langle\!\langle G \longrightarrow D \rangle\!\rangle := D$					
$\dfrac{\texttt{proper}\ E \quad \Gamma, [D\, E] \rhd A}{\Gamma, [\texttt{All}\ D] \rhd A}\ b_all$	1	0	1	Γ	$\texttt{All}\ D$
$t_1 := 0 \quad a_1 := A$ $Q_1\langle\!\langle \Gamma, \texttt{All}\ D \rangle\!\rangle := \texttt{proper}\ E \quad \gamma'_1\langle\!\langle \texttt{All}\ D \rangle\!\rangle := \emptyset \quad F'_1\langle\!\langle \texttt{All}\ D \rangle\!\rangle := D\ E$					
$\dfrac{\Gamma, [D\, E] \rhd A}{\Gamma, [\texttt{Allx}\ D] \rhd A}\ b_allx$	0	0	1	Γ	$\texttt{All}\ D$
$t_1 := 0 \quad a_1 := A$ $\gamma'_1\langle\!\langle \texttt{Allx}\ D \rangle\!\rangle := \emptyset \quad F'_1\langle\!\langle \texttt{Allx}\ D \rangle\!\rangle := D\ E$					
$\dfrac{\forall (E : \texttt{expr con}), (\texttt{proper}\ E \to \Gamma, [D\, x] \rhd A)}{\Gamma, [\texttt{Some}\ D] \rhd A}\ b_some$	0	0	1	Γ	$\texttt{Some}\ D$
$t_1 := 2 \quad y_{1,1} := E \quad S_{1,1} := \texttt{expr con} \quad y_{1,2} := H \quad S_{1,2} := \texttt{proper}\ E \quad a_1 := A$ $\gamma'_1\langle\!\langle \texttt{Some}\ D, E, H \rangle\!\rangle := \emptyset \quad F'_1\langle\!\langle \texttt{Some}\ D, E, H \rangle\!\rangle := D\ E$					

Notice that for the g_dyn rule, D appears in Q_1, even though it is not in the argument list of Q_1. The notation $\langle\!\langle \cdot \rangle\!\rangle$ only specifies arguments from the rule conclusion. Any variables that only appear in the premises of a rule of the SL are also permitted to appear in the propositions, formulas, and contexts when specializing the premises of a GSL rule to obtain the premises of a specific SL rule (these are the signature variables for induction subcases corresponding to these rules).

Chapter 8

Generalized Specification Logic Metatheory

8.1 GSL Induction Part I: A Restricted Theorem

Recall from Chapter 6 that when proving SL metatheory, we were concerned with proving statements of the form

$$\begin{aligned}
&(\forall\ (c : \texttt{context})\ (o : \texttt{oo}),\\
&\qquad (c \rhd o) \to (P_1\ c\ o))\ \ \wedge\\
&(\forall\ (c : \texttt{context})\ (o : \texttt{oo})\ (a : \texttt{atm}),\\
&\qquad (c, [o] \rhd a) \to (P_2\ c\ o\ a))
\end{aligned}$$

In the GSL we now have two rules instead of the 15 of the SL. To prove this statement by mutual structural induction over the GSL we will have two subcases; one for each of the rules *gr_rule* and *bc_rule*. From the rule *gr_rule* (resp. *bc_rule*) the induction subcase has n induction hypotheses for the n goal-reduction sequent premises and p induction hypotheses for the p backchaining sequent premises of the rule. We also assume the m non-sequent premises. After introductions, the proof state is:

$$\begin{array}{r@{\,}l}
\overline{H_m} & : \overline{Q_m}\langle\!\langle c, o\rangle\!\rangle\\
\overline{Hg_n} & : \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle)\\
\overline{IHg_n} & : \forall\overline{(x_{n,s_n} : R_{n,s_n})}, P_1\ (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle)\ (\overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle)\\
\overline{Hb_p} & : \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p})\\
\overline{IHb_p} & : \forall\overline{(y_{p,t_p} : S_{p,t_p})}, P_2\ (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle)\ (\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle)\ \overline{a_p}\\
\hline
& P_1\ c\ o\ (resp.\ P_2\ c\ o\ a)
\end{array}$$

Given specific P_1 and P_2, we could unfold uses of these predicates and continue the proof. Suppose

$$P_1 := \lambda(c : \texttt{context})\ (o : \texttt{oo}).$$
$$\forall(c' : \texttt{context}), IA_1\langle\!\langle c, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c, c'\rangle\!\rangle \to \underline{c' \rhd o} \qquad \text{and}$$
$$P_2 := \lambda(c : \texttt{context})\ (o : \texttt{oo})\ (a : \texttt{atm}).$$
$$\forall(c' : \texttt{context}), IA_1\langle\!\langle c, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c, c'\rangle\!\rangle \to \underline{c', [o] \rhd a}$$

Each IA_i is a predicate that we call an *induction assumption*. P_1 and P_2 can be instantiated to specific statements about the GSL by defining these IA_i. Notice that this is a generalization of all induction predicates we have seen so far. The underlining of sequents in the definitions of P_1 and P_2 is to highlight that these are the sequents we apply the generalized rules to (following introductions).

First we unfold uses of P_1 and P_2 in the proof state.

$$\frac{\begin{array}{r l} \overline{H_m} : & \overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\ \overline{Hg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\ \overline{IHg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}(c' : \texttt{context}), \\ & \quad IA_1\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c' \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle \\ \overline{Hb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p}) \\ \overline{IHb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}(c' : \texttt{context}), \\ & \quad IA_1\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c', [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p} \end{array}}{\begin{array}{c} \forall(c' : \texttt{context}), IA_1\langle\!\langle c, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c, c'\rangle\!\rangle \to \underline{c' \rhd o} \\ (\textit{resp.}\ \forall(c' : \texttt{context}), IA_1\langle\!\langle c, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c, c'\rangle\!\rangle \to \underline{c', [o] \rhd a}) \end{array}}$$

Next we introduce the variables and induction assumptions. Then the goal is either $c' \rhd o$ or $c', [o] \rhd a$. Apply *gr_rule* or *bc_rule* as appropriate, and either will give $(m + n + p)$ new subgoals which come from the three premise forms in these rules, with appropriate instantiations for the externally quantified variables. c' is a new signature variable. See Figure 8.1 for this proof state.

8.1.1 Sequent Subgoals

To prove the last $(n + p)$ subgoals (the "second" and "third" subgoals in Figure 8.1) we first introduce any locally quantified variables as signature variables.

$$
\begin{array}{rl}
\overline{H_m} : & \overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\
\overline{Hg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\
\overline{IHg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}(c' : \texttt{context}), \\
& \quad IA_1\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c' \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle \\
\overline{Hb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p}) \\
\overline{IHb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}(c' : \texttt{context}), \\
& \quad IA_1\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c', [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p} \\
c' : & \texttt{context} \\
\overline{IP_w} : & \overline{IA_w}\langle\!\langle c, c'\rangle\!\rangle \\
\hline
& \overline{Q_m}\langle\!\langle c', o\rangle\!\rangle, \\
& \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle), \\
& \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p})
\end{array}
$$

Figure 8.1: Proof state of GSL induction after rule application

$$
\begin{array}{rl}
\overline{H_m} : & \overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\
\overline{Hg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\
\overline{IHg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}(c' : \texttt{context}), \\
& \quad IA_1\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c' \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle \\
\overline{Hb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p}) \\
\overline{IHb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}(c' : \texttt{context}), \\
& \quad IA_1\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c', [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p} \\
c' : & \texttt{context} \\
\overline{IP_w} : & \overline{IA_w}\langle\!\langle c, c'\rangle\!\rangle \\
\overline{x_{n,s_n}} : & \overline{R_{n,s_n}}\ (resp.\ \overline{y_{p,t_p}} : \overline{S_{p,t_p}}) \\
\hline
& c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle\ \ (resp.\ c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p})
\end{array}
$$

For the goal-reduction (resp. backchaining) subgoals, for $j = 1, \ldots, n$ (resp. $k = 1, \ldots, p$), we apply induction hypothesis IHg_j (resp. IHb_k), instantiating c' in

$$\begin{array}{rl}
\overline{H_m} : & \overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\
\overline{Hg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\
\overline{IHg_n} : & \forall\overline{(x_{n,s_n} : R_{n,s_n})}(c' : \texttt{context}), \\
& \quad IA_1\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c' \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle \\
\overline{Hb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p}) \\
\overline{IHb_p} : & \forall\overline{(y_{p,t_p} : S_{p,t_p})}(c' : \texttt{context}), \\
& \quad IA_1\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to \cdots \to IA_w\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c'\rangle\!\rangle \to c', [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p} \\
c' : & \texttt{context} \\
\overline{IP_w} : & \overline{IA_w}\langle\!\langle c, c'\rangle\!\rangle \\
\overline{x_{n,s_n}} : & \overline{R_{n,s_n}} \ (resp.\ \overline{y_{p,t_p}} : \overline{S_{p,t_p}}) \\
\hline
\multicolumn{2}{c}{\overline{IA_w}\langle\!\langle c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle, c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle\rangle\!\rangle \ \ (resp.\ \overline{IA_w}\langle\!\langle c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle\rangle\!\rangle)}
\end{array}$$

Figure 8.2: Incomplete proof branches for sequent premises

the induction hypothesis with $c' \cup \gamma_j\langle\!\langle o\rangle\!\rangle$ (resp. $c' \cup \gamma'_k\langle\!\langle o\rangle\!\rangle$). This yields the proof state in Figure 8.2 for goal-reduction premises (resp. backchaining premises).

The proof state in Figure 8.2 will be continued for specific theorem statements which will have the induction assumptions defined.

8.1.2 Non-Sequent Subgoals

The proof of the first m subgoals in Figure 8.1 depends on the definition of Q_i for $i = 1 \ldots m$. If the first argument (a `context`) is not used in its definition, then $Q_i\langle\!\langle c', o\rangle\!\rangle$ is provable by assumption H_i since we will have $Q_i\langle\!\langle c', o\rangle\!\rangle = Q_i\langle\!\langle c, o\rangle\!\rangle$. Any other dependencies on signature variables can be ignored since we can instantiate the variables as we choose when backchaining over the generalized rule. We will illustrate this by considering each rule with non-sequent premises, starting from the proof state in Figure 8.1 and, for $(i = 1, \ldots, m)$, $(j = 1, \ldots, n)$, $(k = 1, \ldots, p)$, show how to define Q_i, γ_j, F_j, γ'_k, and F'_k and finish the subproofs where possible.

There are four rules of the SL with non-sequent premises: *g_prog*, *g_dyn*, *g_some*, and *b_all*.

Case $\dfrac{A :- G \quad \Gamma \rhd G}{\Gamma \rhd \langle\, A \,\rangle}\ \textit{g_prog}$ **:**

This rule has one non-sequent premise and one goal-reduction sequent premise with no local quantification, so $m = n = 1$, $p = 0$, $s_1 = 0$, $o = \langle\ A\ \rangle$, and $c = \Gamma$. Then we are proving the following:

$$\begin{array}{rl} H_1 : & Q_1(\!(\Gamma, \langle\ A\ \rangle)\!) \\ Hg_1 : & \Gamma \cup \gamma_1(\!(\langle\ A\ \rangle)\!) \rhd F_1(\!(\langle\ A\ \rangle)\!) \\ IHg_1 : & \forall (c' : \texttt{context}), IA_1(\!(\Gamma, c')\!) \to \cdots \to IA_w(\!(\Gamma, c')\!) \to c' \rhd F_1(\!(\langle\ A\ \rangle)\!) \\ c' : & \texttt{context} \\ \overline{IP_w} : & \overline{IA_w}(\!(\Gamma, c')\!) \\ \hline & Q_1(\!(c', \langle\ A\ \rangle)\!) \end{array}$$

Define $Q_1(\!(_, \langle\ A\ \rangle)\!) \coloneqq A :- G$, $\gamma_1(\!(\langle\ A\ \rangle)\!) \coloneqq \emptyset$, and $F_1(\!(\langle\ A\ \rangle)\!) \coloneqq G$, where $G : \texttt{oo}$ is a signature variable. Now the proof state is

$$\begin{array}{rl} H_1 : & A :- G \\ Hg_1 : & \Gamma \rhd G \\ IHg_1 : & \forall (c' : \texttt{context}), IA_1(\!(\Gamma, c')\!) \to \cdots \to IA_w(\!(\Gamma, c')\!) \to c' \rhd G \\ c' : & \texttt{context} \\ \overline{IP_w} : & \overline{IA_w}(\!(\Gamma, c')\!) \\ \hline & A :- G \end{array}$$

which is completed by assumption H_1.

Case $\dfrac{D \in \Gamma \quad \Gamma, [D] \rhd A}{\Gamma \rhd \langle\ A\ \rangle}$ *g_dyn* **:**

This rule has one non-sequent premise and one backchaining sequent premise with no local quantification, so $m = p = 1$, $n = 0$, $c = \Gamma$, and $o = \langle\ A\ \rangle$. Define $Q_1(\!(\Gamma, \langle\ A\ \rangle)\!) \coloneqq D \in \Gamma$, $\gamma_1'(\!(\langle\ A\ \rangle)\!) \coloneqq \emptyset$, and $F_1'(\!(\langle\ A\ \rangle)\!) \coloneqq D$, where $D : \texttt{oo}$ is a signature variable. Then we need to prove what is displayed in Figure 8.3. Here we do not have enough information to finish this branch of the proof. An induction assumption may be of use, but we will need specific P_1 and P_2. We will refer to Figure 8.3 later when proving specific theorem statements (i.e. each IA_i defined).

Case $\dfrac{\texttt{proper}\ E \quad \Gamma \rhd G\, E}{\Gamma \rhd \texttt{Some}\ G}$ *g_some* **:**

This rule has one non-sequent premise and one goal-reduction sequent premise with no local quantification, so $m = n = 1$, $p = 0$, $c = \Gamma$, and $o = \texttt{Some}\ G$. Define

$$\begin{array}{rl} H_1 & : D \in \Gamma \\ Hb_1 & : \Gamma, [D] \rhd a_1 \\ IHb_1 & : \forall (c' : \texttt{context}), IA_1 \langle\!\langle \Gamma, c' \rangle\!\rangle \to \cdots \to IA_w \langle\!\langle \Gamma, c' \rangle\!\rangle \to c', [D] \rhd a_1 \\ c' & : \texttt{context} \\ \overline{IP_w} & : \overline{IA_w} \langle\!\langle \Gamma, c' \rangle\!\rangle \\ \hline & D \in c' \end{array}$$

Figure 8.3: Incomplete proof branch (*g_dyn* case)

$Q_1\langle\!\langle \Gamma, \texttt{Some}\ G \rangle\!\rangle := \texttt{proper}\ E$, $\gamma_1\langle\!\langle \texttt{Some}\ G \rangle\!\rangle := \emptyset$, and $F_1\langle\!\langle \texttt{Some}\ G \rangle\!\rangle := G\ E$ where $E : \texttt{expr con}$ is a signature variable. Then we are proving the following:

$$\begin{array}{rl} H_1 & : \texttt{proper}\ E \\ Hg_1 & : \Gamma \rhd G\ E \\ IHg_1 & : \forall (c' : \texttt{context}), IA_1 \langle\!\langle \Gamma, c' \rangle\!\rangle \to \cdots \to IA_w \langle\!\langle \Gamma, c' \rangle\!\rangle \to c' \rhd G\ E \\ c' & : \texttt{context} \\ \overline{IP_w} & : \overline{IA_w} \langle\!\langle \Gamma, c' \rangle\!\rangle \\ \hline & \texttt{proper}\ E \end{array}$$

which is completed by assumption H_1.

Case $\dfrac{\texttt{proper}\ E \quad \Gamma, [D\ E] \rhd A}{\Gamma, [\texttt{All}\ D] \rhd A}\ \textit{b_all}$ **:**

This case is proven as above but with $m = p = 1$, $n = 0$, $c = \Gamma$, and $o = \texttt{All}\ D$. Define $Q_1\langle\!\langle \Gamma, \texttt{All}\ D \rangle\!\rangle := \texttt{proper}\ E$, $\gamma'_1\langle\!\langle \texttt{All}\ D \rangle\!\rangle := \emptyset$, and $F'_1\langle\!\langle \texttt{All}\ D \rangle\!\rangle := D\ E$ where $E : \texttt{expr con}$ is a signature variable. Then we are proving:

$$\begin{array}{rl} H_1 & : \texttt{proper}\ E \\ Hb_1 & : \Gamma, [D\ E] \rhd a_1 \\ IHb_1 & : \forall (c' : \texttt{context}), IA_1 \langle\!\langle \Gamma, c' \rangle\!\rangle \to \cdots \to IA_w \langle\!\langle \Gamma, c' \rangle\!\rangle \to c', [D\ E] \rhd a_1 \\ c' & : \texttt{context} \\ \overline{IP_w} & : \overline{IA_w} \langle\!\langle \Gamma, c' \rangle\!\rangle \\ \hline & \texttt{proper}\ E \end{array}$$

The goal `proper` E is provable by the assumption of the same form as in the previous case.

In the next two sections we will return to this idea of proofs about a specification logic from a generalized form of SL rule to prove properties of the SL once we have fully defined P_1 and P_2. The proof states in Figures 8.2 and 8.3 (the incomplete branches) will be roots of these explanations.

8.2 GSL Induction Part II: The Structural Rules Hold

Recall from Section 6.1 we prove the standard rules of weakening, contraction and exchange for both the goal-reduction and backchaining sequents as corollaries of `monotone` (Theorem 6.7) which states

$$\begin{aligned}&(\forall\ (c : \texttt{context})\ (o : \texttt{oo}),\\ &\qquad\qquad (c \rhd o) \to (P_1\ c\ o))\ \ \wedge\\ &(\forall\ (c : \texttt{context})\ (o : \texttt{oo})\ (a : \texttt{atm}),\\ &\qquad\qquad (c, [o] \rhd a) \to (P_2\ c\ o\ a))\end{aligned}$$

where P_1 and P_2 are defined as

$$\begin{aligned}P_1 :=& \lambda\ (c : \texttt{context})(o : \texttt{oo})\ .\\ &\quad \forall\ (c' : \texttt{context}), c \subseteq c' \to c' \rhd o\\ P_2 :=& \lambda\ (c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm})\ .\\ &\quad \forall\ (c' : \texttt{context}), c \subseteq c' \to c', [o] \rhd a\end{aligned}$$

We build on the inductive proof in Section 8.1 over the GSL to prove `monotone` for this new logic. Recall that when we took the proof as far as we could we had three remaining groups of branches to finish ($m+n+p$ subgoals), one group for rules with non-sequent premises depending on the context of the rule conclusion, and one for each kind of sequent premise (see Figures 8.2 and 8.3). We will continue this effort below, using the P_1 and P_2 defined for this theorem. This means we will have one induction assumption (i.e., $w = 1$) which is $IA_1⦇c, c'⦈ := c \subseteq c'$.

8.2.1 Sequent Subgoals

First we will prove the subgoals coming from the sequent premises, building on Figure 8.2 and using IA_1 as defined above. The proof state for goal-reduction (resp. backchaining) premises is

$$\begin{array}{rl}
\overline{H_m} & : \overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\
\overline{Hg_n} & : \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\
\overline{IHg_n} & : \forall\overline{(x_{n,s_n} : R_{n,s_n})}(c' : \texttt{context}), (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle) \subseteq c' \to c' \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle \\
\overline{Hb_p} & : \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p}) \\
\overline{IHb_p} & : \forall\overline{(y_{p,t_p} : S_{p,t_p})}(c' : \texttt{context}), (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle) \subseteq c' \to c', [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p} \\
c' & : \texttt{context} \\
IP_1 & : c \subseteq c' \\
\overline{x_{n,s_n}} & : \overline{R_{n,s_n}}\ (\mathit{resp.}\ \overline{y_{p,t_p}} : \overline{S_{p,t_p}}) \\
\hline
& (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle) \subseteq (c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle)\ (\mathit{resp.}\ (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle) \subseteq (c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle))
\end{array}$$

The goal is provable by `context_sub_sup` (Lemma 5.6) and assumption IP_1.

8.2.2 Non-Sequent Subgoals

Still to be proven are the subgoals for non-sequent premises. As seen in Section 8.1.2, the only rule of the SL whose corresponding subcase still needs to be proven is *g_dyn*. From Figure 8.3 and using P_1 and P_2 as defined here, we are proving

$$\begin{array}{rl}
H_1 & : D \in \Gamma \\
Hb_1 & : \Gamma, [D] \rhd a_1 \\
IHb_1 & : \forall(c' : \texttt{context}), \Gamma \subseteq c' \to c', [D] \rhd a_1 \\
c' & : \texttt{context} \\
IP_1 & : \Gamma \subseteq c' \\
\hline
& D \in c'
\end{array}$$

Unfolding the definition of context subset in IP_1 it becomes $\forall(o : \texttt{oo}), o \in \Gamma \to o \in c'$. Backchaining on this form of the goal gives subgoal $D \in \Gamma$, provable by assumption H_1.

In Section 8.1, we explored how to prove statements about the GSL for a restricted form of theorem statement. There were three classes of incomplete proof branches that had a final form shown in Figures 8.2 and 8.3. In Section 7.1 we saw how to derive the SL from the GSL. So here we have proven a structural theorem for the rules of the GSL in a general way that can be followed for any SL rule. ∎

8.3 GSL Induction Part III: Cut Rule Proven Admissible

Recall from Section 6.2 we are proving $\forall(\delta : \texttt{oo}), P\ \delta$ with P defined as

$$\begin{aligned} P : \texttt{oo} \to \texttt{Prop} := \lambda(\delta : \texttt{oo})\ .\\ (\forall(c : \texttt{context})(o : \texttt{oo}),\\ c \rhd o \to P_1\ c\ o) \wedge\\ (\forall(c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}),\\ c, [o] \rhd a \to P_2\ c\ o\ a), \end{aligned}$$

where

$$\begin{aligned} &P_1 : \texttt{context} \to \texttt{oo} \to \texttt{Prop} :=\\ &\quad \lambda(c : \texttt{context})(o : \texttt{oo})\ .\\ &\quad\quad \forall(c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to \underline{c' \rhd o}\\ &P_2 : \texttt{context} \to \texttt{oo} \to \texttt{atm} \to \texttt{Prop} :=\\ &\quad \lambda(c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm})\ .\\ &\quad\quad \forall(c' : \texttt{context}), c = (c', \delta) \to c' \rhd \delta \to \underline{c', [o] \rhd a} \end{aligned}$$

As in the GSL proof of `monotone` (Theorem 6.7), we build on the inductive proof in Chapter 8, unfolding P_1 and P_2 as defined here. Recall that we have now introduced assumptions and applied the appropriate generalized SL rule to the underlined sequents in the definition of P_1 and P_2. For the proof of cut admissibility, there are two induction assumptions from P_1 and P_2 (so $w = 2$). Define $IA_1\langle\!\langle c, c' \rangle\!\rangle := (c = (c', \delta))$ and $IA_2\langle\!\langle c, c' \rangle\!\rangle := c' \rhd \delta$, where δ is the cut formula in the cut rule.

8.3.1 Sequent Subgoals

First we will prove the subgoals coming from the sequent premises, building on Figure 8.2 and using IA_1 and IA_2 as defined above. For a moment we will ignore the outer induction over the cut formula δ. By ignore we mean let $\delta := \eta$ where $\eta : \texttt{oo}$, and we will not display the induction hypothesis for this induction. The proof state for goal-reduction premises (resp. backchaining premises) is

$$
\begin{array}{rl}
\overline{H_m} & : \overline{Q_m}\langle\!\langle c, o\rangle\!\rangle \\
\overline{Hg_n} & : \forall\overline{(x_{n,s_n} : R_{n,s_n})}, (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle) \\
\overline{IHg_n} & : \forall\overline{(x_{n,s_n} : R_{n,s_n})}(c' : \texttt{context}), \\
& \quad (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle) = (c', \eta) \to c' \rhd \eta \to c' \rhd \overline{F_n}\langle\!\langle o, \overline{x_{n,s_n}}\rangle\!\rangle \\
\overline{Hb_p} & : \forall\overline{(y_{p,t_p} : S_{p,t_p})}, (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle, [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p}) \\
\overline{IHb_p} & : \forall\overline{(y_{p,t_p} : S_{p,t_p})}(c' : \texttt{context}), \\
& \quad (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle) = (c', \eta) \to c' \rhd \eta \to c', [\overline{F'_p}\langle\!\langle o, \overline{y_{p,t_p}}\rangle\!\rangle] \rhd \overline{a_p} \\
c' & : \texttt{context} \\
IP_1 & : c = (c', \eta) \\
IP_2 & : c' \rhd \eta \\
\overline{x_{n,s_n}} & : \overline{R_{n,s_n}} \ (resp.\ \overline{y_{p,t_p}} : \overline{S_{p,t_p}}) \\
\hline
& (c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle = ((c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle), \eta)), (c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \eta) \\
& (resp.\ (c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle = ((c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle), \eta)), (c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle \rhd \eta))
\end{array}
$$

To prove the sequent subgoal $c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle \rhd \eta$ (resp. $c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle \rhd \eta$), first apply weakening and the new subgoal is $c' \rhd \eta$ (resp. $c' \rhd \eta$), provable by assumption IP_2.

The subgoals concerning context equality are proven by context lemmas and assumption IP_1. That is, we rewrite $((c' \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle), \eta)$ to $(c', \eta) \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle$ (resp. $(c' \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle), \eta$ to $(c', \eta) \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle$). The new subgoal is $c \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle = (c', \eta) \cup \overline{\gamma_n}\langle\!\langle o\rangle\!\rangle$ (resp. $(c \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle = (c', \eta) \cup \overline{\gamma'_p}\langle\!\langle o\rangle\!\rangle)$). Apply `context_sub_sup` (Lemma 5.6) to get assumption IP_1.

8.3.2 Non-Sequent Subgoals

In Section 8.1.2 we saw that the only rule of the SL whose corresponding subcase still needs to be proven is *g_dyn*. For the non-sequent subgoals we were able to complete the proof while the cut formula δ was represented as a parameter (and thus could have any formula structure). In the remaining non-sequent proof branch we need to make use of the nested structure of this induction. The proof of this subcase is shown in detail in Section 6.2.2.

In summary, the outer induction over δ gave seven cases for seven `oo` constructors. For each of these, an inner induction over sequents gave 15 new subgoals for 15 rules. We saw that for 14 of 15 rules, each rule has the same proof structure for every form of δ. The remaining subgoals were all for the rule *g_dyn* and were more challenging

due to the presence of a non-sequent premise that depends on the context of the conclusion.

∎

Using the generalized proof presented in this chapter and instantiating the GSL to the SL as in Section 7.1, we have found condensed proofs of `monotone` (Theorem 6.7) and `cut_admissible` (Theorem 6.8).

Chapter 9

Conclusion

In this thesis we have seen how the Coq implementation of Hybrid has been extended by the addition of a new specification logic (SL) based on hereditary Harrop formulas. This extension increases the class of object logics that Hybrid can reason about efficiently. The metatheory of this SL is formalized in Coq with proofs by mutual structural induction over the structure of sequent types. We saw the proofs of some specific subcases and the later insight that many of the cases are proven in a similar way. This led to the development of a generalized SL and form of metatheory statement that we could use to better understand the proofs of the SL metatheory.

9.1 Related Work

Throughout this thesis we have seen some mention of related work. Hybrid is a system implementing HOAS and as seen in Section 3.6 there are other systems with the same goal that also use this technique. As previously discussed, Hybrid is the only known system implementing HOAS in an existing trusted general-purpose theorem prover. See [9] and [10] for a more in-depth comparison of these systems on benchmarks defined there.

Although this work is contributing to the area of mechanizing programming language metatheory, the majority of the research presented here is applicable to the more general field of proof theory. We have seen proofs of the admissibility of structural rules of a specific sequent calculus, as well as a generalized sequent calculus which we tried to make only as general as necessary to encapsulate the specification logic presented earlier. Typically these kinds of proofs are by an induction on the height of derivations, but here we have proofs by mutual structural induction over dependent sequent types; the structural proofs in this thesis follow the style of Pfenning in [19]. The sequents in our logic do not have a natural number to represent the height of the derivation. So our presentation of this sequent calculus is perhaps more "pure" in some sense, but we may have lost a way to reason about some object logics.

It is not yet clear if building proof height into the definition sequents is necessary for studying some object logics. Overall, a better understanding of the relationship between proofs of the metatheory of sequent calculi by induction on the height of derivations versus over the structure of sequents is desirable.

9.2 Future Work

The highest priority future task is to show the utility of the new specification logic in Hybrid. This will be done by presenting an object logic that makes use of the higher-order nature (in the sense of unrestricted implicational complexity) of the new specification logic. Object logics that we plan to represent include:

- correspondence between HOAS and de Bruijn encodings of untyped λ-terms; this is our example OL of Chapter 3 but we have not yet proven Theorems 3.5.1 and 3.5.2 of Section 3.1 (see [25])
- structural characterization of reductions on untyped λ-terms (see [25])
- algorithmic specification of bounded subtype polymorphism in System F (see [21]); this comes from the POPLmark challenge [1]

We would also like to add automation to proofs containing object logic judgments so that the user of Hybrid will not need to be an expert user of proof assistants to be able to use the system.

The encoding of the new Hybrid SL follows the development of the specification logic of Abella as presented in [25], but it seems that the proofs of the admissibility of the structural rules differ between these systems. These proofs in Abella are not fully explained in [25] so some work will need to be done to compare the different proofs. Also, the proof of cut admissibility for this specification logic in Abella requires a third conjunct that we did not need for our proof:

$$\forall(c : \texttt{context})(o : \texttt{oo})(a : \texttt{atm}), c \rhd o \to c, [o] \rhd a \to c \rhd \langle\ a\ \rangle$$

Our understanding so far is that these proofs in Abella are over the height of derivations, which is an implicit parameter; it is not by structural induction over sequents in the fashion of the proofs founding this thesis.

The End.

Appendix A

Notations

Many symbols are used to denote values of different types. This allows us to impose some structure that is useful in understanding the work presented later, but is not actually built in to the system. We summarize the meta-variables used in the contributions chapters of this thesis. All symbols described here may also be seen with subscripts or the prime notation (i.e.$'$) when we need to talk about more than one term of a given type.

Symbol	Type	Description
α	`atm`	atom representing OL formula in pretty-printed inference rule notation for Coq statements
a	`atm`	atom representing OL formula in linear forms of Coq statements
A	`atm`	atom representing OL formula in SL inference rules
β	`oo`	SL formula in pretty-printed inference rule notation for Coq statements
o	`oo`	SL formula in linear form of Coq statements
G	`oo`	SL formula representing a goal in SL inference rules
D	`oo`	SL formula representing a clause in SL inference rules
Γ	`context`	context of assumptions in pretty-printed inference rule notation for Coq statements and SL inference rules
c	`context`	context of assumptions in linear form of theorem statements

We collect here the following additional notations seen in this thesis:

Definition	Type	Notation	Notes
`atom` a	`atm` $\rightarrow$ `oo`	$\langle\, a \,\rangle$	coerces an atom to a formula
`Imp` o_1 o_2	`oo` $\rightarrow$ `oo` $\rightarrow$ `oo`	$o_1 \longrightarrow o_2$	implication in SL formula (right associative)
`Conj` o_1 o_2	`oo` $\rightarrow$ `oo` $\rightarrow$ `oo`	$o_1 \& o_2$	conjunction in SL formula
`prog` a o	`atm` $\rightarrow$ `oo` $\rightarrow$ $Prop$	$a :- o$	parameter of SL representing OL inference rules

Bibliography

[1] Brian E. Aydemir et al. Mechanized metatheory for the masses: The PoplMark challenge. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005.

[2] Chelsea Battell and Amy Felty. The logic of hereditary Harrop formulas as a specification logic for Hybrid. In *11th International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP)*, ACM Digital Library, 2016.

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[4] Kaustuv Chaudhuri. Focusing strategies in the sequent calculus of synthetic connectives. In *Logic for PRogramming, Artificial Intelligence, and Reasoning*, volume 5330 of *LNCS*, pages 467–481. Springer, 2008.

[5] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 1988.

[7] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae Elsevier*, 34:381–392, 1972.

[8] Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

[9] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1—a common infrastructure for benchmarks. *CoRR*, abs/1503.06095, 2015.

[10] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2—a survey. *Journal of Automated Reasoning*, 55(4):307–372, 2015.

[11] Andrew Gacek. The Abella interactive theorem prover (system description). In *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.

[12] William A. Howard. The formulae-as-type notion of construction, 1969. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[13] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[14] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.

[15] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

[16] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[17] Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys*, 31(3es):1–6, 1999. Article No. 11.

[18] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[19] Frank Pfenning. Structural cut elimination I: Intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, 2000.

[20] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Languages Design and Implementation*, pages 199–208. ACM Press, 1988.

[21] Brigitte Pientka. Proof pearl: The power of higher-order encodings in the logical framework LF. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *LNCS*, pages 246–261. Springer, 2007.

[22] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Fifth International Joint Conference on Automated Reasoning (IJCAR)*, volume 6173 of *LNCS*, pages 15–21. Springer, 2010.

[23] Carsten Schürmann. The Twelf proof assistant. In *Twenty-Second International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 79–83. Springer, 2009.

[24] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. TypiCal Project, April 2014. Version 8.4pl4.

[25] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. Reasoning about higher-order relational specifications. In *15th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 157–168. ACM Press, 2013.

Index

λ-tree syntax, 2

Abella, 37
ambient logic, 30

backchaining, 20
backchaining rule, 49
Beluga, 38
binder, 2

calculus of constructions, 6
calculus of inductive constructions, 6
constructor, 21
contraction, 56, 82
Coq, 1, 6
cut admissibility, 61, 84

de Bruijn indices, 3, 28
dependent product, 8
dependent type, 12
derivation, 8

exchange, 56, 82

focusing, 42
forward chaining, 20

generalized specification logic (GSL), 73
goal-reduction rule, 49

higher-order abstract syntax, 2, 30
higher-order hereditary Harrop formulas, 39
Hybrid, 1, 27

induction, 22
induction assumption, 53, 77
induction hypothesis, 22
induction principle, 22
induction property, 22
inductive type, 21
inhabited, 10

mutual induction, 24
mutually inductive type, 24

object logic, 2, 28

polymorphism, 14
POPLmark, 2
proof state, 53
provable sequent, 8

reasoning logic, 30

simply typed λ-calculus, 11
specification logic, 3, 31
structural rules, 56
subgoal, 18
substitution, 8

tactic, 17, 20
tactical, 20
two-level logical framework, 3
type operator, 16

uniform proof, 42

valid sequent, 8

weakening, 56, 82