Thesis Proposal:

Practical Automated Theorem Proving with the Polarized Inverse Method

Sean McLaughlin

May 1, 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Jeremy Avigad	CMU, Philosophy
Robert Harper	CMU, SCS
Dale Miller	École Polytechnique
Frank Pfenning **(Chair)**	CMU, SCS
André Platzer	CMU, SCS

Abstract

Mechanically checked, formal reasoning as envisioned by Leibniz has in the past few decades become a reality. *Proof assistants* are programs that can rigorously and mechanically check the details of logical arguments. Such formal reasoning has applications to nearly all areas of computer science, from formal proof in mathematics to specifying and verifying critical properties of hardware, software systems, and security protocols.

Twelf is a proof assistant specialized for reasoning about deductive systems such as logics and programming languages. In its domain, it is one of the most powerful tools available. Significant Twelf developments cover many different application areas. For example, in the theory of programming languages, the formalization and proof of type safety for Standard ML represents the first machine verified proofs of important properties of a full-featured programming language. Proof carrying authentication frameworks based on Twelf allow for computer security based on formal proof. Proof carrying code and typed assembly languages also based on Twelf allow users to check properties of programs such as memory safety before they are executed.

The combination of higher order abstract syntax for representing structures with variable binding, the Elf logic programming interpretation, and the $\mathbf{M}_2^+$ meta-logic for reasoning about encodings make Twelf uniquely suited to reasoning about such systems.

For all the strengths of Twelf however, one glaring weakness is its lack of automation. As we will see in this proposal, the obligations of the Twelf user are much greater than they need to be. We propose the design and implementation of a *proof-producing meta-theorem prover* for Twelf. We conjecture that such a theorem prover will greatly facilitate the use of Twelf, making the resulting developments shorter and easier.

We intend to use the *polarized inverse method* as the basis of our automated reasoning. The polarized inverse method combines the well-known *inverse method* with more recent techniques such as *focusing* and *polarization*. We have already had demonstrable success with the polarized inverse method in a number of variants of first order intuitionistic logic. This proposal describes our prior work as well as our plan for building a Twelf theorem prover based on the polarized inverse method.

Contents

Part I
The Polarized Inverse Method

1 Introduction

> *We can judge immediately whether propositions presented to us are proved, and that which others could hardly do with the greatest mental labor and good fortune, we can produce with the guidance of symbols alone... As a result of this, we shall be able to show within a century what many thousands of years would hardly have granted to mortals otherwise.*
>
> – Leibniz

Mechanically checked, formal reasoning as envisioned by Leibniz has in the past few decades become a reality. *Proof assistants* are programs that can rigorously and mechanically check the details of logical arguments. Such formal reasoning has applications to nearly all areas of computer science, from formal proof in mathematics to specifying and verifying critical properties of hardware, software systems, and security protocols.

Twelf [Pfenning and Schürmann, 1999] is a proof assistant specialized for reasoning about deductive systems such as logics and programming languages. In its domain, it is one of the most powerful tools available. Significant Twelf developments cover many different application areas. For example, in the theory of programming languages, the formalization and proof of type safety for Standard ML [Lee et al., 2007] represents the first machine verified proofs of important properties of a full-featured programming language. Proof carrying authentication frameworks based on Twelf [Appel and Felten, 1999] allow for computer security based on formal proof. Proof carrying code and typed assembly languages also based on Twelf [Necula, 1997, Appel, 2001] allow users to check properties of programs such as memory safety before they are executed.

Twelf is based on the LF [Harper et al., 1993] dependent type theory, and a logic programming interpretation called *Elf* [Pfenning, 1989]. Twelf augments Elf with the meta-logic $\mathbf{M}_2^+$ [Schürmann, 2000] for reasoning about LF representations. In $\mathbf{M}_2^+$ it is possible to prove theorems about the encodings of deductive systems. These theorems are called *meta-theorems* because they are theorems *about* an encoded logic, rather than theorems of the logic itself. The $\mathbf{M}_2^+$ meta-theorems, combined with proofs of adequacy of the encodings, imply theorems about the encoded systems themselves.

Example For an illustration of the use of Twelf, we will compare an informal proof of type preservation for the simply typed lambda calculus with the corresponding Twelf proof. We will present this example only briefly to illustrate our main points. An expository explanation of the theory behind Twelf is [Harper and Licata, 2007]. We begin by defining terms and types. A term is either a constant such as $\langle\rangle$, a variable, an application, or a lambda abstractions. The judgements for term and type construction follow Martin-Löf's principle of judgments-as-types.

$$\frac{}{\Gamma \vdash \mathtt{tm}\ \langle\rangle} \qquad \frac{x \in \Gamma}{\Gamma \vdash \mathtt{tm}\ x} \qquad \frac{\Gamma, x \vdash \mathtt{tm}\ e}{\Gamma \vdash \mathtt{tm}\ (\lambda x.\ e)} \qquad \frac{\Gamma \vdash \mathtt{tm}\ e_1 \quad \Gamma \vdash \mathtt{tm}\ e_2}{\Gamma \vdash \mathtt{tm}\ (e_1 \cdot e_2)}$$

A type is either a base type such as `unit` or a function type.

$$\frac{}{\vdash \mathtt{tp}\ \mathtt{unit}} \qquad \frac{\vdash \mathtt{tp}\ \tau_1 \quad \vdash \mathtt{tp}\ \tau_2}{\vdash \mathtt{tp}\ (\tau_1 \Rightarrow \tau_2)}$$

These definitions are encoded in Twelf in Figure 1, lines 4-15. Note that lambda abstractions have a higher-order LF type, and are encoded in Twelf using higher-order abstract syntax [Pfenning and Elliott, 1988]. There is thus no need for a variable case. Extrinsic typing is defined using a context for variables. The Twelf encoding is on lines 19-28.

```
1  %%% Terms
2
3  tm : type.
4
5  @ : tm -> tm -> tm. %infix left 10 @.
6  lam : (tm -> tm) -> tm.
7  <> : tm.
8
9  %%% Types
10
11 tp : type.
12
13 unit : tp.
14 => : tp -> tp -> tp. %infix right 10 =>.
15
16 %%% Extrinsic Typing
17
18 of : tm -> tp -> type.
19
20 of/unit : of <> unit.
21
22 of/lam : of (lam T) (A => B)
23           <- ({x} of x A -> of (T x) B).
24
25 of/app : of (T1 @ T2) B
26           <- of T1 (A => B)
27           <- of T2 A.
28
29 %%% Evaluation
30
31 eval : tm -> tm -> type.
32
33 eval/unit : eval <> <>.
34 eval/lam : eval (lam T) (lam T).
35 eval/app : eval (T1 @ T2) T4
36             <- eval T1 (lam T3)
37             <- eval (T3 T2) T4.
38
39 %%% Type preservation
40
41 pres : of T A -> eval T T' -> of T' A -> type.
42 %mode pres +O1 +E -O2.
43
44 - : pres of/unit eval/unit of/unit.
45 - : pres (of/lam D) eval/lam (of/lam D).
46 - : pres
47       (of/app
48          (O2 : of T2 A)
49          (O1 : of T1 (A => B)))
50       (eval/app
51          (E2 : eval (T3 T2) T4)
52          (E1 : eval T1 (lam T3)))
53       (O3 : of T4 B)
54       <- pres O1 E1
55           (of/lam (O1' : {x} (of x A -> of (T3 x) B)))
56       <- pres (O1' T2 O2) E2 O3.
57
58 %block b : some {A : tp} block {x : tm} {px : of x A}.
59 %worlds (b) (pres _ _ _).
60 %total E (pres _ E _).
```

Figure 1: Twelf Encoding of Type Preservation

$$\frac{}{\Gamma \vdash \texttt{of}\ \langle\rangle\ \texttt{unit}}\ \text{of-unit} \qquad \frac{\Gamma, \texttt{of}\ x\ t_1 \vdash \texttt{of}\ e\ t_2}{\Gamma \vdash \texttt{of}\ (\lambda x.\ e)\ (t_1 \Rightarrow t_2)}\ \text{of-lam} \qquad \frac{\Gamma \vdash \texttt{of}\ e_1\ (t_2 \Rightarrow t_1) \quad \Gamma \vdash \texttt{of}\ e_2\ t_2}{\Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t_1}\ \text{of-app}$$

Evaluation is defined in a similar fashion. The Twelf equivalent is on lines 32-38.

$$\frac{}{\vdash \texttt{eval}\ \langle\rangle\ \langle\rangle}\ \text{eval-lam} \qquad \frac{}{\vdash \texttt{eval}\ (\lambda x.\ e)\ (\lambda x.\ e)}\ \text{eval-lam}$$

$$\frac{\vdash \texttt{eval}\ e_1\ (\lambda x.\ e_3) \quad \vdash \texttt{eval}\ ([e_2/x]\ e_3)\ e}{\vdash \texttt{eval}\ (e_1 \cdot e_2)\ e}\ \text{eval-app}$$

Now we can state the type preservation theorem.

Theorem. *If* $\Gamma \vdash \texttt{of}\ e\ \tau$ *and* $\vdash \texttt{eval}\ e\ e'$ *then* $\Gamma \vdash \texttt{of}\ e'\ \tau$.

Proof. Let $\mathcal{D}$ be the derivation of $\vdash \texttt{eval}\ e\ e'$ and $\mathcal{E}$ the derivation of $\Gamma \vdash \texttt{of}\ e\ \tau$. The proof is by induction on $\mathcal{D}$.

Case:

$$\mathcal{D} = \frac{}{\vdash \texttt{eval}\ \langle\rangle\ \langle\rangle}\ \text{eval-unit}$$

Then $e = e' = \langle\rangle$, so $\mathcal{E}$ is a derivation of $\Gamma \vdash \texttt{of}\ e'\ \tau$.

Case:

$$\mathcal{D} = \frac{}{\vdash \texttt{eval}\ (\lambda x.\ e_0)\ (\lambda x.\ e_0)}\ \text{eval-lam}$$

Then $e = e' = \lambda x.\ e_0$, so $\mathcal{E}$ is a derivation of $\Gamma \vdash \texttt{of}\ e'\ \tau$.

Case:

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdash \texttt{eval}\ e_1\ (\lambda x.\ e_3) & \vdash \texttt{eval}\ ([e_2/x]\ e_3)\ e'\end{array}}{\vdash \texttt{eval}\ (e_1 \cdot e_2)\ e'}\ \text{eval-app}$$

$$\mathcal{E} = \frac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \vdash \Gamma \vdash \texttt{of}\ e_1\ (\tau_2 \Rightarrow \tau) & \vdash \Gamma \vdash \texttt{of}\ e_2\ \tau_2\end{array}}{\vdash \Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ \tau}\ \text{of-app}$$

1. $e = e_1 \cdot e_2$	Assumption
2. $\Gamma \vdash \texttt{of}\ (\lambda x.\ e_3)\ (\tau_2 \Rightarrow \tau)$	Induction with $\mathcal{D}_1, \mathcal{E}_1$
3. For any x, if $\Gamma \vdash \texttt{of}\ x\ \tau_2$ then $\Gamma \vdash \texttt{of}\ e_3\ x\ \tau$	Inversion on 2.
4. $\Gamma \vdash \texttt{of}\ (e_3\ e_2)\ \tau$	(3) with $\mathcal{E}_2$
5. $\Gamma \vdash \texttt{of}\ e'\ t$	Induction with (4) and $\mathcal{D}_2$

□

The corresponding Twelf proof is found on lines 42-56.

Analysis Twelf developments consist of three kinds of code. The first is the *specification* of a logical system. This is the process of defining the LF constants that encode the given logic, and the relationships between the constants. In our example, the specification of types, terms, typing, and evaluation is found in lines 1-37 of Figure 1. Second, there are the statements of the theorems (and associated lemmas) that the user wishes to prove about the encoded system. Theorems are encoded as LF type families. In the example, the type family `eval` on line 41 along with its mode declaration on line 42 corresponds to the statement that evaluation preserves typing. Finally, there are the relations that constitute the *proofs* of the meta-theorems. These relations are realized as Elf logic programs. The relations witnessing the proof of type preservation is found on lines 44-56. (The declarations on lines 58-60 asks that Twelf check that the proof is correct.) The proofs of meta-theorems often receives the lion's share of both the development time and sheer code bulk. One of the authors of the Standard ML development estimated a ratio of proof to specification of 4 to 1 in the simplest parts, to 20 to 1 in the more difficult sections. While Twelf can check a proof automatically, the Twelf user must write proofs by hand in full detail (cf. lines 44-56).

Thesis Proposal This thesis deals with automating the creation of Twelf proofs. A capable Twelf theorem prover would be able to completely eliminate large portions of these proofs, allowing the user more time for the creative tasks of encoding systems and determining the necessary lemmas. At the same time it would reduce or even eliminate the time to modify proofs if the specification changes. In our simple example, our proposed theorem prover would obviate the need for writing the proof of type preservation on lines 44-56, eliminating %20 of the code. In larger developments where the amount of proof to specification is much greater, we could hope to eliminate a much higher percentage.

In our proposed thesis, we intend to build a *practical* and *useful* proof-producing meta-theorem prover for Twelf[1]. To be *practical*, it should be able to prove many of the routine lemmas in a proof development. To be *useful* it should be seamlessly integrated into Twelf and allow the user to easily check, inspect, and record the evidence for the theorem prover's judgments.

To build a practical and useful meta-theorem prover for Twelf, we intend to use the *polarized inverse method*. The polarized inverse method is a refinement of the focused inverse method used by Chaudhuri [Chaudhuri, 2006] to build a theorem prover for linear logic. It is comprised of three primary tools. The *inverse method* [Maslov, 1964, Degtyarev and Voronkov, 2001b] uses forward saturation, generalizing resolution to non-classical logics satisfying the subformula property and cut elimination. *Focusing* [Andreoli, 1992, Liang and Miller, 2007] reduces the search space in a sequent calculus by restricting the application of inference rules based on the polarities [Lamarche, 1995] of the connectives and atomic formulas. Assigning *explicit polarities* to subformulas using polarized logic in turn allows for additional flexibility in guiding proof search, often with dramatic effect [McLaughlin and Pfenning, 2008, McLaughlin and Pfenning, 2009]. The bulk of this thesis will be devoted to defining, proving properties of, and implementing the polarized inverse method.

> **Thesis Statement.** The polarized inverse method is a fruitful basis for automated reasoning in non-classical logics. In particular, it forms the basis for a practical and useful proof-producing meta-theorem prover for the Twelf proof assistant.

[1] Twelf originally included a theorem prover, but due to various shortcomings it is no longer used in practice. We will discuss the existing theorem prover in more detail in Section 9.

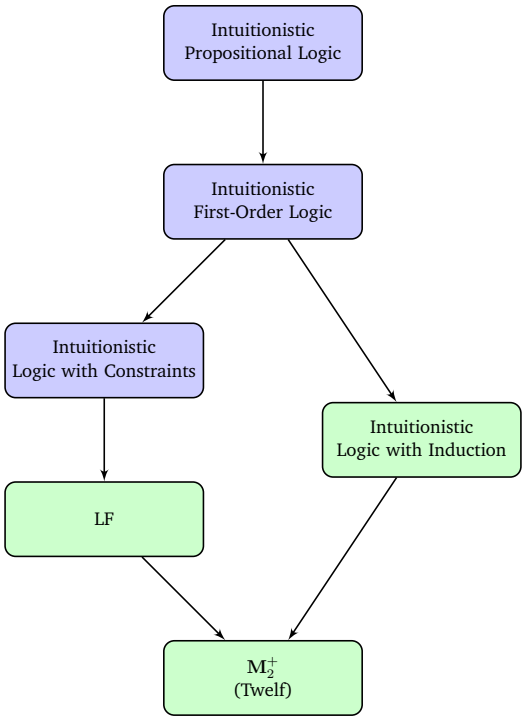

Figure 2: Logical systems discussed in this proposal.

Because developing a useful proof-producing meta-theorem prover is a complicated task, we will decompose the problem by building six different theorem provers for logics of increasing complexity. Each logic will address some aspect of the requirements of a Twelf meta-theorem prover, and will build on the previous parts. These logical systems are shown in Figure 2. Logics whose implementations are complete, or nearly so, are shown in blue. Implementations that are not started, or barely begun are shown in green. While the goal of the thesis is a Twelf theorem prover, results related to the component systems are of independent interest. The rest of the proposal is organized as follows.

The Inverse Method. In Section 2 we will review the main tool of our developments: the inverse method. Using intuitionistic propositional logic as an example, we give a thorough description of the method of theorem proving that will be used in the remainder of the proposal.

Imogen. In Section 3 we introduce our inverse method framework, called *Imogen*, and describe the principal algorithms involved in inference and redundancy elimination.

Intuitionistic Propositional Logic. In Section 4 we continue the discussion of propositional logic. We introduce focusing and polarities, give a polarized backward calculus, and finally a polarized forward calculus in which we search for proofs using the inverse method. IPL is a good introduction to our methods because most of the properties exhibited by the other calculi are present there in their simplest form. Finally, we detail Imogen's performance on established benchmark problems for IPL from the ILTP [Raths et al., 2007] library of intuitionistic propositional problems. Imogen is competitive with the best theorem provers for IPL.

Intuitionistic First-Order Logic. In Section 5 we move to intuitionistic first order logic. There we enrich focusing and the inverse method with first order quantification. The important operations of *subsumption*, *rule matching*, and *contraction* become significantly more difficult. We extend the results for propositional logic to this richer setting, and describe the results of our implementation on the ILTP library of first-order intuitionistic problems. In short, Imogen has the best performance of any existing intuitionistic theorem prover.

Intuitionistic Logic with Constraints. LF adds to predicate logic a number of additional complications, the most significant of which is the undecidability of its unification problem [Goldfarb, 1981]. Therefore *constraints* must be added to the inverse method calculus for LF to handle unification equations that fall outside of the decidable *pattern fragment* [Miller, 1992]. We anticipate this development in Section 6 by developing an inverse method for a first order intuitionistic sequent calculus with constraints. We define the intended semantics of constraints, and show how to use constraints in the forward direction with the inverse method. We also describe an implementation that handles explicit unification equations and disequations.

Intuitionistic Logic with Induction. The primary component of the proof of type preservation is induction on LF types. Induction in LF is complicated by many factors. One must handle an undecidable unification problem, explicit substitutions, and the regular world hypothesis [Schürmann, 2000]. Many of the challenges of an inductive theorem prover for LF are already present in the simpler setting of first order logic. Thus, we will study the polarized inverse method with induction in this simpler setting. In Section 7 we propose the implementation of a prototype for an inductive theorem prover for first order logic by extending first order logic with fixpoint operators and induction schemas.

LF. In Section 8 we continue to our first primary objective, a focused sequent calculus, inverse method, and theorem prover for LF. This provides an alternative method for searching for objects of a given LF type, and forms an important component of the meta-theorem prover.

$\mathbf{M}_2^+$. In Section 9 we introduce our ultimate goal of a practical and useful inductive meta-theorem prover for the $\mathbf{M}_2^+$ logic of Twelf. We describe the general structure such a prover will take and argue why we think the task is feasible in the near future.

2 The Inverse Method

We begin by describing the inverse method [Maslov, 1964, Mints, 1993] which is the principal reasoning tool of the proposal. This section closely follows [Degtyarev and Voronkov, 2001b]. There are two broad categories of proof search methods. *Backward chaining* (also called *backward search* and *top-down search*) results from decomposing a goal into subgoals, reading an inference rule from its conclusion to its premises.[2] Subgoals are solved recursively. Prolog's SLD resolution and tableaux [Hähnle, 2001] are examples of backward chaining. The other method is called *forward chaining* (also *forward search* and *bottom-up search*). In forward chaining one maintains a database of known *facts*, usually clauses or sequents. The database begins the search process containing only axioms. Search proceeds by repeatedly matching the premises of an inference rule with facts from the database to generate new facts, a process known as *rule application* or *rule matching*. This process is repeated until the goal is found. Resolution [Robinson, 1965, Bachmair and Ganzinger, 2001] and the inverse method are examples of forward chaining search algorithms. The inverse method differs from resolution because while resolution operates on clauses of literals in a normal form, the inverse method works directly with formulas. This makes it suitable for logics with no convenient normal form such as intuitionistic and linear logic. The inverse method was first devised by Gentzen [Gentzen, 1934] and used to prove the decidability of intuitionistic propositional logic. The name "inverse method" was coined in [Maslov, 1964]. For a more detailed history with copious references, see [Degtyarev and Voronkov, 2001b] Section 9.1. As we will see, some sequent calculi are designed for forward search, others for backward search. We call them *forward (sequent) calculi* and *backward calculi* respectively.

In this section we briefly outline the steps involved in constructing a forward sequent calculus for a logic, and describe the basic principles of the inverse method. We will use intuitionistic propositional logic (IPL) as a running example. In Section 2.1 we briefly remind the reader of the backward (tableaux) rules for IPL. In Section 2.2 we motivate and define the forward (inverse method) calculus. Section 2.3 discusses the main elements of inverse method. Section 2.4 presents some standard optimizations that will recur throughout the proposal. Section 2.5 gives a complete example.

2.1 Intuitionistic Propositional Logic

Formulas of IPL have the following form:

$$\text{Formulas } A ::= P \mid A \wedge A \mid A \vee A \mid A \supset A \mid \top \mid \bot$$

The meta-variable P ranges over atomic propositions. We consider $\neg A$ and $A_1 \Leftrightarrow A_2$ as abbreviations for $A \supset \bot$ and $(A_1 \supset A_2) \wedge (A_2 \supset A_1)$ respectively. We assume the reader is familiar with the proof theory of IPL, in particular that the backward sequent calculus is sound and complete for the usual semantics of natural deduction. We refer the reader to [Troelstra and Schwichtenberg, 1996] for such background. Backward sequents are written $\Gamma \Longrightarrow A$ where Γ is a *multiset* of formulas and A is a formula.[3] Γ is called the *antecedents* and A is called the *succedent*. Multisets will be used exclusively for antecedents in this proposal. Multisets, which allow multiple occurrances of the same formula that can be distinguished, are crucial for a computational interpretation of sequents (e.g. in the translation of sequent proofs into natural deduction proof terms). The backward sequent calculus for IPL is given in Figure 3. This is essentially the propositional fragment of Gentzen's system LJ. We will sometimes abuse notation and write $\Gamma \Longrightarrow A$ as if it were the judgment "$\Gamma \Longrightarrow A$ is derivable". This will be a common abuse of notation in this proposal, and we will not

[2] Thus, rather confusingly, in top-down search the inference rules are read from the lower line to the upper line.

[3] Throughout the proposal we will use double arrows ($\Gamma \Longrightarrow A$) for backward sequents and single arrows ($\Gamma \longrightarrow A$) for forward sequents.

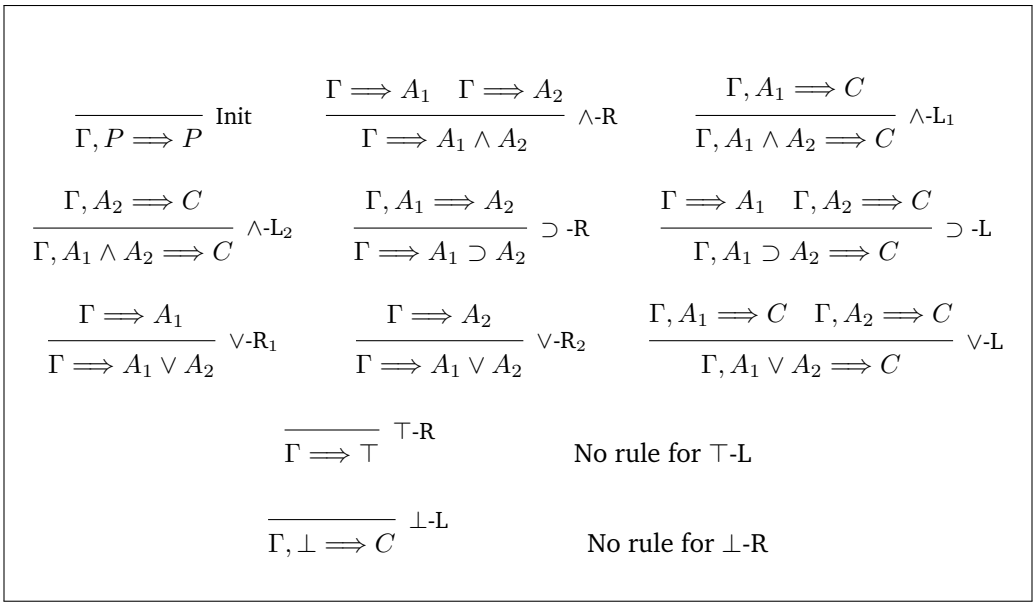

Figure 3: IPL: The Backward Calculus

$$
\begin{array}{lll}
A_1^r \leq (A_1 \wedge A_2)^r & A_2^r \leq (A_1 \wedge A_2)^r & A_1^l \leq (A_1 \supset A_2)^r \\
A_1^r \leq (A_1 \vee A_2)^r & A_2^r \leq (A_1 \vee A_2)^r & A_2^r \leq (A_1 \supset A_2)^r \\
A_1^l \leq (A_1 \wedge A_2)^l & A_2^l \leq (A_1 \wedge A_2)^l & A_1^r \leq (A_1 \supset A_2)^l \\
A_1^l \leq (A_1 \vee A_2)^l & A_2^l \leq (A_1 \vee A_2)^l & A_2^l \leq (A_1 \supset A_2)^l
\end{array}
$$

Figure 4: IPL: Signed Subformulas

mention it further. Another notational convention we will adopt is in the left rules of backward calculi. We will not copy the principle formula to the premises, but assume it is present in the meta-variable standing for the non-principle formulas (Γ in Figure 3).

The Subformula Property One important property of the backward calculus is *the signed subformula property*. Simply stated, any formula occurring in a backward derivation of a formula A is a signed subformula of A.

Definition 2.1 (Signed subformulas)**.** A *sign* is either *left* (written "l") or *right* (written "r").[4] The *signed subformulas* of a formula are given by the reflexive transitive closure of the rules in Figure 4. The *signed subformulas of a sequent* $\Gamma \Longrightarrow A$ are the signed subformulas of the elements of the sequent $\{\Gamma^l, A^r\}$.

[4]This notation is nonstandard. The reason we choose left and right instead of the traditional *positive* and *negative* is that one of the principal concepts in this proposal deals with the *polarities* of formulas. Polarities are also called positive and negative. The polarity of a formula has nothing to do with its sign. Moreover, the two concepts are sometimes used side by side. To avoid confusion, we chose the nonstandard *left/right* notation.

Theorem 2.1 (The Signed Subformula Property)**.** *Any formula occurring in a sequent in a derivation of* $\Gamma \Longrightarrow A$ *is a signed subformula of* $\Gamma \Longrightarrow A$*.*

Proof. Routine inspection of the inference rules. □

The signed subformula property will be critical in reducing the search space of the forward calculus.

2.2 Forward Sequent Calculus

We will now design a forward calculus based on the backward rules that is suitable for forward reasoning. In order for the forward chaining strategy to be complete, a set of inference rules $\mathcal{L}$ needs to satisfy the *finite rule property*.

Definition 2.2 (Finite rule property)**.** A set of inference rules $\mathcal{L}$ satisfies the finite rule property if the following two conditions hold:

1. $\mathcal{L}$ has a finite number of axioms.
2. Given a finite number of sequents, there is only a finite number of inferences (one-step derivations) of $\mathcal{L}$ that are applicable to these sequents.

A calculus satisfying the finite rule property can then naively enumerate all consequences of the axioms. Examining the backward rules, there are a number of ways the finite rule property fails. To have a finite number of axioms, we must maintain only a finite number of axioms (initial sequents). Initial sequents arise from the rules Init, $\top$-R, and $\bot$-L. We define forward sequents such that we can restrict the calculus to a finite number of instances of these rules.

Definition 2.3 (Forward sequents)**.** Forward sequents have the form $\Gamma \longrightarrow \gamma$ where Γ is a multiset of formulas and γ is a *set* which is either empty or a singleton. Γ is still called the *antecedents* and γ the *succedent*. We write $\cdot$ for the empty set.

The intention is that if $\Gamma \longrightarrow \cdot$ is derivable, then $\Gamma' \Longrightarrow A$ is derivable for any formula A, and any $\Gamma' \supseteq \Gamma$. The second condition means that *weakening* will not be an admissible rule in the forward calculus. We can now finitely simulate the initial rules:

Backward	Forward
$\dfrac{}{\Gamma, \bot \Longrightarrow C}\ \bot\text{-L}$	$\dfrac{}{\bot \longrightarrow \cdot}\ \bot\text{-L}$
$\dfrac{}{\Gamma \Longrightarrow \top}\ \top\text{-R}$	$\dfrac{}{\cdot \longrightarrow \top}\ \top\text{-R}$
$\dfrac{}{\Gamma, P \Longrightarrow P}\ \text{Init}$	$\dfrac{}{P \longrightarrow P}\ \text{Init}$

Note that while the rule Init still appears to have an infinite number of instances, in the proof of any particular formula A it can only be instantiated with atomic formulas occurring as a subformula of A. Thus the subformula property assures us that there are a finite number of axioms, thus satisfying the first part of the finite rule property.

Now we turn to the problematic rules that introduce an arbitrary new formula in the conclusion that does not appear in a premise. For example, A_2 in $\wedge$-L$_1$. Again, this apparent infinitude of rules is made finite by

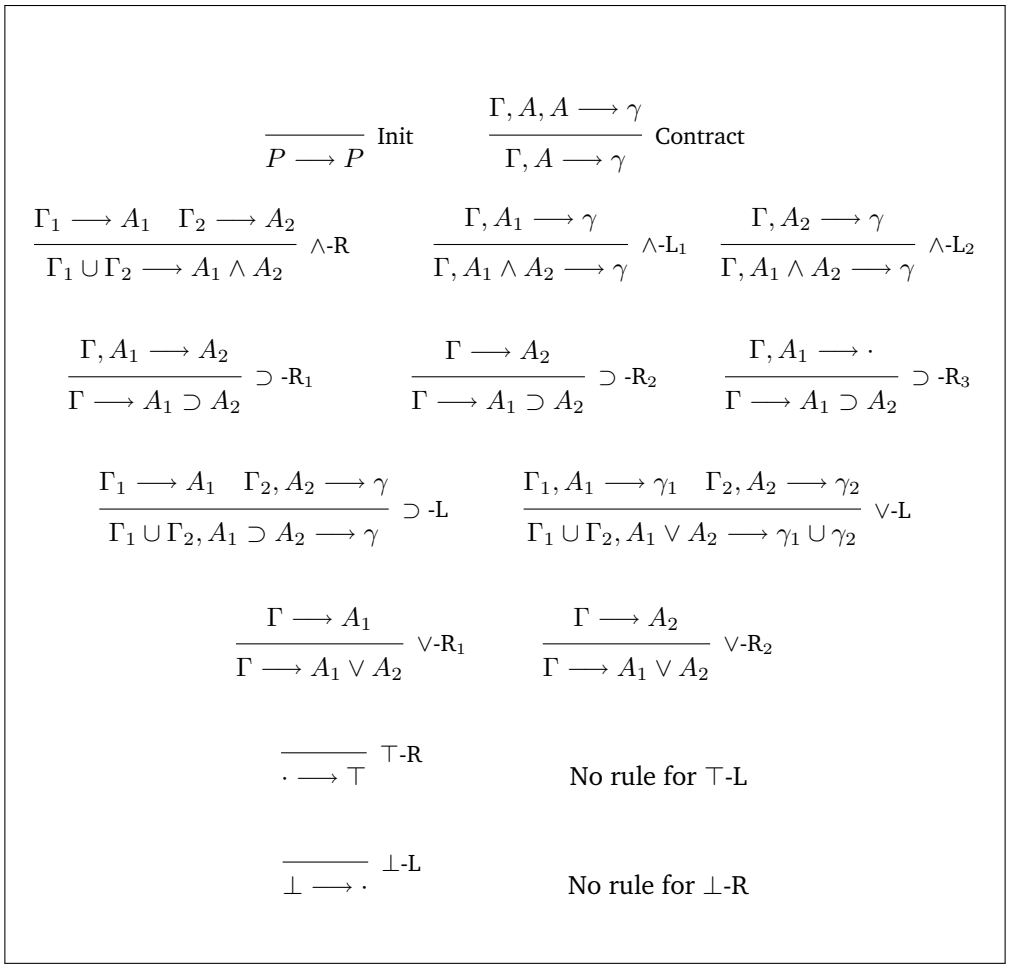

Figure 5: IPL: Forward Sequent Calculus

the subformula property: only formulas $A_1 \wedge A_2$ that are subformulas of the initial formula can be generated by the rule.

The final issue regards the antecedents Γ in multi-premise rules. Define a formula of an inference rule as *principal* if it is introduced by the rule. For instance, in the rule $\wedge$-R, the formula $A_1 \wedge A_2$ is principal. Call a formula *non-principal* if it is not principal, and not a subformula of a principal formula. Forcing the antecedents Γ to be the same in both branches is problematic. We would essentially have to guess how to correctly weaken the antecedents of the two known sequents so that the rule would be applicable. This is solved by combining the non-principal antecedents, rather than assuming they are identical. For example The rule $\wedge$-R becomes:

$$\frac{\Gamma \Longrightarrow A_1 \quad \Gamma \Longrightarrow A_2}{\Gamma \Longrightarrow A_1 \wedge A_2} \wedge\text{-R} \qquad \frac{\Gamma_1 \longrightarrow A_1 \quad \Gamma_2 \longrightarrow A_2}{\Gamma_1 \cup \Gamma_2 \longrightarrow A_1 \wedge A_2} \wedge\text{-R}$$

The full forward sequent calculus $\mathcal{P}_{\text{inv}}$ is shown in Figure 5. Figure 6 shows a backward derivation. Figure 7 its corresponding forward derivation. We have the following theorems:

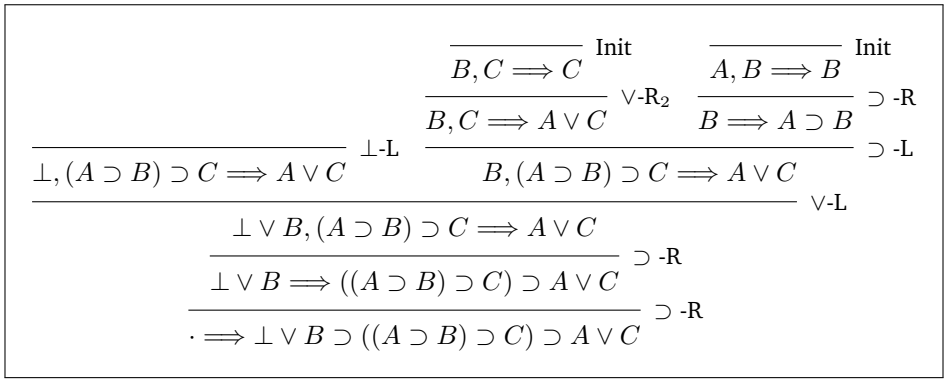

Figure 6: Backward Derivation

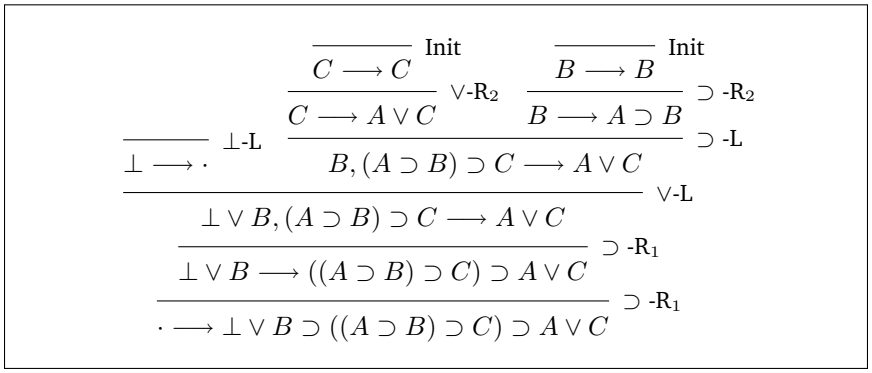

Figure 7: Forward Derivation

Theorem 2.2 (Soundness)**.** *If* $\Gamma \longrightarrow A$ *is derivable then so is* $\Gamma \Longrightarrow A$*. If* $\Gamma \longrightarrow \cdot$ *is derivable then for any* A*,* $\Gamma \Longrightarrow A$ *is derivable.*

Proof. By induction on the derivation. □

Theorem 2.3 (Completeness)**.** *If* $\Gamma \Longrightarrow A$ *is derivable then there is a* $\Gamma' \subseteq \Gamma$ *and* $\gamma \subseteq \{A\}$ *such that* $\Gamma' \longrightarrow \gamma$ *is derivable.*

Proof. Standard. E.g. [Degtyarev and Voronkov, 2001b]. □

2.3 The Inverse Method

The forward rules (Figure 5), together with the subformula property (Theorem 2.1) yield a complete method for proof search. Suppose we are given a formula ξ for which we want to find a proof. By the subformula property, we only need to consider inference rules that generate signed subformulas of ξ. We can thus specialize the forward inference rules to subformulas of ξ, yielding a finite set of axioms and rules, called $\mathcal{P}^{\xi}_{\text{inv}}$. Then $\mathcal{P}^{\xi}_{\text{inv}}$ satisfies the finite rule property, and we can carry out a simple search procedure. We maintain

a database of sequents we have derived so far. This database initially contains only the axioms. At each step, we search for instances of the inference rules whose premises are instances of sequents in the database. This process produces derivations of new sequents. To avoid generating the same sequent over and over, adding redundant sequents to the database, we introduce the subsumption relation.

Definition 2.4 (Subsumption)**.** A sequent $\Gamma_1 \longrightarrow \gamma_1$ *subsumes* a sequent $\Gamma_2 \longrightarrow \gamma_2$ if $\Gamma_1 \subseteq \Gamma_2$ and $\gamma_1 \subseteq \gamma_2$. A set (database) of sequents subsumes a sequent if one of its elements does. For sequents Q_1, Q_2, write $Q_1 \preceq Q_2$ if Q_1 subsumes Q_2.

If a sequent is derived that is subsumed by the database, that sequent is discarded since it can not lead to any new sequents. We continue matching rules to sequents until either the goal is subsumed or no more progress can be made. In the later case, the set of facts is said to be *saturated*. A saturated database is evidently a proof that the formula is not derivable. Indeed, since no new sequents can be derived, and the procedure is complete, the formula is not derivable. Pseudocode for the algorithm is roughly in Algorithm 2.1, and a graphical depiction in Figure 8. The line numbers of the image correspond to the lines of Algorithm 2.1. The function matchRule$(R, \text{KeptSequents}, \text{ActiveSequents})$ tries to match a sequent of KeptSequents to a premise of R. If the match succeeds, it tries to find matches for the remaining premises in KeptSequents $\cup$ ActiveSequents.

Algorithm 2.1 The Imogen Fixed-Rule Loop

```
1:  Input a formula ξ
2:  NewSequents ← ∅
3:  KeptSequents ← the axioms of P^ξ_inv
4:  Rules ← the rules of P^ξ_inv
5:  while KeptSequents ≠ ∅ do
6:    for R ∈ Rules do
7:      NewSequents ← matchRule(R, KeptSequents, ActiveSequents) ∪ NewSequents
8:    end for
9:    if ∃s ∈ NewSequents. s ⪯ · ⟶ ξ then
10:     print "Proof found. ξ is true."
11:     return
12:   end if
13:   ActiveSequents ← KeptSequents ∪ ActiveSequents
14:   KeptSequents ← NewSequents
15:   NewSequents ← ∅
16: end while
17: print "The database is saturated. ξ is false."
18: return
```

Theorem 2.4 (Soundness)**.** *If the algorithm of Figure 2.1 prints "ξ is true", then $\cdot \longrightarrow \xi$. If the algorithm prints "ξ is false", then there is no derivation $\cdot \longrightarrow \xi$.*

Proof. By the soundness of the forward calculus. Each sequent that enters the database is the result of matching an inference rule to a sequent. By induction, the new sequent has a sound derivation. □

Theorem 2.5 (Completeness)**.** *If $\cdot \longrightarrow \xi$, then the algorithm will eventually print "ξ is true".*

Proof. The algorithm enumerates all proofs whose sequents contain subformulas of the goal. By the subformula property, any proof of ξ is of this form. □

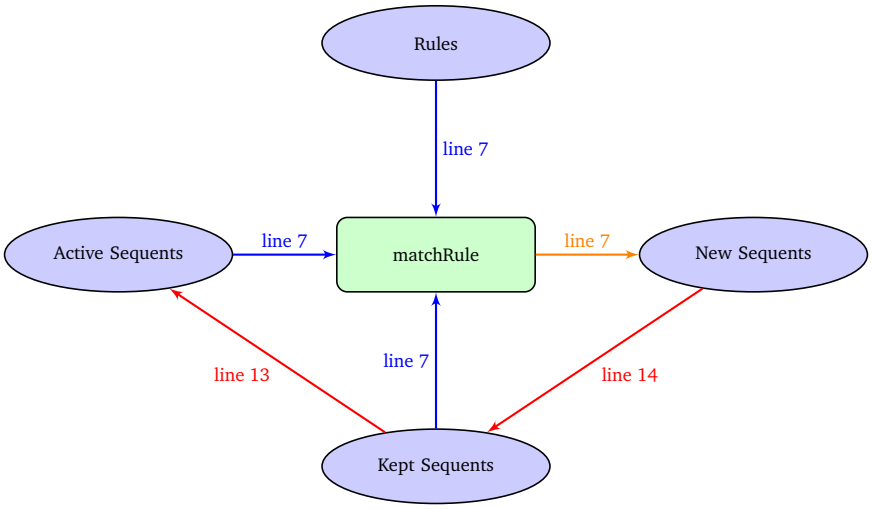

Figure 8: Imogen's Fixed-Rule Loop

In IPL, this procedure gives us even more. Since there are only a finite number of subformulas, there are only a finite number of (fully contracted) sequents. Because of subsumption, we can only ever add a finite number of sequents to the database. Therefore, the process just described terminates, and is a decision procedure for IPL.

2.4 Optimizations

There are obviously many improvements that can be made to the naive algorithm. Much of the work of this proposal will be pruning the search space by considering a subset of possible forward derivations and showing that any formula with a derivation in the forward calculus has a derivation in the subset. Focusing is an example of such an optimization which we will describe in Section 4. In addition to logical restrictions of the space of possible derivations, there are numerous non-logical optimizations that play an important part in the implementation of a practical and efficient theorem prover. The most significant of these are discussed briefly in this section.

Labeling. The rule matching process compares formulas for equality (and later, unifiability). Working directly with formulas is computationally expensive. A method to avoid frequently traversing entire formulas is to *label* subformulas by abstracting the formula over its free variables and assigning it a predicate label. This allows equality checks to be done quickly. It is possible to formalize this idea using a *name calculus* [Degtyarev and Voronkov, 2001b]. Labeling leads to a deeper optimizations, such as a *path calculus* [Degtyarev and Voronkov, 2001b]. Instead of only storing labels, a path calculus also remembers the location of a subformula in the goal formula. Saving this information allows for additional optimizations. For instance, if $(A_1 \vee A_2)^l$ is a subformula of ξ, it is easy to show that no sequent that has both A_1 and A_2 in its antecedents is necessary to build a derivation of ξ. Such sequents can be discarded from the database.

Redundancy Elimination. An essential optimization in forward-chaining style theorem proving is redundancy elimination. Due to the latency of memory access, the limiting factor in forward proof search is the size of the database of facts [Ramakrishnan et al., 2001]. Removing any redundant sequents from the database

is essential to reasonable performance. A newly derived sequent that is subsumed by an existing sequent of the database is said to be *forward subsumed*. A newly derived sequent can also subsume existing database sequents. Those existing sequents are said to be *backward subsumed*. A sequent is called *redundant* if is either forward or backward subsumed. Much research into theorem proving implementations is directed at quickly checking redundancy of a sequent with respect to a database.

Term Indexing. Since redundancy elimination, and therefore subsumption checking in the database, is such an important operation, it is essential to use data structures for which finding possible subsumption candidates are efficient. *Term indexing* is the study of the efficient storage and retrieval of terms satisfying certain properties. For instance, we may wish to retrieve all the terms that are instances, or generalizations of another term. Many term indexing data structures have been studied for both forward and backward subsumption. Good general references are [Graf, 1996, Ramakrishnan et al., 2001]. We will see some statistics for subsumption in Sections 4 and 5

2.5 Example

Here we give a concrete example of the Algorithm 2.1 proving the formula

$$\xi = (\bot \vee B) \supset ((A \supset B) \supset C) \supset (A \vee C)$$

Labeling. First we label the signed subformulas of ξ.

$$\begin{array}{lll} L_1^+ = A^+ \vee C^+ & L_2^+ = A^- \supset B^+ & L_3^- = L_2^+ \supset C^- \\ L_4^+ = L_3^- \supset L_1^+ & L_5^- = \bot^- \vee B^- & L_6^+ = L_5^- \supset L_4^+ \end{array}$$

Rule Specialization. Now we specialize the rules of $\mathcal{P}_{\text{inv}}$ to the calculus $\mathcal{P}_{\text{inv}}^{\xi}$ by generating rules for the labels $L_1, \ldots, L_6$. As an implementation trick, we combine the three different rules for $\supset$-R from Figure 5 into one rule by following the convention that a given formula doesn't need to be present for a match to succeed. For instance, the sequents $A \longrightarrow B$, $B \longrightarrow B$, and $A \longrightarrow \cdot$ can all match the rule

$$\frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow L_2}$$

For brevity we can omit the non-principal antecedents Γ from the rule; it is implicit. We also omit the conclusion formula when it is not a principal formula of the rule. We call such Γ and non-principal conclusions *side formulas*. Written in this abbreviated style, the inference rules for ξ are

$$\frac{\cdot \longrightarrow A}{\cdot \longrightarrow L_1}\ R_{1a} \qquad \frac{\cdot \longrightarrow C}{\cdot \longrightarrow L_1}\ R_{1b} \qquad \frac{A \longrightarrow B}{\cdot \longrightarrow L_2}\ R_2 \qquad \frac{\cdot \longrightarrow L_2 \quad C \longrightarrow \cdot}{L_3 \longrightarrow \cdot}\ R_3$$

$$\frac{L_3 \longrightarrow L_1}{\cdot \longrightarrow L_4}\ R_4 \qquad \frac{\bot \longrightarrow \cdot \quad B \longrightarrow \cdot}{L_5 \longrightarrow \cdot}\ R_5 \qquad \frac{L_5 \longrightarrow L_4}{\cdot \longrightarrow L_6}\ R_6$$

The initial axioms are

$$A \longrightarrow A \qquad B \longrightarrow B \qquad C \longrightarrow C \qquad \bot \longrightarrow \cdot$$

Goal

$\cdot \longrightarrow L_6$

Stage 0

(Q_0) $A \longrightarrow A$
(Q_1) $B \longrightarrow B$
(Q_2) $C \longrightarrow C$
(Q_3) $\bot \longrightarrow \cdot$

Stage 1

(Q_4) $A \longrightarrow L_1$	(R_{1a}, Q_0)
(Q_5) $B \longrightarrow L_2$	(R_2, Q_1)
(Q_6) $C \longrightarrow L_1$	(R_{1b}, Q_2)
(Q_7) $L_5 \longrightarrow B$	$(R_5, [Q_3, Q_1])$

Stage 2

(Q_8) $L_5 \longrightarrow L_2$	(R_2, Q_7)
(Q_9) $B, L_3 \longrightarrow C$	$(R_3, [Q_5, Q_2])$
(Q_{10}) $B, L_3 \longrightarrow L_1$	$(R_3, [Q_5, Q_6])$
(Q_{11}) $A \longrightarrow L_4$	(R_4, Q_4)
(Q_{12}) $C \longrightarrow L_4$	(R_4, Q_6)
$L_5 \longrightarrow L_2$	$(R_5, [Q_3, Q_5]) \sqsupset Q_8$

Stage 3

$B, L_3 \longrightarrow L_1$	$(R_{1b}, Q_9) \sqsupset Q_{10}$
(Q_{13}) $L_5, L_3 \longrightarrow L_4$	$(R_3, [Q_8, Q_{12}])$
(Q_{14}) $L_5, L_3 \longrightarrow L_1$	$(R_3, [Q_8, Q_6])$
(Q_{15}) $L_5, L_3 \longrightarrow C$	$(R_3, [Q_8, Q_2])$
$(Q_{16}) B, L_3 \longrightarrow L_4$	$(R_3, [Q_5, Q_{12}]) \sqsubset Q_{17}$
(Q_{17}) $B \longrightarrow L_4$	(R_4, Q_{14})
$L_5, L_3 \longrightarrow C$	$(R_5, [Q_3, Q_9]) \sqsubset Q_{15}$
$L_5, L_3 \longrightarrow L_1$	$(R_5, [Q_3, Q_{10}]) \sqsubset Q_{14}$
(Q_{18}) $A \longrightarrow L_6$	(R_6, Q_{11})
(Q_{19}) $C \longrightarrow L_6$	(R_6, Q_{12})

Stage 4

$L_5, L_3 \longrightarrow L_1$	$(R_{1b}, Q_{15}) \sqsupset Q_{14}$
(Q_{20}) $L_5, L_3 \longrightarrow L_6$	$(R_3, [Q_8, Q_{19}]) \sqsubset Q_{22}$
(Q_{21}) $L_5 \longrightarrow L_4$	(R_3, Q_{14})
$L_5 \longrightarrow L_4$	$(R_5, Q_{17}) \sqsupset Q_{21}$
(Q_{22}) $L_3 \longrightarrow L_6$	$(R_6, Q_{13}))$
(Q_{23}) $B \longrightarrow L_6$	$(R_6, Q_{17}))$

Stage 5

(Q_{24}) $L_5 \longrightarrow L_6$	$(R_5, Q_{23}))$
(Q_{25}) $\cdot \longrightarrow L_6$	$(R_6, Q_{21}))$

Figure 9: Inverse Method Example

We show a run of the algorithm in Figure 9. Each sequent in the database is numbered. The non-axioms have the rule and sequents from which they are derived in the right column. The grayed-out sequents are either forward or backward subsumed. Forward subsumed sequents are marked by $\sqsubset$ and backward by $\sqsupset$. (It is also easy to tell whether a sequent is forward or backward subsumed because a forward subsumed sequent is not give a number since it never enters the database.) A "stage" corresponds to an iteration of the while loop on line 5.

In this section we've introduced the theory of the inverse method. In the next section we describe the generic infrastructure we've used for implementing inverse method theorem provers. This infrastructure is called *Imogen*.

3 Imogen

Imogen is the name given to our family of inverse method implementations. Though the logics we will discuss in this proposal differ, proof search is roughly the same in all cases. Because search is the same, but the data structures and operations are different, we designed Imogen in two distinct components that we call *the front end* and *the back end*. Each different logic has a corresponding front end. The front end handles things like rule and sequent representation and the operations on those representations. The search itself is common to all logics, and is the responsibility of the back end. There is only one back end for all logics. The next section, Section 3.1, describes the basic workings of the generic back end search procedure. In Section 3.3 we describe the front end interface.

3.1 The Back End

The Imogen back end treats sequents and rules as abstract types. The front end supplies the concrete implementations and operations on them. These include rule matching, subsumption, and contraction among others. Since rule generation is different for each logic as well, the front end must also specify how to generate initial rules and axioms from a given formula. Once the axioms and rules are known, the back end can saturate the search space in a manner similar to that shown in Figure 9.

3.1.1 The Variable-Rule Loop

Imogen can use two different search strategies for proof search. The fixed-rule method is Algorithm 2.1. This algorithm has the benefit that the number of inference rules is fixed. However, the same sequent may be matched to the same premise of a multi-premise inference rule many times. Another way to search is to memoize multi-premise rule application by building partially applied rules. That is, Imogen fixes an order on the premises of a rule, and then matches them in order. When a match is made, a new inference rule is generated. Information from the first match is propagated to the remaining premises and conclusion. For example, consider the $\supset$-L rule

$$\frac{\cdot \longrightarrow A \quad B \longrightarrow \cdot}{A \supset B \longrightarrow \cdot}$$

If the sequent $A \longrightarrow A$ was selected for matching to this rule, it could only match the first premise and the result would be the rule

$$\frac{B \longrightarrow \cdot}{A, A \supset B \longrightarrow \cdot}$$

If the order of premises was reversed

$$\frac{B \longrightarrow \cdot \quad \cdot \longrightarrow A}{A \supset B \longrightarrow \cdot}$$

then a match with $B \longrightarrow B$ would yield rule

$$\frac{\cdot \longrightarrow A}{A \supset B \longrightarrow B}$$

The method of memoized inference rules is called the Variable-Rule Loop. Pseudocode for the variable-rule loop is in Algorithm 3.1. We maintain 4 databases: ActiveRules, KeptRules, ActiveSequents, KeptSequents. Elements of the active sets have already been considered for rule application, while those in the kept sets have not. In lines 1-4 we initialize the databases. If both of the kept databases are empty, we can not derive any additional sequents. By the soundness of the inverse method, the formula is not provable. Otherwise, on lines 6, 8, and 18 we nondeterministically select a kept rule or a kept sequent. Now we match the active sets to the kept element, generating new rules and sequents that are added to the kept databases. The process is then repeated.

Algorithm 3.1 Imogen's Variable-Rule Loop

```
 1: ActiveSequents ← ∅
 2: ActiveRules ← ∅
 3: KeptSequents ← InitialSequents
 4: KeptRules ← InitialRules
 5: while KeptSequents ≠ ∅ ∨ KeptRules ≠ ∅ do
 6:    x ← choose({ChooseRule, ChooseSequent})
 7:    if x = ChooseSequent ∧ KeptSequents ≠ ∅ then
 8:       S ← choose(KeptSequents)
 9:       ActiveSequents ← {S} ∪ ActiveSequents
10:       (NewSequents, NewRules) ← matchSeq(S, ActiveRules)
11:       if any element of NewSequents subsumes the goal then
12:          print "Proof found. The formula is true."
13:          return
14:       end if
15:       KeptRules ← KeptRules ∪ NewRules
16:       KeptSequents ← KeptSequents ∪ NewSequents
17:    else if x = ChooseRule ∧ KeptRules ≠ ∅ then
18:       R ← choose(KeptRules)
19:       ActiveRules ← {R} ∪ ActiveRules
20:       (NewSequents, NewRules) ← matchRule(R, ActiveSequents)
21:       KeptRules ← KeptRules ∪ NewRules
22:       KeptSequents ← KeptSequents ∪ NewSequents
23:    else
24:       continue
25:    end if
26: end while
27: print "The database is saturated. The formula is false."
28: return
```

Fair Selection. The variable rule loop is slightly more complicated than the fixed rule loop. The selection of rules and sequents must be *fair*: if a sequent is in the kept database, then it will eventually be considered for rule application. In opposition to this requirement is the need to give a *priority* to sequents in the database; some are more valuable to a proof than others. The measure of value is given by some heuristic, and having a good heuristic is very important to a successful theorem prover. Imogen satisfies both requirements by maintaining both a FIFO queue and a priority queue for each kept database. Usually Imogen selects the sequent (rule) with highest priority. Every so often however, it selects the sequent from the FIFO queue so as to ensure completeness. This is the "choose" operation in the algorithm.

3.2 Other Features

Redundancy Criteria. The performance of the inverse method can be improved by recognizing that some sequents it generates are redundant. Colloquially, a sequent is redundant if has no more information than an existing sequent. Concretely, this usually means that the sequent is subsumed by a sequent that is already in the database. Imogen implements both forward and backward subsumption. Backward subsumption comes in two forms. In one case, called simply backward subsumption, the backward-subsumed sequent is removed from the active sequent database. In *recursive backward subsumption*, if $Q \preceq Q'$, then not only Q' but all the descendants of Q' that are not ancestors of Q are removed. This is justified because all the sequents and rules that had been derived before will be derived again, this time with a stronger sequent to begin the process. In the variable-rule loop, it is also possible to do forward and (recursive) backward subsumption on rules, called *rule subsumption*. If a sequent in the database subsumes the conclusion of a newly generated rule, than that rule could never result in a sequent that is not subsumed as well. We can thus safely throw out the rule, in both the forward and backward cases.

Proof Terms. Each front end may provide a mechanism for constructing proof terms when proof search is successful. The back end provides the proof term with the graph of rule matches, and it is then a simple matter for the front end to reconstruct a proof term from the graph.

3.3 The Front End

Each logic must provide a front end interface to the search procedures just described. We call a front end implementation an *instance* of Imogen. We will describe the front end for a logic in the section where the logic is defined. In this section we simply point out the common features of all front ends.

Concrete Types. The front end must choose a concrete type for the abstract sequents and rules of the back end. In the case of IPL we use a set of integers that represent subformula labels for the antecedents, and a label for the succedent. In addition to concrete types, the front end must provide the principal operations of forward reasoning: subsumption, contraction, and rule matching.

Term Indexing. Since the back end is agnostic with respect to the actual form of sequents and rules, term indexing must be provided by the front end. The provided indexing data structures are used to store and retrieve rules and sequents for fast subsumption checking.

Combination. The separation of Imogen into front and back ends is helpful when different instances of Imogen are combined. A simple example is in the first order instance. If the first order front end notices that an input formula is propositional, it can use the propositional instance that is much faster due to simpler operations like matching and contraction. Likewise, if a constraint formula doesn't use the constraint domain, it can be solved by the more efficient first order prover. Looking ahead, the $\mathbf{M}_2^+$ instance will frequently use the LF instance during proof search. Thus the same back end will be used in two radically different ways during a single search.

In this section we've briefly introduced the methods by which we implement the inverse method. In the next section we describe the important optimization techniques of polarization and focusing, while describing our implementation of a theorem prover for intuitionistic propositional logic.

4 Propositional Logic

In this section we complete the exposition of the main tools of this proposal. In Section 2 we described the primary search strategy, the inverse method. In this section we discuss *focusing* and *explicit polarization* using *polarized formulas*. These are two strategies that greatly reduce the search space of the backward and forward sequent calculi. This reduction in the search space has a dramatic effect on the efficiency of proof search. We continue to use IPL as a running example.

For numerous reasons, intuitionistic formulas turn out to be too rough for our analyses. We can gain a finer control over the search space by enriching the language of formulas. In the next section we introduce polarized formulas, and show translations between standard and polarized formulas. We continue by describing a polarized backward sequent calculus, and giving soundness and completeness results with respect to the unpolarized sequent calculus. The polarized backward calculus has far better search behavior than the unpolarized calculus. In the remainder we proceed as in Section 2 to give a forward calculus suitable for the inverse method. Results for the propositional instance of Imogen are given in Section 4.7. In short, Imogen is competitive with all existing provers for IPL.

4.1 Polarization

The backward calculus defined above has an unacceptable degree of nondeterminism in its search space. Call a rule *invertible* if its premises are derivable if and only if its conclusion is derivable. One simple observation is that invertible rules can be applied in any order, without the need to consider other options. For instance, in deriving $\Gamma \Longrightarrow A \wedge B \wedge C$ we must find derivations for $\Gamma \Longrightarrow A, \Gamma \Longrightarrow B, \Gamma \Longrightarrow C$, regardless of any left rules that may be applied. A more surprising result of Andreoli [Andreoli, 1992] is that a similar result holds for non-invertible rules. To take full advantage of these observations, we will refine propositional formulas to *polarized formulas*.

Polarized Formulas. Each logical connective can be assigned a polarity, either *positive* or *negative*.[5] A connective is positive if its left rule in the sequent calculus is invertible and *negative* if its right rule is invertible. As shown below, the polarized inverse method fundamentally depends on the polarity of connectives. In intuitionistic logic, the status of conjunction and truth is ambiguous in the sense that they are both positive and negative, while their status in linear logic is uniquely determined. We syntactically distinguish positive and negative formulas with so-called *shift* operators [Lamarche, 1995] explicitly coercing between them. Since it is so convenient for our purposes, we use the notation of linear logic, even though the behavior of the connectives is not linear. Note also that both in formulas and sequents, the signs are not actual syntax but mnemonic guides to the reader.

$$\text{Positive formulas } A^+ ::= P^+ \mid A^+ \otimes A^+ \mid \mathbf{1} \mid A^+ \oplus A^+ \mid \mathbf{0} \mid \downarrow A^-$$
$$\text{Negative formulas } A^- ::= P^- \mid A^- \mathbin{\&} A^- \mid \top \mid A^+ \multimap A^- \mid \uparrow A^+$$

The translation A^- of an (unpolarized) formula F in intuitionistic logic is nondeterministic, subject only to the constraints that the *erasure* defined below coincides with the original formula: $|A^-| = F$, and all atomic formulas are assigned a consistent polarity. For example, the formula $((p \vee r) \wedge (q \supset r)) \supset (p \supset q) \supset r$ can be interpreted as any of the following polarized formulas (among others):

[5] See the footnote on nomenclature on page 13

$$
\begin{array}{rlrlrl}
|A_1^+ \oplus A_2^+| & = |A_1^+| \vee |A_2^+| & |\mathbf{0}| & = \bot & |\mathbf{1}| & = \top \\
|A_1^+ \otimes A_2^+| & = |A_1^+| \wedge |A_2^+| & |{\downarrow}A^-| & = |A^-| & |P^+| & = P \\
|A_1^- \,\&\, A_2^-| & = |A_1^-| \wedge |A_2^-| & |\top| & = \top & |P^-| & = P \\
|A_1^+ \multimap A_2^-| & = |A_1^+| \supset |A_2^-| & |{\uparrow}A^+| & = |A^+| & &
\end{array}
$$

Figure 10: IPL: Erasure of polarized formulas

$$
\begin{array}{l}
((\downarrow p^- \oplus \downarrow r^-) \otimes \downarrow(\downarrow q^- \multimap r^-)) \multimap (\downarrow(\downarrow p^- \multimap q^-) \multimap r^-) \\
\downarrow\uparrow((\downarrow p^- \oplus \downarrow r^-) \otimes \downarrow(\downarrow q^- \multimap r^-)) \multimap (\downarrow\uparrow\downarrow(\downarrow p^- \multimap q^-) \multimap r^-) \\
\downarrow(\uparrow(p^+ \oplus r^+) \,\&\, (q^+ \multimap \uparrow r^+)) \multimap (\downarrow(p^+ \multimap \uparrow q^+) \multimap \uparrow r^+)
\end{array}
$$

Shift operators have highest binding precedence in our presentation of the examples. As we will see from the inference rules given below, the choice of translation determines the search behavior on the resulting polarized formula. Different choices can lead to search spaces with radically different structure [Chaudhuri et al., 2008, McLaughlin and Pfenning, 2008].

4.2 Focusing

The backward calculus is a refinement of Gentzen's LJ that eliminates don't-care nondeterministic choices, and manages don't-know nondeterminism by chaining such inferences in sequence. Andreoli was the first to define this *focusing* strategy and prove it complete [Andreoli, 1992], and similar proofs for other logics soon followed [Howe, 1998, Liang and Miller, 2007, Zeilberger, 2008, Baelde, 2008]. Polarization can be applied to optimize search in a wide variety of logics.

The polarized calculus is defined via four mutually recursive judgments. In the judgments, we separate the antecedents into positive and negative zones. We write Γ for an unordered collection of negative formulas or positive atoms. Dually, C stands for a positive formula or a negative atom. The first two judgments concern formulas with invertible rules on the right and left. Together, the two judgments form the *inversion phase* of focusing. The context Δ^+ is consists entirely of positive formulas and is ordered so that inference rules can only be applied to the rightmost formula, eliminating don't-care nondeterminism. The next two judgments are concerned with non-invertible rules. These two judgments make up the *focusing phase*.

Backward search for a proof of A^- would start with inversion from $\cdot; \cdot \Uparrow A^-$ and then alternate between focusing and inversion phases. Call a focusing phase followed by an inversion phase a *block* (also called a *dipole* in the literature). The boundary between blocks is of particular importance. The sequents at the boundary have the form $\Gamma; \cdot \Uparrow C$, which we call *stable sequents*. There are two rules that control the phase changes and make the choices at block boundaries. The full calculus is shown in Figure 11. Note that the positive formulas Δ of the inversion phases are ordered, thus making those phases deterministic.

Soundness is straightforward for the polarized backward calculus, as we can erase the formulas and contract shift steps. This will directly yield a backward proof. (Recall that C stands either for a negative formula or a positive atom.) Completeness is non-trivial, and the reader is referred to the literature.

Theorem 4.1 (Soundness). *If* $\Gamma; \Delta \Uparrow C$ *then* $|\Gamma, \Delta| \Longrightarrow |C|$.

Right Inversion $\boxed{\Gamma; \Delta^+ \Uparrow A^-}$

$$\frac{\Gamma; \Delta^+ \Uparrow P^-}{\Gamma; \Delta^+ \Uparrow P^-}\ \text{ß(RA-Atom)} \qquad \frac{\Gamma; \Delta^+ \Uparrow A_1^- \quad \Gamma; \Delta^+ \Uparrow A_2^-}{\Gamma; \Delta^+ \Uparrow A_1^- \& A_2^-}\ \text{ß(RA-\&)}$$

$$\frac{\Gamma; \Delta^+, A_1^+ \Uparrow A_2^-}{\Gamma; \Delta^+ \Uparrow A_1^+ \multimap A_2^-}\ \text{ß(RA-}\multimap\text{)} \qquad \frac{}{\Gamma; \Delta^+ \Uparrow \top}\ \text{ß(RA-}\top\text{)} \qquad \frac{\Gamma; \Delta^+ \Uparrow A^+}{\Gamma; \Delta^+ \Uparrow \uparrow A^+}\ \text{ß(RA-}\uparrow\text{)}$$

Left Inversion $\boxed{\Gamma; \Delta^+ \Uparrow C}$

$$\frac{\Gamma, P^+; \Delta^+ \Uparrow C}{\Gamma; \Delta^+, P^+ \Uparrow C}\ \text{ß(LA-Atom)} \qquad \frac{\Gamma; \Delta^+, A_1^+, A_2^+ \Uparrow C}{\Gamma; \Delta^+, A_1^+ \otimes A_2^+ \Uparrow C}\ \text{ß(LA-}\otimes\text{)} \qquad \frac{\Gamma; \Delta^+ \Uparrow C}{\Gamma; \Delta^+, \mathbf{1} \Uparrow C}\ \text{ß(LA-}\mathbf{1}\text{)}$$

$$\frac{\Gamma; \Delta^+, A_1^+ \Uparrow C \quad \Gamma; \Delta^+, A_2^+ \Uparrow C}{\Gamma; \Delta^+, A_1^+ \oplus A_2^+ \Uparrow C}\ \text{ß(LA-}\oplus\text{)} \qquad \frac{}{\Gamma; \Delta^+, \mathbf{0} \Uparrow C}\ \text{ß(LA-}\mathbf{0}\text{)} \qquad \frac{\Gamma, A^-; \Delta^+ \Uparrow C}{\Gamma; \Delta^+, \downarrow A^- \Uparrow C}\ \text{ß(LA-}\downarrow\text{)}$$

Right Focusing $\boxed{\Gamma \Downarrow [A^+]}$

$$\frac{}{\Gamma, P^+ \Downarrow [P^+]}\ \text{ß(RS-Atom)} \qquad \frac{\Gamma \Downarrow [A_1^+] \quad \Gamma \Downarrow [A_2^+]}{\Gamma \Downarrow [A_1^+ \otimes A_2^+]}\ \text{ß(RS-}\otimes\text{)} \qquad \frac{}{\Gamma \Downarrow [\mathbf{1}]}\ \text{ß(RS-}\mathbf{1}\text{)}$$

$$\frac{\Gamma \Downarrow [A_i^+]}{\Gamma \Downarrow [A_1^+ \oplus A_2^+]}\ \text{ß(RS-}\oplus_i\text{)} \qquad \text{No rule for } \mathbf{0} \qquad \frac{\Gamma; \cdot \Uparrow A^-}{\Gamma \Downarrow [\downarrow A^-]}\ \text{ß(RS-}\downarrow\text{)}$$

Left Focusing $\boxed{\Gamma; [A^-] \Downarrow C}$

$$\frac{}{\Gamma; [P^-] \Downarrow P^-}\ \text{ß(LS-Atom)} \qquad \frac{\Gamma; [A_1^-] \Downarrow C}{\Gamma; [A_1^- \& A_2^-] \Downarrow C}\ \text{ß(LS-}\&_1\text{)} \qquad \frac{\Gamma; [A_2^-] \Downarrow C}{\Gamma; [A_1^- \& A_2^-] \Downarrow C}\ \text{ß(LS-}\&_2\text{)}$$

$$\frac{\Gamma \Downarrow [A_1^+] \quad \Gamma; [A_2^-] \Downarrow C}{\Gamma; [A_1^+ \multimap A_2^-] \Downarrow C}\ \text{ß(LS-}\multimap\text{)} \qquad \text{No rule for } \top \qquad \frac{\Gamma; A^+ \Uparrow C}{\Gamma; [\uparrow A^+] \Downarrow C}\ \text{ß(LS-}\uparrow\text{)}$$

Stable Rules

$$\frac{\Gamma \Downarrow [A^+]}{\Gamma; \cdot \Uparrow A^+}\ \text{ß(FocusR)} \qquad \frac{\Gamma, A^-; [A^-] \Downarrow C}{\Gamma, A^-; \cdot \Uparrow C}\ \text{ß(FocusL)}$$

Figure 11: The Backward Focused Calculus

Proof. By induction on the derivation. □

Theorem 4.2 (Completeness)**.** *If* $\cdot \Longrightarrow |C|$ *then* $\cdot;\cdot \Uparrow C$.

Proof. The proof is a special case of [Liang and Miller, 2007]. □

4.3 Synthetic Connectives and Derived Rules

We have already observed that backward proofs have the property that the proof is broken into blocks, with stable sequents at the boundary. The only rules applicable to stable sequents are the rules that select a formula on which to focus. It is the formulas occurring in stable sequents that form the primary objects of our further inquiry.

It helps to think of such formulas, abstracted over their free variables, as *synthetic connectives* [Andreoli, 2001]. Define the synthetic connectives of a formula A as all subformulas of A that could appear in stable sequents in a focused backward proof. In a change of perspective, we can consider each block of a proof as the application of a left or right rule for a synthetic connective. The rules operating on synthetic connectives are derived from the rules for its constituent formulas. We can thus consider a backward proof as a proof using only these synthetic (derived) rules. Each derived rule then corresponds to a block of the original proof.

Since we need only consider stable sequents and synthetic connectives, we can simplify notation, and ignore the (empty) positive left and negative right zones in the derived rules. Write $\Gamma;\cdot \Longrightarrow \cdot;C$ as $\Gamma \Longrightarrow C$. As a further simplification, we can give formulas a predicate label and abstract over its free variables. This labeling technique is described in detail in [Degtyarev and Voronkov, 2001b]. For the remainder, we assume this labeling has been carried out. Define an *atomic formula* as either a label or a predicate applied to a (possibly empty) list of terms. After labeling, our sequents consist entirely of atomic formulas.

Example 4.1. In Figure 12, frame boxes surround the three blocks of the proof. The synthetic connectives are a, $a \supset b$ and $(a \supset b) \supset c$. There is a single derived rule for each synthetic connective (though this is not the case in general). The atoms are assigned negative polarity. We implicitly carry the principal formula of a left rule to all of its premises.

$$\frac{}{\Gamma, a \Longrightarrow a}\ \mathrm{Syn}_1 \qquad \frac{\Gamma \Longrightarrow a}{\Gamma, a \supset b \Longrightarrow b}\ \mathrm{Syn}_2 \qquad \frac{\Gamma, a \Longrightarrow b}{\Gamma, (a \supset b) \supset c \Longrightarrow c}\ \mathrm{Syn}_3$$

These rules correspond to the blocks shown in Figure 12. Corresponding labeled rules for $L_1 = A \supset B$ and $L_2 = (A \supset B) \supset C$ are

$$\frac{}{\Gamma, a \Longrightarrow a}\ \mathrm{Syn}_1 \qquad \frac{\Gamma \Longrightarrow a}{\Gamma, L_1 \Longrightarrow b}\ \mathrm{Syn}_2 \qquad \frac{\Gamma, a \Longrightarrow b}{\Gamma, L_2 \Longrightarrow c}\ \mathrm{Syn}_3$$

Then the blocks of the proof from Figure 12 can be compressed to the succinct

$$\dfrac{\dfrac{\dfrac{}{L_1, L_2, a \Longrightarrow a}\ \mathrm{Syn}_1}{L_1, L_2, a \Longrightarrow b}\ \mathrm{Syn}_2}{L_1, L_2 \Longrightarrow c}\ \mathrm{Syn}_3$$

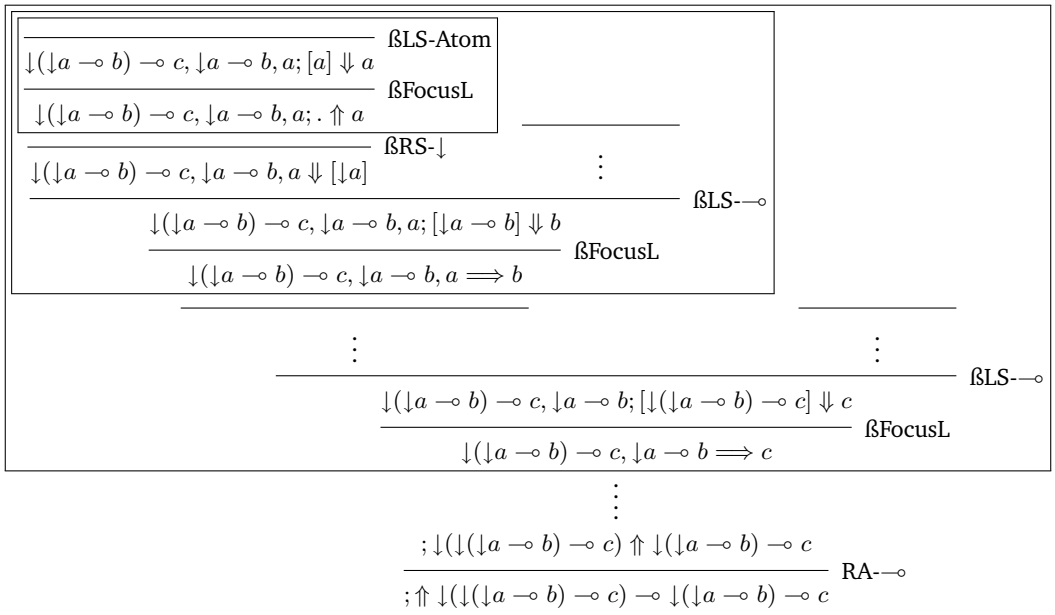

Figure 12: Backward proof, with blocks

4.4 The Polarized Inverse Method

We now wish to turn the above observations into an efficient proof search procedure using the inverse method. The combination of the inverse method combined with the optimizations of polarization and focusing we call the *polarized inverse method*. In the remainder of this section we show how to implement the polarized inverse method for IPL. The implementation will consist of providing a front end for Imogen. Since we've already discussed the sequent representation and subsumption, and contraction is trivial, we need only show how to match rules and sequents.

4.4.1 Matching

In the backward direction in IPL, matching is a simple matter, consisting of decomposing the principal connective and copying the context. The presence of possibly empty succedents and the union of contexts complicates the operation in the forward direction. Additionally, focusing can make rules of arbitrary size, making the rules relatively complicated objects. Since formally defining matching obscures this rather simple process, we first give an example of matching a rule with an empty succedent to motivate the definition. Consider the rule obtained by focusing on $\uparrow(A \oplus B)$ on the left.

$$\frac{A \longrightarrow \cdot \quad B \longrightarrow \cdot}{\uparrow(A \oplus B) \longrightarrow \cdot}$$

We can match this rule one premise at a time. If we match the leftmost premise with the initial sequent $A \longrightarrow A$, then the resulting rule will be

$$\frac{B \longrightarrow A}{\uparrow(A \oplus B) \longrightarrow A}$$

Notice that the concrete succedent A instantiated the $\cdot$ in the rule. Now the succedent of a sequent that matches the second must match A as a succedent. However, if the matching sequent were $A, \neg A \longrightarrow \cdot$, then the resulting rule would be

$$\frac{B \longrightarrow \cdot}{\uparrow(A \oplus B), \neg A \longrightarrow \cdot}$$

and the succedents of the rule would not be instantiated. Note that the unmatched antecedent $\neg A$ was added to the antecedents of the conclusion of the rule. As mentioned in Section 2.5, we can avoid using three different rules for $\supset$-R. We require neither the antecedents nor the succedent to match exactly. Rule matching is then defined by cases. In the simplest case, there are no empty succedents.

Definition 4.1 (Rule Matching 1)**.** Sequents $\Delta_1 \longrightarrow \delta_1, \ldots, \Delta_n \longrightarrow \delta_n$ match rule

$$\frac{\Gamma_1 \longrightarrow A_1 \quad \cdots \quad \Gamma_n \longrightarrow A_n}{\Gamma \longrightarrow A}$$

if for all $1 \leq i \leq n$, either $\delta_i = A_i$ or $\delta_i = \cdot$. In that case, the resulting sequent is

$$\Gamma \cup (\Delta_1 \setminus \Gamma_1) \cup \ldots \cup (\Delta_n \setminus \Gamma_n) \longrightarrow A$$

If there is a premise with an empty succedents in the rule, then the conclusion also has an empty succedent. This can be seen by a routine investigation of the focusing rules. In this case, we can rearrange the premises so that the first k premises have an empty antecedent. Then we can use the following definition of matching.

Definition 4.2 (Rule Matching 2)**.** Sequents $\Delta_1 \longrightarrow \delta_1, \ldots, \Delta_n \longrightarrow \delta_n$ match rule

$$\frac{\Gamma_1 \longrightarrow \cdot \quad \cdots \quad \Gamma_k \longrightarrow \cdot \quad \Gamma_{k+1} \longrightarrow A_{k+1} \quad \cdots \quad \Gamma_n \longrightarrow A_n}{\Gamma \longrightarrow \cdot}$$

if one of the following conditions holds.

1. - For all $1 \leq i \leq k$, $\delta_i = \cdot$.
   - For all $k+1 \leq i \leq n$, $\delta_i = \cdot$ or $\delta_i = A_i$.

   In this case the resulting sequent is

   $$\Gamma \cup (\Delta_1 \setminus \Gamma_1) \cup \ldots \cup (\Delta_n \setminus \Gamma_n) \longrightarrow \cdot$$

2. - There exists $1 \leq i \leq k$, $\delta_i = A$.
   - For all $1 \leq i \leq k$, $\delta_i = \cdot$ or $\delta_i = A$.
   - For all $k+1 \leq i \leq n$, $\delta_i = \cdot$ or $\delta_i = A_i$.

   In this case the resulting sequent is

   $$\Gamma \cup (\Delta_1 \setminus \Gamma_1) \cup \ldots \cup (\Delta_n \setminus \Gamma_n) \longrightarrow A$$

4.4.2 Search

Using the inverse method we can now search for polarized proofs in the forward direction. We suppose the input formula ξ has negative polarity. (If ξ is positive, simply prepend an up arrow.) First we *stabilize* ξ. This means we decompose ξ using the asynchronous rules yielding a set of stable sequents. This step corresponds to the lower part of Figure 12 that is not in a block. We search for proofs of each stable sequent separately. Given a stable sequent q, we generate the corresponding derived inference rules and initial sequents by focusing on the synthetic connectives of q. This process creates all possible blocks that could arise in a backward focused proof. Then we start a saturating search using one of the Imogen loops. If we eventually generate a sequent that subsumes the goal sequent q, then we have found a proof. If the database saturates, we stop searching, as no proof exists.

Theorem 4.3 (Soundness)**.** *The above algorithm is sound. That is, if it reports a proof has been found for every q, then $\cdot \longrightarrow \xi$. If for any q the algorithm reports that no proof exists, then then there is no derivation $\cdot \longrightarrow \xi$.*

Proof. If a proof is found, then we can construct the derivation using the derived inference rules, which can in turn be expanded into rules of the original forward calculus for IPL. If the database saturates for one of the q, by the completeness of the focused system no proof can exist. □

Theorem 4.4 (Completeness)**.** *If $\cdot \longrightarrow \xi$, then the algorithm will eventually report that a proof exists.*

Proof. By the correctness of the Imogen loops and the completeness of the focused calculus. □

4.5 Example

As an example, we consider the example formula from Section 2.

$$\xi = (\bot \vee B) \supset ((A \supset B) \supset C) \supset (A \vee C)$$

First we choose a polarization ξ_p for ξ such that $|\xi_p| = \xi$. We arbitrarily decide that A, B, C will be positive, and we will use a minimal number of shifts. This yields

$$\xi_p = (\mathbf{0} \oplus B) \multimap \downarrow(\downarrow(A \multimap \uparrow B) \multimap \uparrow C) \multimap \uparrow(A \oplus C)$$

The initial inversion phase yields the single sequent

$$B, \downarrow(A \multimap \uparrow B) \multimap \uparrow C \Longrightarrow A \oplus C$$

First we label subformulas:

$$L_1 = A \oplus C \quad L_2 = \uparrow B \quad L_3 = \downarrow(A \multimap L_2) \multimap \uparrow C$$

Now we generate the initial rules and sequents. Focusing on L_1 on the right yields the two axioms

$$A \longrightarrow L_1 \qquad C \longrightarrow L_1$$

Focusing on L_3 on the left yields the derived rule

$$\frac{C \longrightarrow \gamma \quad A \longrightarrow L_2}{L_3 \longrightarrow \gamma}\ R_1$$

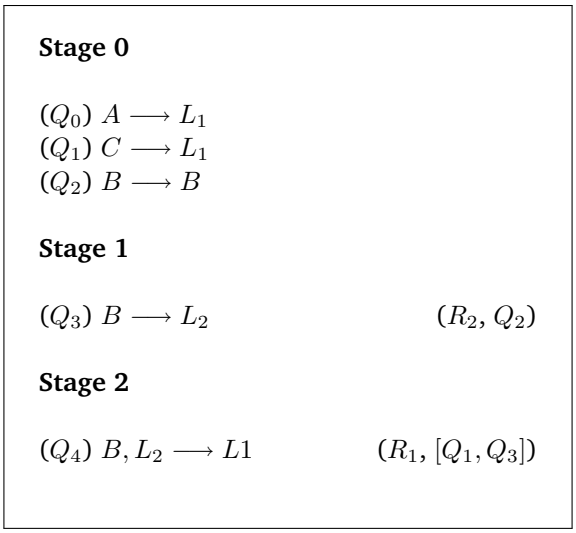

Figure 13: Polarized Inverse Method Example

Focusing in turn on the occurrence of L_2 in R_1 yields the rule

$$\frac{\cdot \longrightarrow B}{\cdot \longrightarrow L_2}\ R_2$$

Finally we focus on B, occurring on the right in R_2 giving the axiom $B \longrightarrow B$. The goal is $B, L_3 \longrightarrow L_2$. The trace of the polarized inverse method is shown in Figure 13. Compare to Figure 9.

4.6 Heuristics

We will see in later sections that the ability to assign polarities to atomic formulas will greatly influence the search behavior. In this section we show that the ability to add shifts also can drastically improve performance. Since we don't have a general theory about the optimal choice of atom polarities or the insertion of shifts, we use heuristics.

Notice that in some cases focusing will lead to undesirable behavior. For instance, formulas of the form

$$(A_1 \oplus B_1) \otimes (A_2 \oplus B_2) \ldots \otimes (A_n \oplus B_n)$$

will generate an exponential number of inference rules and

$$(A_1 \oplus B_1) \multimap (A_2 \oplus B_2) \ldots \multimap (A_n \oplus B_n) \multimap C$$

will produce a single rule with an exponential number of premises. To counteract such deleterious effects, one can insert double shifting operators to break up the focusing and inversion phases, leading to a smaller number of rules at the cost of a larger search space. For instance,

$$\downarrow\uparrow(A_1 \oplus B_1) \otimes \downarrow\uparrow(A_2 \oplus B_2) \ldots \otimes \downarrow\uparrow(A_n \oplus B_n)$$

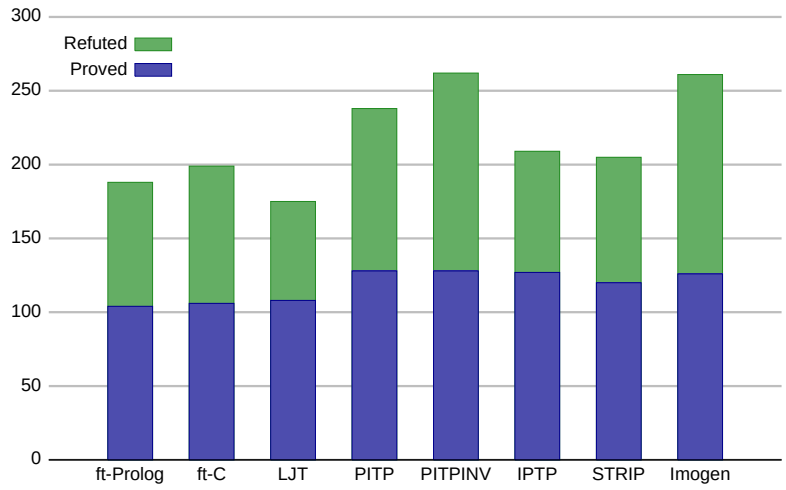

Figure 14: ILTP Propositional Benchmark

generates a linear number of rules. By making such observations on a target set of formulas, we designed heuristics to prevent this kind of formula from ruining the chances that the proving process will never even begin because the front end spends an exponential amount of time creating the inference rules.

4.7 Implementation

The propositional instance of Imogen gives promising results. We evaluated our prover on the propositional fragment of the Intuitionistic Logic Theorem Proving (ILTP) [Raths et al., 2007, version 1.1.2] library of problems for intuitionistic theorem provers. The 274 problems are divided into 12 families of difficult problems such as the pigeonhole principle, labeled SYJ201 to SYJ212. For each family, there are 20 instances of increasing size. There are also 34 miscellaneous problems. The provers that are currently evaluated are ft-C [Sahlin et al., 1992, version 1.23], ft-Prolog [Sahlin et al., 1992, version 1.23], LJT [Dyckhoff, 1992], PITP [Avellone et al., 2004, version 3.0], PITPINV [Avellone et al., 2004, version 3.0], and STRIP [Larchey-Wendling et al., 2001, version 1.1]. These provers represent a number of different methods of theorem proving in IPL, yet forward reasoning is conspicuously absent. Imogen solved 261 of the problems. PITPINV [Avellone et al., 2004] was the only prover to solve more, at 262. A summary results are shown in Figure 14, and a table with some representative times is shown in Table 1.

The table uses the notation of [Raths and Otten,]. All times are in seconds. The entry "memory" indicates that the prover process ran out of memory. A "time" entry indicates that the prover was unable to solve the problem within the ten minute time limit. A negative number indicates the time to ascertain that a formula is *not* valid. All statistics except for those of Imogen were executed on a 3.4 GHz Xeon processor running Linux [Raths and Otten,]. The Imogen statistics are a 2.4 GHz Intel Core 2 Duo on Mac OS X. Thus the Imogen statistics are conservative.

Analysis. There are a few things to be learned from Imogen's performance on this database. First, the polarized inverse method can be competitive with tableaux and connection style theorem provers. Second, by manipulating the polarities of conjunction and the atomic formulas with heuristics, we can dramatically

Prover	ft-Prolog	ft-C	LJT	PITP	PITPINV	IPTP	STRIP	Imogen
Solved (out of 274)	188	199	175	238	262	209	205	261
SYN007+1.014	-0.01	-0.01	stack	large	large	large	alloc	-0.1
SYJ201+1.018	0.28	0.04	0.4	0.01	0.01	2.31	0.23	25.5
SYJ201+1.019	0.36	0.04	0.47	0.01	0.01	2.82	0.32	28.0
SYJ201+1.020	0.37	0.05	0.55	0.01	0.01	3.47	0.34	28.35
SYJ202+1.007	516.55	76.3	memory	0.34	0.31	13.38	268.59	64.6
SYJ202+1.008	time	time	memory	3.85	3.47	97.33	time	time
SYJ202+1.009	time	time	memory	50.25	42.68	time	time	time
SYJ202+1.010	time	time	memory	time	time	time	time	time
SYJ205+1.018	time	time	0.01	0.01	7.49	0.09	time	0.01
SYJ205+1.019	time	time	0.01	0.01	15.89	0.09	time	0.01
SYJ205+1.020	time	time	0.01	0.01	33.45	0.1	time	0.01
SYJ206+1.018	time	time	memory	1.01	0.96	9.01	8.18	56.2
SYJ206+1.019	time	time	memory	1.95	1.93	18.22	14.58	394.14
SYJ206+1.020	time	time	memory	3.92	3.89	36.35	33.24	42.7
SYJ207+1.018	time	time	time	time	-68.71	time	time	-42.6
SYJ207+1.019	time	time	time	time	-145.85	time	time	-63.6
SYJ207+1.020	time	time	time	time	-305.21	time	time	-97.25
SYJ208+1.018	time	time	memory	-0.99	-0.95	time	time	-184.14
SYJ208+1.019	time	time	memory	-1.36	-1.35	memory	mem	-314.31
SYJ208+1.020	time	time	memory	-1.76	-1.80	memory	mem	-506.02
SYJ209+1.018	time	time	time	time	-13.44	time	time	-0.01
SYJ209+1.019	time	time	time	time	-28.68	time	time	-0.01
SYJ209+1.020	time	time	time	time	-60.54	time	time	-0.02
SYJ211+1.018	time	time	time	-43.65	-31.51	time	time	-0.02
SYJ211+1.019	time	time	time	-91.75	-66.58	time	time	-0.02
SYJ211+1.020	time	time	time	-191.57	-139.67	time	time	-0.02
SYJ212+1.018	-0.01	-0.01	memory	-1.31	-1.37	time	-8.5	-0.02
SYJ212+1.019	-0.01	-0.01	memory	-2.7	-2.75	time	-17.41	-0.03
SYJ212+1.020	-0.01	-0.01	memory	-5.51	-5.51	time	-38.94	-0.04

Table 1: Imogen Propositional Statistics

affect Imogen's performance. Third, Imogen is good at *disproving* formulas quickly. This is evident in problems SYJ209-212. In contrast to the backward systems, Imogen can immediately see that the problem is unsolvable. WE will see that this will be a very important property for a Twelf meta-theorem prover.

5 First-Order Logic

Now that we've demonstrated the general mechanism of focusing with polarities and the polarized inverse method, extending the prover to first-order logic is straightforward. First-order logic is similar to propositional logic except that we need to reconsider the primary operations in the presence of variables and first-order unification. For instance, with first-order terms, contraction becomes non-trivial. In this section we define those operations and extend the focusing judgments from the last section to first order quantification. We assume the reader is familiar with the usual language of substitutions and first-order unification. A thorough introduction to the area, which also contains implementation details, is Nipkow and Baader [Baader and Nipkow, 1998].

5.1 Lifting

First order formulas extend propositional formulas with terms and quantifiers.

$$\begin{aligned} \text{Terms } T &::= x \mid f(T_1, \ldots, T_n) \\ \text{Formulas } A &::= P(T_1, \ldots, T_n) \mid \ldots \mid \forall x.\ A \mid \exists x.\ A \end{aligned}$$

Following [Degtyarev and Voronkov, 2001b], to build a forward calculus suitable for proof search one would first give backward rules for the quantifiers, assuming all terms are *ground*, i.e. containing no variables. Then to make the calculus suitable for proof search, one would add free variables and unification to the calculus, leveraging the *free signed subformula property* to satisfy the finite rule property. This is all standard, and the notation is baroque due to ubiquitous substitutions. We therefore elide the details of this process, and refer the interested reader to the Handbook articles on the inverse method and resolution, where the technique sketched above, called *lifting*, is widely employed. In this section, we extend the focused calculus to include the quantifiers, and discuss the extensions of the important operations of the inverse method.

5.2 First-Order Focusing

Extending polarization to first order formulas is straightforward. The polarities are

$$\begin{aligned} \text{Positive formulas } A^+ &::= P^+(T_1, \ldots, T_n) \mid \ldots \mid \exists x.\ A^+ \\ \text{Negative formulas } A^- &::= P^-(T_1, \ldots, T_n) \mid \ldots \mid \forall x.\ A^- \end{aligned}$$

with erasure

$$|\forall x.\ A^-| = \forall x.\ |A^-| \qquad |\exists x.\ A^+| = \exists x.\ |A^+|$$

and subformulas

$$\begin{aligned} A(a)^r &\leq (\forall x.A(x))^r & A(t)^l &\leq (\forall x.A(x))^l \\ A(t)^r &\leq (\exists x.A(x))^r & A(a)^r &\leq (\exists x.A(x))^l \end{aligned}$$

where a is a meta-variable ranging over parameters (discussed below) and t ranges over arbitrary terms. The focused backward rules are the same as for the propositional fragment. Quantifier rules are shown in Figure 15.

Right Inversion

$$\frac{\Gamma; \Delta^+ \Uparrow A(a)^-}{\Gamma; \Delta^+ \Uparrow \forall x.\ A(x)^-}\ \text{ß(RA-}\forall\text{)}^a$$

Left Inversion

$$\frac{\Gamma; \Delta^+, A(a)^+ \Uparrow C}{\Gamma; \Delta^+, \exists x.\ A(x)^+ \Uparrow C}\ \text{ß(LA-}\exists\text{)}^a$$

Right Focusing

$$\frac{\Gamma \Downarrow \left[A(t)^+\right]}{\Gamma \Downarrow \left[\exists x.\ A^+\right]}\ \text{ß(RS-}\exists\text{)}$$

Left Focusing

$$\frac{\Gamma; \left[A(t)^-\right] \Downarrow C}{\Gamma; \left[\forall x.\ A^-\right] \Downarrow C}\ \text{ß(LS-}\forall\text{)}$$

Figure 15: The Backward Focused Calculus (Quantifiers)

Parameters. Note that there are two common ways to handle the meta-variable a in the rules $\forall$-R and $\exists$-L. In the most common presentation, a is a new variable, called an *eigenvariable*, that does not occur free in $\Gamma \cup \Delta \cup \{A, C\}$. In other presentations, one can distinguish a syntactically different class of variables, called *parameters*. Then only parameters are introduced through these two rules. Parameters affects unification because a parameter may be instantiated with another parameter, but not an arbitrary term. Conversely, variables can be instantiated with any term, including parameters. In the remainder, we use the later convention. A formal treatment of parameters is found in Chaudhuri [Chaudhuri, 2006]. If a derived rule introduces parameters during the inversion phase, we indicate this by maintaining the set of introduced parameters along with the rule. For instance, focusing on $L = \forall x.\ \forall y.\ P(x, y)$ on the right will result in the derived rule

$$\frac{\longrightarrow P(a, b)}{\longrightarrow L}\ R^{\{a,b\}}$$

The matching algorithm will make sure the eigenvariable restrictions hold.

5.3 Contraction, Subsumption, and Matching

With the generic backward inference rules in hand, we can easily generate the specialized initial rules and sequents for the polarized inverse method as we did in the propositional case. Saturation then proceeds by matching known sequents to synthetic inference rules, and contraction. Before we begin defining the operations we give a convention

Definition 5.1 (Renaming convention)**.** When comparing different sequents, for example in subsumption, and matching rules premises to known sequents, we implicitly rename the sequents apart. Because variables of sequents and rules are interpreted as being implicitly quantified outside the sequent, this causes no trouble, and the convention obviates the pesky renaming substitutions that would otherwise need to be

applied at every turn. This convention is analogous to Barendregt's convention for α-equivalent renamings [Barendregt, 1984].

5.3.1 Contraction

An essential rule in forward chaining is the *contraction* rule:

$$\frac{\Gamma, A, A \longrightarrow \gamma}{\Gamma, A \longrightarrow \gamma}\ \text{Contract}$$

In propositional logic we didn't need to implement this rule because we represented antecedents as a set. In first-order logic, we must unify members of the antecedents with the same predicate label.

$$\frac{\Gamma, A_1, A_2 \longrightarrow \gamma \quad A_1\theta = A_2\theta}{\Gamma\theta, A_1\theta \longrightarrow \gamma\theta}\ \text{Contract}$$

In practice, this poses a problem for an implementation. The sequent number of contraction instance of $A(x_1), \ldots, A(x_n) \longrightarrow C$ is exponential in n. The Imogen back end generates all contraction instances eagerly, and unfortunately this exponential behavior can be observed in practice.

5.3.2 Subsumption

Subsumption must also take unification into account. Since a sequent stands for all of its substitution instances, a sequent Q can subsume another Q' only if the instances of Q are a superset of the instances of Q'. This is captured by the following definition.

Definition 5.2. $\Gamma \longrightarrow C$ *subsumes* $\Gamma' \longrightarrow C'$ if there exists a substitution σ such that $\Gamma\sigma \subseteq \Gamma'$ and $C\sigma \subseteq C'$.

5.3.3 Matching

Rule matching is again complicated by unification and parameters.[6] First note that we must collect the parameters introduced by the rule to ensure the eigenvariable condition. Consider the rule

$$\frac{\longrightarrow p(x, a)}{\longrightarrow \exists x.\ \forall y.\ p(x, y)}\ R^{\{a\}}$$

We must ensure when matching rule R to sequent $\Delta \longrightarrow \delta$ that parameter a does not occur in Δ. Moreover, x must not be unified with any term containing a. We will again define matching by cases, c.f. Section 4.4.1, taking these complications into account. Let $\texttt{vars}(t)$ denote the free variables of term t.

Definition 5.3 (Rule Matching 1)**.** Sequents $\Delta_1 \longrightarrow \delta_1, \ldots, \Delta_n \longrightarrow \delta_n$ match rule

$$\frac{\Gamma_1 \longrightarrow A_1 \quad \cdots \quad \Gamma_n \longrightarrow A_n}{\Gamma \longrightarrow A}\ R^{\Pi}$$

with substitution θ if the following conditions hold for all $1 \leq i \leq n$.

1. Either $\delta_i\theta = A_i\theta$ or $\delta_i = \cdot$.

[6] Recall [Chaudhuri, 2006] throughout that if a parameter a is in the domain of a substitution θ, then $a\theta$ is also a parameter.

2. The parameters $\Pi\theta$ do not occur in $\Delta_i\theta \setminus \Gamma_i\theta$.
3. The parameters $\Pi\theta$ do not occur in $\mathtt{vars}(\Gamma_i, A_i)\theta$
4. For any two parameters $a, b \in \Pi$, $a\theta \neq b\theta$

In that case, the resulting sequent is

$$\Gamma\theta \cup (\Delta_1\theta \setminus \Gamma_1\theta) \cup \ldots \cup (\Delta_n\theta \setminus \Gamma_n\theta) \longrightarrow A\theta$$

As in the propositional case, if there is a premise with an empty succedents in the rule, then the conclusion also has an empty succedent. This can be seen by a routine investigation of the focusing rules. In this case, we can rearrange the premises so that the first k premises have an empty antecedent. Then we can use the following definition of matching.

Definition 5.4 (Rule Matching 2)**.** Sequents $\Delta_1 \longrightarrow \delta_1, \ldots, \Delta_n \longrightarrow \delta_n$ match rule

$$\frac{\Gamma_1 \longrightarrow \cdot \quad \cdots \quad \Gamma_k \longrightarrow \cdot \quad \Gamma_{k+1} \longrightarrow A_{k+1} \quad \cdots \quad \Gamma_n \longrightarrow A_n}{\Gamma \longrightarrow \cdot}\ R^{\Pi}$$

if there exists a substitution θ such that

1. The parameters $\Pi\theta$ do not occur in $\Delta_i\theta \setminus \Gamma_i\theta$.
2. The parameters $\Pi\theta$ do not occur in $\mathtt{vars}(\Gamma_i, A_i)\theta$
3. For any two parameters $a, b \in \Pi$, $a\theta \neq b\theta$

and one of the following conditions holds:

1. - For all $1 \leq i \leq k$, $\delta_i = \cdot$.
   - For all $k+1 \leq i \leq n$, $\delta_i = \cdot$ or $\delta_i\theta = A_i\theta$.

   In this case the resulting sequent is

   $$\Gamma\theta \cup (\Delta_1\theta \setminus \Gamma_1\theta) \cup \ldots \cup (\Delta_n\theta \setminus \Gamma_n\theta) \longrightarrow \cdot$$

2. - There exists $1 \leq i \leq k$, $\delta_i\theta = A\theta$.
   - For all $1 \leq i \leq k$, $\delta_i = \cdot$ or $\delta_i\theta = A\theta$.
   - For all $k+1 \leq i \leq n$, $\delta_i = \cdot$ or $\delta_i\theta = A_i\theta$.

   In this case the resulting sequent is

   $$\Gamma\theta \cup (\Delta_1\theta \setminus \Gamma_1\theta) \cup \ldots \cup (\Delta_n\theta \setminus \Gamma_n\theta) \longrightarrow A\theta$$

Example 5.1. If the synthetic connective is $L_1 = \downarrow(((\exists y.\ \downarrow p(y)) \multimap \forall x.\ (p(x)\ \&\ q(x)))$ on the right (atoms are negative), then the backward and forward synthetic rules are

$$\frac{\Gamma, p(a) \Longrightarrow p(b) \quad \Gamma, p(a) \Longrightarrow q(b)}{\Gamma \Longrightarrow L_1}\ R^{\{a,b\}}$$

$$\frac{p(a) \longrightarrow p(b) \quad p(a) \longrightarrow q(b)}{\longrightarrow L_1}\ R^{\{a,b\}}$$

5.4 Implementation

The first-order instance of Imogen was implemented using the focusing strategy and operations defined above. We ran Imogen on the ILTP library of first order problems [Raths and Otten,]. The results are promising. Imogen solved far more problems than any other intuitionistic first order prover. Again, it also was comparatively successful at *disproving* theorems (i.e. the database saturated).

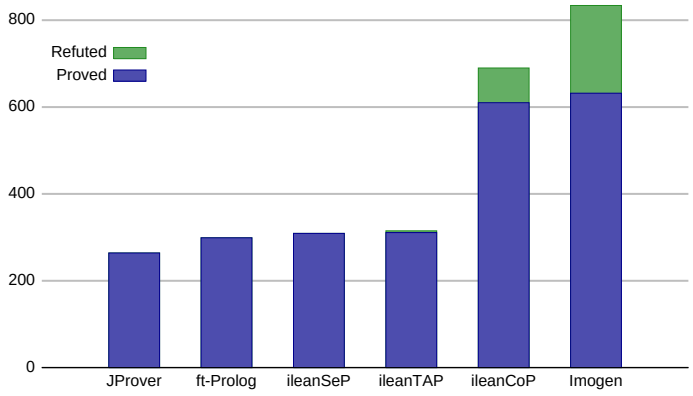

Figure 16: ILTP Results

Polarization. To measure the effects of focusing and polarization, we pursued a number of experiments. With the method of explicit polarities, it is easy to simulate partially or unfocused calculi by inserting double-shifts between connectives that break the current phase and generate a block boundary. Figure 17 shows the results of these simulations. *Single Step* simulates the unfocused inverse method. *Weak Focusing* makes all focusing phases complete, but breaks the inversion phases into single steps. *Weak Inversion* makes the inversion phase complete, but breaks the focusing phase into single steps. *Fair Weak Focusing* is like weak focusing but allows the initial stabilization phase to run unchecked. In all of these experiments, we assigned negative polarity to all atoms. *Positive Atoms* makes all atoms positive, but otherwise does no unnecessary shifting.

Subsumption. Another experiment involved the benefits of backward subsumption. Figure 17 shows the performance of Imogen with different backward subsumption settings. The definitions of recursive and rule subsumption are given in Section 3.2.

5.5 Improvements

There are many improvements we would like to make to the first-order instance of Imogen. In this section we briefly outline some of the most important.

Polarity Assignments. It is known [Chaudhuri et al., 2008] that assigning positive polarity to atoms can simulate backward chaining in the inverse method. This is good when a specification has a natural backward interpretation, e.g. as Horn clauses. When we are outside of a well understood fragment like Horn clauses

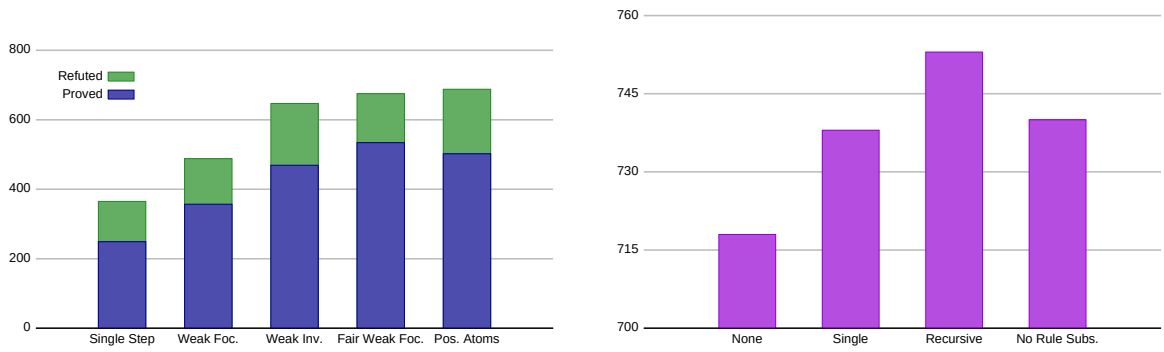

Figure 17: Polarization and Subsumption Experiments

however, it becomes difficult to know how to assign atom and conjunction polarity. It is important to discover good heuristics, because the impact of the assignment is often dramatic.

Subordination. The process of labeling leaves us with sequents made up of collections of atomic predicates. It may be that some collections could never arise in an actual proof. This observation was made earlier in the the discussion of path calculi (Section 2.4). We can carry this observation further. Define a *subordination relation* on the atomic predicates by the transitive closure of the rules shown in Figure 18. We never need to generate a sequent of the form $\Gamma, A \longrightarrow B$ where $A \not\leq B$.

Subterm Property. The subformula property is a powerful tool at pruning the search space in the cut-free sequent calculus. Unfortunately this is no help whatever in controlling the *terms* that appear in sequents. The subformula property is mute with respect to terms. Given a rule such as

$$\frac{\longrightarrow A(x)}{\longrightarrow A(s(x))}$$

can lead to the inverse method building sequents with larger and larger terms. Often these spurious derivations are totally useless. We can sometimes recover a form of *subterm property* by a *local*, i.e. rule-by-rule, examination of the inference rules. Call a rule *expansive in* A if the occurrences of the arguments of A in the premise are subterms of the arguments in the conclusion. The rule shown above is expansive. Call a rule *contractive* if the opposite is the case. A simple observation is that, if all of the rules of a calculus are either expansive in A, or all the rules are contractive in A, then we need only consider sequents where A is applied to subterms of the arguments of A in the goal. With more work, it seems possible to deduce analogous *global*, i.e. considering all the rules together, subterm properties of a calculus. Since the subterm property will be essential in the LF theorem prover (see Example B), we intend to implement this optimization immediately and explore its ramifications.

Logic Programming. Though Imogen is empirically successful on many first-order problems, it performs poorly on other very natural problems. For example, in logic programs expressed as Horn clauses, while we can simulate the reasoning with positive atoms, many useless sequents can also generated. Consider the Horn clauses

Initial Rules[a]

$$\frac{}{P \leq P}$$

Right Rules

$$\frac{A \leq B}{A \leq B \wedge C} \qquad \frac{A \leq C}{A \leq B \wedge C} \qquad \text{No rule for } A \leq \top$$

$$\frac{A \leq B}{A \leq B \vee C} \qquad \frac{A \leq C}{A \leq B \vee C} \qquad \text{No rule for } A \leq \bot$$

$$\frac{A \leq C}{A \leq B \supset C} \qquad \frac{A \leq B}{A \leq \forall x.\ B} \qquad \frac{A \leq B}{A \leq \exists x.\ B}$$

Left Rules

$$\frac{A \leq C}{A \wedge B \leq C} \qquad \frac{B \leq C}{A \wedge B \leq C} \qquad \text{No rule for } \top \leq C$$

$$\frac{A \leq C \quad B \leq C}{A \vee B \leq C} \qquad \frac{A \supset B \leq C}{B \leq C} \qquad \text{No rule for } A \leq \bot$$

$$\frac{A \leq B}{\forall x.\ A \leq B} \qquad \frac{A \leq B}{\exists x.\ A \leq B}$$

Transitivity

$$\frac{A \leq B \quad B \supset C \leq D}{A \leq D}$$

[a] The init rule is only applicable if the atom occurs with both left and right sign.

Figure 18: Subordination Rules

$$\forall x.\ \texttt{plus}(0, x, x)$$
$$\forall x\ y\ z.\ \texttt{plus}(x, y, z) \supset \texttt{plus}(s(x), y, s(z))$$

Assigning positive polarity to `plus` gives the inference rules

$$\frac{\texttt{plus}(0, x, x) \longrightarrow \cdot}{\cdot \longrightarrow \cdot} \qquad \frac{\texttt{plus}(s(x), y, s(z)) \longrightarrow \cdot}{\texttt{plus}(x, y, z) \longrightarrow \cdot}$$

These rules can clearly simulate SLD resolution. However, they are also less well-behaved. For instance, if the database contains the initial sequent $\texttt{plus}(x_1, y_1, z_1) \longrightarrow \texttt{plus}(x_1, y_1, z_1)$ then it is possible to apply the second rule by unifying x_1 with $s(x)$, etc. The newly generated sequent can then be used to match the rule again, yielding an infinite regression. In general, forward chaining can generate a lot of sequents that are never used in any proof. This phenomenon has been investigated in the logic programming community. One solution has been dubbed *magic sets* [Bancilhon et al., 1986, Beeri and Ramakrishnan, 1991]. It is not clear how to extend this technique to non-Horn programs, but it would be interesting to explore.

Implementation. Imogen implements subsumption checks using path indexing and substitution tree indexing. The former is generally faster, even though it is not a perfect filter. It would be interesting to experiment with other term indexing techniques such as code trees [Voronkov, 1995]. As the implementation becomes more mature, it might also be advantageous to explore more efficient term representations like flatterms, and implement a efficient unification algorithm, though we do not expect to implement this for the thesis.

In this section we extended the polarized inverse method to first order logic. While a good general reasoning tool, first order logic can often not capture efficient domain-specific reasoning, for example reasoning in particular theories like linear arithmetic. In the next section we extend the sequent calculus to allow such domain specific reasoning using constraints.

Part II
Applications

6 Constraints

In the first order case, the generation of new facts by matching was accomplished via unification. The antecedents and succedent are unified with various parts of the premise of an inference rule, and the resulting unifier is applied to the conclusion of the rule. In LF, unification is undecidable. Therefore we will eagerly do unification when the unified terms fall in the decidable pattern fragment, but in general we will not be able to solve the unification equations immediately. Therefore it is necessary to postpone solving some unification equations, and keep them as *constraints* of the sequent. To prepare for this eventuality, in this section we describe the implementation of an inverse method theorem prover for first-order intuitionistic logic with generic constraints. We chose to extend the language of first-order logic to include unification equalities, so as to be analogous to the LF case. Note that the equality denotes free term equality, not the traditional Leibniz equality. For example, $a \doteq b \supset \bot$ where a, b are distinct constants is a valid formula in this theory.

$$\text{Formulas } A ::= t_1 \doteq t_2 \mid t_1 \not\doteq t_2 \mid \ \ldots$$

The prover is parameterized over a given set of function symbols with arity. For example, if the allowed constants are $(z, 0), (s, 1)$, then some valid formulas of this logic are

$$\forall X\ Y.\ f(X) \doteq f(Y) \supset X \doteq Y$$
$$\forall X.\ s(X) \doteq z \supset \bot$$
$$\forall X.\ X \not\doteq z \supset \exists Y.\ X \doteq s(Y)$$

We recently implemented an Imogen instance enriched with this new interpreted predicate. Since equalities can occur in a negative position, we decide the domain with *disunification* over finite trees [Comon and Lescanne, 1989]. In this section we describe the constraint sequent calculi underlying our implementation that will be the basis of unification constraints for LF. While we chose the particular theory of first-order disunification for a concrete implementation, most of the discussion that follows holds for many useful domains. One application that we are currently investigating is linear arithmetic constraints, in the context of verifications in constructive authorization logic by Garg [Garg and Pfenning, 2006]. Another rich source of examples for future work is a generic encoding of focused proofs in substructural logics into focused intuitionistic logic with a preorder. Examples include constructive and classical linear logic, ordered logic, and the logic of bunched implications, as seen in Pfenning and Reed [Pfenning and Reed, 2009].

We start from a cut-free backward sequent calculus for an intuitionistic logic augmented with constraints. Proofs appeal to a notion of constraint entailment which must satisfy some basic properties but is otherwise left unspecified. From this we systematically derive a forward sequent calculus with constraints which serves as an appropriate foundation for the inverse method. The main complications arise from quantifier alternations and the interactions between quantification in the constraint domain and quantification in the underlying logic. We are currently working on proofs of the following properties: (1) the forward system is sound and complete with respect to the backward system, and (2) the constraint sequent calculus is conservative over the pure logic on one hand and the constraint domain on the other.

6.1 Backward Constraints

We make the presentation slightly general. We assume we are given a theory $\mathcal{T}$ with an entailment relation $\Psi \models \Psi'$ and a finite first-order axiomatization $\mathbf{Ax}$. The intention is that reasoning with formulas from the theory will take place outside the sequent calculus. We write a backward constraint sequent as $\Psi \mid \Gamma \Longrightarrow A$ where Ψ is from the language of $\mathcal{T}$. In this proposal, the theory $\mathcal{T}$ is that of equalities and disequalities of

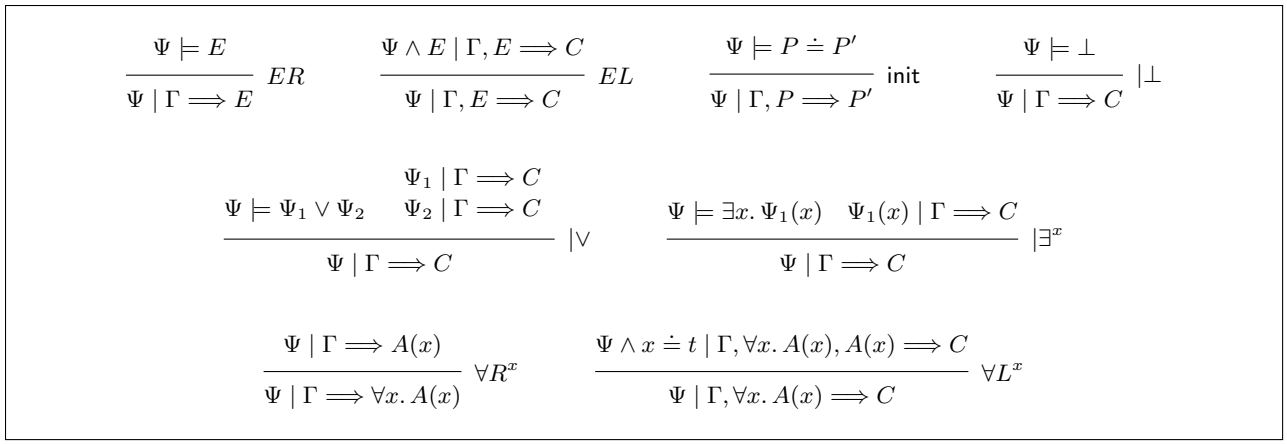

Figure 19: Backward Constraint Calculus

finite trees. The backward rules that affect the constraints are shown in Figure 19. The middle rules $\mid\!\vee, \mid\!\exists^x$ should be necessary only if the theory is not convex (though see 6.5).

6.2 Subsumption

To even express the completeness theorem we wish to hold between the backward and forward calculus, we need to define subsumption. Subsumption is complicated by the presence of constraints. The definition we use is

Definition 6.1. $\Psi_1 \mid \Gamma_1 \longrightarrow \gamma_1$ *subsumes* $\Psi_2 \mid \Gamma_2 \longrightarrow \gamma_2$ if there is a renaming substitution σ such that $\Psi_2 \models \Psi_1\sigma$, $\Gamma_1\sigma \subseteq \Gamma_2$ and $\gamma_1\sigma \subseteq \gamma_2$.

6.3 Forward Constraints

We can now design forward rules that are sound and complete with respect to the backward rules. The rules are shown in Figure 20. Given some basic properties of the $\models$ judgment, we have proof sketches of the soundness and completeness of the forward calculus with respect to the backward calculus.

Theorem 6.1 (Soundness)**.** *If* $\Psi \mid \Gamma \longrightarrow A$ *then* $\Psi \mid \Gamma \Longrightarrow A$*. If* $\Psi \mid \Gamma \longrightarrow \cdot$ *then* $\Psi \mid \Gamma \Longrightarrow A$ *for any* A*.*

Theorem 6.2 (Completeness)**.** *If* $\Psi \mid \Gamma \Longrightarrow A$ *then there are* Ψ', Γ', γ' *such that* $\Psi' \mid \Gamma' \longrightarrow \gamma'$ *and* $\Psi' \mid \Gamma' \longrightarrow \gamma'$ *subsumes* $\Psi \mid \Gamma \longrightarrow A$*.*

Though the backward and forward calculi are strongly related by the above theorems, the theorems linking the constraint calculus to a concrete semantics are still lacking. This is work to consider in the near future.

6.4 Implementation

We implemented an Imogen front end for constraints. Since the theory is less well understood, we consider it a prototype at this time. The only novelty apart from handling constraints as described above is that all unification occurs in the constraints, using the disunification algorithm. This was done to give us some idea

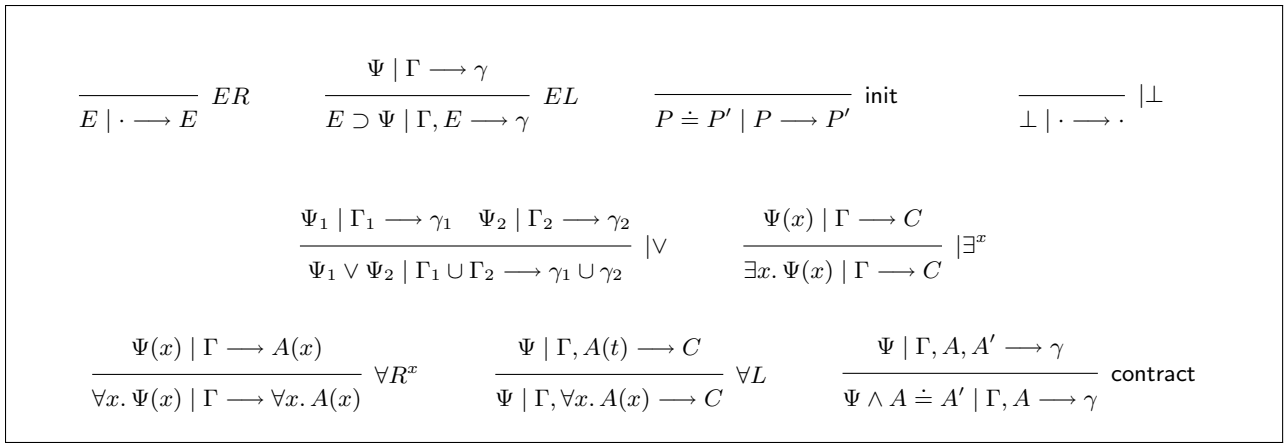

Figure 20: Forward Constraint Calculus

of the coming complications of undecidable unification. While the implementation is less efficient than the first order prover described above, it is still competitive on purely first-order theorems with any of the other provers from the ILTP database.

6.5 Future Work

Applications. After working out the theory to our satisfaction, our next steps in this direction will be to complete constraint solvers for Garg's authentication logic, and Pfenning and Reed's preorder for intuitionistic linear logic.

Richer Notions of Subsumption. Our definition of subsumption makes intuitive sense. The constraints of the subsuming sequent must be weaker (contravariant) than those of the subsumed sequent. However, this can fail to account for situations that might be necessary for completeness. [7]

Example 6.1. Suppose we are given two sequents in a constraint calculus where the domain is a dense linear order.

$$Q_1 = X < Z \mid q(U), r(X, Y), r(Y, Z) \longrightarrow p(U)$$
$$Q_2 = X < Y \mid q(U), r(X, Y) \longrightarrow p(U)$$

Then for any grounding substitution satisfying Q_1 there is a related substitution satisfying Q_2, independent of the meaning of p, q and r. This is because if for some values X, Y, Z of the variables, if $X < Z$ then either $X < Y$ or $Y < Z$. That means if u, x, y, z are values satisfying $x < z \mid q(u), r(x, y), r(y, z) \longrightarrow p(u)$ then either $U = u, X = x, Y = y$ satisfies Q_2 or $U = u, X = y, Y = z$ satisfies Q_2. In this sense, Q_1 is subsumed by Q_2, even though it is not subsumed in our definition of subsumption.

Example 6.2. With constraints, two or more sequents may combine to be "stronger than" another sequent. For example:

[7] Thanks to Michael Maher for these examples.

$$Q_0 = \top \mid q(U), r(U) \longrightarrow p(U)$$
$$Q_1 = X > Y \mid q(U) \longrightarrow p(U)$$
$$Q_2 = X \leq Y \mid q(U) \longrightarrow p(U)$$

Then Q_1 and Q_2 together subsume Q_0. This can be accounted for using our notion of subsumption and the disjunction rule of the calculus. However, we would hope to not need such a rule in a convex domain like dense linear orders.

Despite these examples, it is still possible to define forward sequents for a general calculus. Concerns such as those raised by the above examples will be specific to proof that the calculus is complete with respect to a given domain. The definition of subsumption may thereby need to be strengthened accordingly.

Leibniz Equality. Also of great interest to us is the case of Leibniz equality. It is known [Degtyarev and Voronkov, 2001a] that a proof procedure for handling Leibniz equality in intuitionistic logic is equivalent to simultaneous rigid E-unification, which is undecidable. Thus, the entailment relation for a constraint domain of equality would be difficult to implement efficiently. Since equality is ubiquitous in first-order theorem proving, handling this case efficiently is highly desirable. In the case of LF, there is a sizable subset, the pattern fragment, that is decidable. Unification problems that fall inside this fragment can be decided, and the resulting substitution applied accordingly. Only equations outside the pattern fragment need to be kept as constraints. Perhaps there is an analogous situation for first-order Leibniz equality.

In this section we have outlined a theory of constraints for the inverse method. We have recently implemented the theory described here, though the results are not as thorough as the previous chapters. The remainder of this proposal is even more speculative in nature. We have not yet attempted implementations of the next sections, nor have we worked out the theory in minute detail. Thus, the sections will be correspondingly shorter and will be example oriented, and give more motivations than results.

7 First-Order Induction

Recall that the eventual goal of this thesis is an inductive theorem prover for $\mathbf{M}_2^+$, the meta-language of Twelf. In this section, we will describe how we intend to the simpler problem of induction in first-order logic using the polarized inverse method. To this end, we will extend intuitionistic logic with inductive definitions.

7.1 The $\mathbf{M}_2^+$ Loop

Following Schürmann [Schürmann, 2000], we can build an inductive theorem prover for first-order logic using a generic strategy we call the $\mathbf{M}_2^+$ *loop*. We can combine the $\mathbf{M}_2^+$ loop together with our first-order intuitionistic prover to yield a first-order theorem prover for first-order induction. The loop is comprised of three operations: filling, splitting, and recursion, and is shown graphically in Figure 21.

Filling. The *proof stack* is a collection of sequents. If each of these are proved, then the overall goal is true. The proof stack begins containing only the goal sequent. If the proof stack is empty, the theorem is proved. Given a sequent to prove from the proof stack, the loop first attempts to find a direct proof. This is called *filling* because it uses an underlying object-level theorem prover to attempt to fill any existentially quantified variables. In Twelf, the underlying prover is the backward prover for LF. In this work it will be the first-order inverse method prover described in Section 5. If the prover can find a witness for each such variable, the case is closed, and the process returns to select a new element of the proof stack.

Splitting. If filling fails, the loop chooses a variable on which to do case analysis, or *splitting*. The variable is chosen by a heuristic. Splitting generally leads to a number of new subgoals. Since splitting could be done indefinitely, we include a bound on the depth to which a variable can be split. If there are no variables in the sequent below that depth, the proof procedure fails.

Recursion. Finally, for each new subgoal from the splitting phase, the loop finds all the possible ways to use the induction hypothesis on the fresh variables. This process is finite and deterministic, and thus never fails. The sequents derived from splitting, augmented with new induction hypotheses, are returned to the proof stack, and the process repeats.

7.2 Induction

To implement the recursion phase, we need to know what counts as an induction hypothesis. There are a number of ways to add induction to a logic. One, as pursued in [Baelde, 2008], is to extend the language of formulas with syntactic equality and fixpoint operators. This solution would extend the syntax to

$$\text{Formulas } A ::= t_1 \doteq t_2 \mid \mu B \vec{t} \mid \nu B \vec{t} \mid \cdots$$

where ι and o are the types of individuals and formulas respectively (following Church), $\mu, \nu : (\vec{\iota} \to o) \to (\vec{\iota} \to o)$ and B is a second order predicate $B : \vec{\iota} \to o$, denoting a least and greatest fixpoint operator respectively. Then typical inductive definitions can be defined, for example

$$\text{nat } n \stackrel{\text{def}}{=} \mu(\lambda \text{nat } \lambda n.\ n \doteq 0 \vee \exists n'.\ n = Sn' \wedge \text{nat } n')\ n$$
$$\texttt{plus}\ n\ m\ k \stackrel{\text{def}}{=} \mu(\lambda \texttt{plus}\ \lambda n.\ (n \doteq 0 \wedge m \doteq k) \vee \exists n'\ k'.\ n = Sn' \wedge k = Sk' \wedge \texttt{plus}\ n'\ m\ k')\ n$$

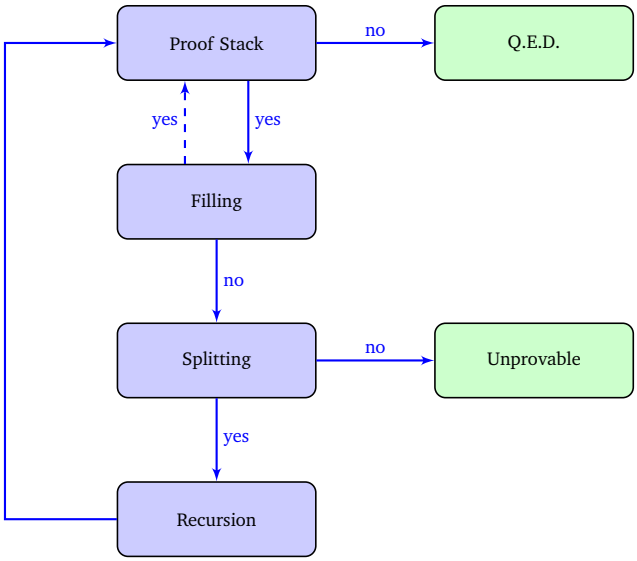

Figure 21: The $\mathbf{M}_2^+$ Loop

Then left and right rules are introduced for the new connectives. S is called the *invariant* for the induction.

$$\frac{\Gamma, S\ t \Longrightarrow P \quad B\ S\ x \Longrightarrow S\ x}{\Gamma, \mu B t \Longrightarrow P}\ \mu\text{-L} \qquad \frac{\Gamma \Longrightarrow B(\mu B)t}{\Gamma \Longrightarrow \mu B t}\ \mu\text{-R}$$

$$\frac{\Gamma, B(\nu B)t \Longrightarrow P}{\Gamma, \nu B t \Longrightarrow P}\ \nu\text{-L} \qquad \frac{\Gamma \Longrightarrow S\ t \quad S\ x \Longrightarrow B\ S\ x}{\Gamma \Longrightarrow \nu B t}\ \nu\text{-R}$$

While this system is elegant and general, it seems difficult to implement in the forward (or even backward) direction in its full generality because of the unknown invariant S. Instead of reaching for a fully general inductive theorem prover, we can syntactically restrict the kinds of formulas for which we will search. Our goal is a Twelf meta-theorem prover, which is restricted to Π_2^0 sentences, (those having the form $\forall \vec{x}.\ \exists \vec{y}.\ \phi$ where ϕ is quantifier free). It seems natural therefore to restrict ourselves to the same fragment for the first-order case. To have a chance of success, we will further restrict the second-order variable S. A natural choice for S is the goal formula itself. This means we restrict our efforts to formulas that can be proven directly by induction. No attempt will be made at generalizing the induction hypothesis. With these simplifications, we have a rather straightforward way of implementing induction using the $\mathbf{M}_2^+$ loop. These restrictions are roughly the same as those on the current Twelf theorem prover, and thus if we can build an empirically successful inductive theorem prover for first-order logic using the inverse method, it will be evidence that the method can scale to LF with induction.

Example. An extend example of our proposed method of first-order inductive theorem proving is given in Appendix A, where we use the method outlined above to prove that list reversal is an involution.

7.3 Possibilities to Explore

Of course, a lot can be done to improve on this design to make a useful and practical tool.

Focusing. We've avoided the discussion of focusing above, primarily because we have not yet explored how it will interact with the generated induction rules. Focusing is described in detail in [Baelde, 2008], and completeness results are proved for the inductive first-order system μLJ. However, our treatment of induction as a meta-logical makes Baelde's results not directly applicable.

Types. In untyped first-order logic, types must be defined explicitly as predicates. For instance, in the list reverse example of Appendix A, we needed to define lists via the `list` predicate. This leads to additional proof obligations. In LF this is not necessary because `list` is an inductively defined type rather than a first-order predicate. It would be interesting to add types to the untyped system given above, as in LF. This will make the theorem proving process closer to what we will expect to happen in the $\mathbf{M}_2^+$ prover, and will then tell us more about the expected behavior of our proposed system.

Coinduction. While Twelf doesn't support coinductive definitions directly, it seems interesting, and of little additional labor, to include them in this prover. This would allow attempts at proving results about coinductive types such as streams and finite state machines.

8 LF: A Logical Framework

We have finally reached the heart of our proposed contribution, a new theorem prover for the logical framework LF based on the polarize inverse method. There are three existing implementations of theorem provers for LF or some fragment thereof. Two of them are part of Twelf. These are the Elf logic programming engine as originally described in [Pfenning, 1989]. The other is Pientka's tabled logic programming engine [Pientka, 2003]. Both of these logic programming interpreters are extended to theorem provers using iterative deepening in Twelf. The third existing implementation, also by Pientka, implements the first-order Horn clause subset of LF via the focused inverse method [Pientka et al., 2007]. This third theorem prover is the one closest to our intended design. As such, we will review the definitions from Pientka's paper, and then discuss the issue of unification constraints that fall outside the decidable pattern fragment. We finish by arguing that the inverse method will provide a basis for a theorem prover with a substantially different performance profile than the existing provers.

8.1 LF

LF [Harper et al., 1993, Pfenning, 2001] a framework for representing deductive systems. The language, sometimes called λ^Π, is a dependent type theory. Using the Martin-Löf style of judgments-as-types, one can use λ^Π to represent deductive systems. The language is divided as usual into terms and types, with the difference that terms can appear in types. *Kinds* classify types, providing for the declaration of *type families*. Type families will be the primary object of study in the meta-theorem prover. Here we give the syntax of LF canonical forms.

Kinds	$K ::= \texttt{type} \mid \Pi x : A.\ K$
Types	$A ::= a\ M_1\ \ldots\ M_n \mid A_1 \to A_2 \mid \Pi x : A_1.\ A_2$
Normal Objects	$M ::= \lambda x.\ M \mid R$
Neutral Objects	$R ::= x \mid c \mid R\ M$

Here we are concerned with the theorem proving problem; given a type A, find a term T such that $T : A$. We do not show the full typing rules here, as they run at least three pages and are given precisely in Harper [Harper and Licata, 2007]. Here we will concern ourselves with the sequent calculi. We mostly follow Pientka [Pientka et al., 2007], though we augment the system with positive atoms and shift operators.

Polarized LF. We refine LF with positive and negative atoms. Objects remain the same.

Kinds	$K ::= \texttt{type} \mid \Pi x : A^+.\ K$
Positive Types	$A^+ ::= a^+\ M_1\ \ldots\ M_n \mid \downarrow A^-$
Negative Types	$A^- ::= a^-\ M_1\ \ldots\ M_n \mid A_1^+ \to A_2^- \mid \Pi x : A_1^+.\ A_2 \mid \uparrow A^+$

Note that the arrow $A \to B$ may be abbreviated as $\Pi x : A.\ B$ where x does not occur in B. However, for our purposes we wish to separate them. Hypotheses introduced with an arrow will be used (e.g. unified, case analyzed) during proof search, while those introduced with a Π become parameters. One of the most difficult tasks of this proposed thesis will be building a polarized inverse method theorem prover for LF.

8.2 Sequent Calculus

A backward calculus for LF is simpler than that for intuitionistic logic because there are so few connectives. The backward calculus is shown in Figure 22. Focusing is straightforward, as all the connectives are negative.

We need only consider positive atoms in focus on the right. To lift the ground calculus to free variables and proof search, we will follow [Pientka et al., 2007, Nanevski et al., 2008] and introduce contextual modal type variables in order to account for dependencies between variables. The polarities add no additional complications, and the results carry over directly.

Proof Terms. Proof terms will be an important aspect of the theorem prover, as the witnesses will be required by the meta-theorem prover. Proof terms for LF are well studied, and can be found in [Schürmann, 2000, Pientka, 2003].

Example. An example of using the polarized inverse method for LF with higher-order terms is given in Appendix B.

Related Work. There is a piece of related work that we feel deserved particular attention due to its relevance to our study. Pientka et al. [Pientka et al., 2007] describe a focused inverse method theorem prover for the Horn fragment of LF. This paper is interesting because it shows the inverse method performing poorly vis-à-vis the existing backward tabled search methods of Twelf, especially when the selection of which predicates to table is hand-tuned by the user. While this is the closest related work, it differs from ours in substantial ways. For example, [Pientka et al., 2007] doesn't assign explicit polarities to formulas, including atoms. Atoms are automatically considered negative. This leaves room for our implementation to consider alternative search spaces [Chaudhuri et al., 2008]. Also, our implementation seem to differ markedly with respect to performance. For instance, computing the 18th Fibonacci number using their implementation was significantly slower than the tabled logic programming engine of Twelf while our implementation was significantly faster. Moreover, the examples from that paper aren't particularly good examples of the intended use of such a theorem prover. While the inverse method will certainly not be able to compete with backward chaining on many algorithm-like specifications because of the space overhead and the problems described in Section 5, its primary use will be as a subroutine of an inductive theorem prover for $\mathbf{M}_2^+$. The requirements for the LF prover in this respect are very different than the requirements of a logic programming engine designed to execute Prolog-like specifications written as Horn clauses.

In this section we briefly described LF and a polarized sequent calculus that will form the basis for an inverse method prover. While this is only a sketch, we have confidence that the method will be applicable. An extended example is found in Appendix B. In the next section we describe the final logic of our proposal, $\mathbf{M}_2^+$.

Right Inversion $\boxed{\Gamma; \Delta^+ \Uparrow A^-}$

$$\frac{\Gamma; \Delta^+ \Uparrow a^- \ M_1 \ \ldots \ M_k}{\Gamma; \Delta^+ \Uparrow a^- \ M_1 \ \ldots \ M_k} \text{ (RA-Atom)} \qquad \frac{\Gamma; \Delta^+ \Uparrow A^+}{\Gamma; \Delta^+ \Uparrow \uparrow A^+} \text{ (RA-}\uparrow\text{)}$$

$$\frac{\Gamma; \Delta^+, x : A_1^+ \Uparrow A_2^-}{\Gamma; \Delta^+ \Uparrow \Pi x : A_1^+ .\ A_2^-} \text{ (RA-}\Pi^x\text{)} \qquad \frac{\Gamma; \Delta^+, A_1^+ \Uparrow A_2^-}{\Gamma; \Delta^+ \Uparrow A_1^+ \to A_2^-} \text{ (RA-}\to\text{)}$$

Left Inversion $\boxed{\Gamma; \Delta^+ \Uparrow C}$

$$\frac{\Gamma, a^+ \ M_1 \ \ldots M_k; \Delta^+ \Uparrow C}{\Gamma; \Delta^+, a^+ \ M_1 \ \ldots \ M_k \Uparrow C} \text{ (LA-Atom)} \qquad \frac{\Gamma, A^-; \Delta^+ \Uparrow C}{\Gamma; \Delta^+, \downarrow A^- \Uparrow C} \text{ (LA-}\uparrow\text{)}$$

Right Focusing $\boxed{\Gamma \Downarrow [A^+]}$

$$\frac{}{\Gamma, a^+ \ M_1 \ \ldots \ M_k \Downarrow [a^+ \ M_1 \ \ldots \ M_k]} \text{ (RS-Atom)} \qquad \frac{\Gamma; \cdot \Uparrow A^-}{\Gamma \Downarrow [\downarrow A^-]} \text{ (RS-}\downarrow\text{)}$$

Left Focusing $\boxed{\Gamma; [A^-] \Downarrow C}$

$$\frac{}{\Gamma; [a^- \ M_1 \ \ldots \ M_k] \Downarrow a^- \ M_1 \ \ldots \ M_k} \text{ (LS-Atom)} \qquad \frac{\Gamma \vdash M : A_1^+ \quad \Gamma, [M/x]A_2^-; [C] \Downarrow}{\Gamma; [\Pi x : A_1^+ .\ A_2^-] \Downarrow C} \text{ (LS-}\Pi\text{)}$$

$$\frac{\Gamma \Downarrow [A_1^+] \quad \Gamma; [A_2^-] \Downarrow C}{\Gamma; [A_1^+ \to A_2^-] \Downarrow C} \text{ (LS-}\to\text{)} \qquad \frac{\Gamma; A^+ \Uparrow C}{\Gamma; [\uparrow A^+] \Downarrow C} \text{ (LS-}\uparrow\text{)}$$

Stable Rules

$$\frac{\Gamma \Downarrow [A^+]}{\Gamma; \cdot \Uparrow A^+} \text{ (FocusR)} \qquad \frac{\Gamma, A^-; [A^-] \Downarrow C}{\Gamma, A^-; \cdot \Uparrow C} \text{ (FocusL)}$$

Figure 22: Backward Polarized Calculus for LF

9 $\mathbf{M}_2^+$

The overall goal of this thesis is a new theorem prover for $\mathbf{M}_2^+$ based on the polarized inverse method. In this section we will briefly sketch the existing $\mathbf{M}_2^+$ theorem prover. Then we argue that the proposed prover will potentially be a significant improvement.

$\mathbf{M}_2^+$ is the meta-logic of Twelf. It is a functional programming language whose types are LF type families. Total functions in $\mathbf{M}_2^+$ realize Twelf meta-theorems. $\mathbf{M}_2^+$ was devised by Schürmann [Schürmann, 2000] to formalize the meta-theory of Elf. $\mathbf{M}_2^+$ itself is not exposed to the Twelf user as a language. Instead, Twelf constructs $\mathbf{M}_2^+$ programs from Elf programs and schema checking. In the following section we will describe roughly how the current Twelf theorem prover constructs proofs. The $\mathbf{M}_2^+$ logic is quite complicated. Describing it in detail is beyond the scope of this proposal. The interested reader can refer to [Schürmann, 2000] and $\mathbf{M}_2^+$'s successor Delphin [Poswolsky, 2008]. This section will be accordingly abstract, with few details.

9.1 Current Twelf Theorem Prover

As we discussed in Section 7, the current Twelf meta-theorem prover uses the $\mathbf{M}_2^+$ loop that consists of three principal operations during proof search: filling, splitting, and recursion. Currently only the backward LF prover can be used with this strategy. The original design uses the backward chaining logic programming engine of Twelf. Since backward chaining may easily loop, Schürmann added a depth bound, and maintains completeness up to that bound with iterative deepening.

Example. An example of the proposed functioning of a meta-theorem prover based on the inverse method LF theorem prover is given in Appendix C.

9.2 Analysis

Note that the design above is agnostic with respect to the procedure used for filling. Currently only Schürmann backward can be used, though Pientka's could be used with minor changes to Twelf. We propose to add a third in the form of the inverse method prover proposed in Section 8. We can think of two primary reasons why the inverse method stands to be an improvement over the current implementations.

Failing Quickly. In inductive theorem proving, *failing quickly* is perhaps more important than *succeeding*. The splitting phase, i.e., selecting variables for case analysis and induction, will in general create many possibilities that will not lead to a solution. This occurs, for instance, when the heuristic selects a variable that will not make progress toward a solution. The longer it takes to realize that such a branch is fruitless, the worse the performance of the theorem prover. Indeed, the vast majority of the time the current prover spends is on exhausting the depth bound of fruitless branches. This makes the prover very delicate, depending critically on a seemingly arbitrary depth bound for the iterative deepening search.

Monotonic Search. In the current prover, the information learned on any branch is discarded when a new branch is explored, both during iterative deepening, and when the depth bound is exceeded. This is inherent in backward chaining algorithms, where meta-variables are shared between branches. In contrast, the inverse method generates only valid sequents. Anything that is learned in one branch is valid in another. Forward chaining thus gives the same benefits as Pientka's tabling, but also saves information throughout different subgoals, and even across filling and recursion phases. This can prevent the same sequents from being generated multiple times, and we conjecture that it will be more efficient in many cases.

9.3 Goals

Now that we have motivated our proposed meta-theorem prover based on the LF inverse method prover, we will highlight some of the required steps we will take once the LF prover is complete.

New Search Procedures. The simple strategy of filling, splitting and recursion is only one of many possible designs. After exploring the performance of the new prover using the existing strategy, we can explore other top-level ideas. One example is to use inversion of formulas on the left of a sequent, which is not allowed in the current design.

Properties of the Object Logic. As we've seen in the first part of this proposal, the subformula property is absolutely essential to having a decent proof search strategy for a logic. One might argue that Twelf's *raison d'être* is to encode and reason about exactly such logics. Might it be possible to discover that encoded object logics have the subformula property and use this fact in LF's proof search strategy? By restricting the LF *terms* to object logic subformulas of the goal, the performance of the LF prover and meta-theorem prover would be much more effective. This is equally true of slightly more sophisticated relations like subordination and subterm properties that were discussed in earlier sections. If we can syntactically deduce structural properties of the object logic, we will be much more successful.

9.3.1 Proof Terms

The current Twelf meta-theorem provers provide no proof terms of any kind.[8] The user must simply trust the output conclusion of the system. Given the Twelf implementation's complexity, this is not ideal. We propose to construct two different kinds of proof terms for the output of the meta-theorem prover, both realizing the proofs as programs.

Elf programs. A simple form of proof term will be to construct an Elf logic program as the output of the theorem prover. In the best case, this would be similar to the program that the user would have written if the theorem prover was not available. The user can then record the proof term and check it with Twelf's mode, world, and totality checkers as is the usual practice. This method has been attempted before, though there are some difficulties.

Delphin programs. A more interesting idea is to generate programs in the Delphin language [Poswolsky, 2008]. Delphin is a functional dependently-typed language similar to $\mathbf{M}_2^+$, and it seems straightforward to generate Delphin terms during the splitting, filling, and recursion phases such that the final Delphin program will check against the type of the original goal. This would be a more satisfying method of proof term generation because Delphin has a totally independent code base from Twelf. Beluga [Pientka, 2008] is an alternative target language.

[8] Originally the meta-theorem prover produced Elf programs as proof witnesses. For technical reasons, the current implementation does not support that capability.

10 Conclusion

In conclusion, we have proposed as a thesis topic the design and implementation of a number of different logics for which we intend to implement theorem provers based on the polarized inverse method. The propositional and first-order logic cases are mature, though we intend to experiment with numerous method to reduce the search space and improve efficiency. The prover for first-order constraints is at an advanced stage of development. Considerable theoretical details still need to be worked out however, and this will be some of the first work we undertake. The bulk of the remaining effort will be the design and implementation of theorem provers for first-order logic with induction, LF, and $\mathbf{M}_2^+$. We intend the $\mathbf{M}_2^+$ meta-theorem prover to be a useful and practical tool for Twelf developers. In this section we describe some related work, and work that we don't intend to undertake in this thesis, but which is a natural extension of the research presented thus far.

10.1 Related Work

Inverse Method. This proposal builds mainly on work of Chaudhuri [Chaudhuri, 2006]. In his thesis, Chaudhuri describes a focused inverse method theorem prover for intuitionistic linear logic and discusses some details of his implementation *Linprover*. Much of the basis of the theory and implementation of our systems comes directly or indirectly from his work. Besides our different target logics, the most fundamental change in our proposal is to use explicit polarities to guide the search behavior. We also consider more aggressive techniques of redundancy elimination, subordination, and subterm analyses as being essential to the success of a practical inverse method theorem prover.

Tammet [Tammet, 1996] describes a forward theorem prover for intuitionistic first-order logic. While some ideas, in particular inversion and labeling, are shared between our prover and his, ours differs in that it implements full focusing. Note that Tammet's notion of polarity corresponds to our notion of *sign* rather than our notion of polarity.

Voronkov [Voronkov, 1992] describes *strategies* for the inverse method. A strategy is basically a method for decreasing the search space. He also describes [Voronkov, 2001b, Voronkov, 1999, Voronkov, 2000, Degtyarev and Voronkov, 2001b] numerous implementation techniques, and methods for redundancy elimination. He also describes some implementations, for instance an inverse method theorem prover for the modal logic K [Voronkov, 1999]. It would be very interesting to emulate some of the strategies he recommends.

Chaudhuri et al. [Chaudhuri et al., 2008] describes how the inverse method can simulate forward and backward chaining using different atom polarities. Finding heuristics to select atom polarity will be important for the success of our theorem provers.

An alternative to the inverse method and tableaux methods for modal logics are *connection* methods such as Andrews' method of *matings*. Wallen [Wallen, 1990] describes theorem proving in non-classical logics using connection methods in great detail.

Constraints. Constraint logic programming [Jaffar and Lassez, 1987] extends logic programming languages to handle constraint domains. This is implemented as Prolog-style backward search in a sequent calculus, and thus differs from our approach using forward reasoning in the inverse method. Moreover, our system handles arbitrary first order quantification. A backward sequent calculus with constraints for first-order linear logic has been presented by Jia [Jia, 2008] and also by Saranli and Pfenning [Saranli and Pfenning, 2007].

Degtyarev and Voronkov [Degtyarev and Voronkov, 2001a] show how to use constraints to implement a semi-decision procedure for first order intuitionistic logic with equality using an intuitionistic calculus with constraints. Similar to our work, they separate constraint reasoning from logical reasoning. Unlike our work,

details of the equality domain are built into the logical rules. Proof search proceeds by searching for proof skeletons that represent the logical content of the proof, and then separately solving a constraint associated with each skeleton.

Lassez and McAloon [Lassez and McAloon, 1990] give a classical sequent calculus for solving constraints. Their work is orthogonal to ours, in the sense that we ignore the details of the proof system of the constraint domain, instead focusing on the interaction between logical reasoning and constraint reasoning. Rummer gives a classical sequent calculus with constraints [Rümmer, 2008].

LF/$\mathbf{M}_2^+$. The current implementations of LF theorem provers are described in Section 8. There are currently two implementations of backward theorem provers for LF, both part of the Twelf distribution. The most commonly used is by Schürmann [Schürmann, 2000], implementing a Prolog-style search procedure. Pientka [Pientka, 2003] implemented a tabled backward search procedure that is more effective in many domains. The current version of the $\mathbf{M}_2^+$ prover, using both Schürmann and Pientka's LF, is described in Section 9. The Bedwyr system [Baelde et al., 2007] is a logic programming language similar to Elf. It extends traditional logic programming with a logically grounded notion of finite failure and the ∇ generic quantifier. It seems possible to extend our inverse method techniques to this setting, though this will be regarded as future work.

Inductive Theorem Proving. There is a large literature in inductive theorem proving of which we have only scratched the surface. Much of the work is aimed at proving theorems of mathematics, as described in [Bundy et al., 1993, Bundy et al., 1990, Bundy, 2001]. Kreitz and Pientka [Kreitz and Pientka, 2001] describe induction theorem proving, including Bundy's rippling technique, in connection-based methods. They combine rippling with matrix-based constructive theorem proving. Tac [Baelde, 2008] is an inductive theorem prover for the logic μ-LJ, first-order intuitionistic logic extended by fixed and co-fixed points. Reasoning by induction is not a meta-operation, as in most such systems, but is an integral part of the logic. Tac has an interactive component so the user can declare invariants, but will also search for inductive proofs automatically. Inka [Hutter and Sengler, 1996, Autexier et al., 1999] is a classical first order automated inductive theorem prover. It has been developed since the early 1980s and has been used to verify industrial applications. Inka also uses the rippling technique. ACL2 [Kaufmann and Moore, 1997] is another inductive proof assistant targeted at industrial proofs that has been widely used and successful in practice. The logic is basically an unquantified first order logic with induction.

Implementation. Implementation techniques for theorem proving abound. Apart from logical optimizations like focusing that limit the search space, implementation details make an enormous difference in a practical, useful theorem prover. While we have sketched a few implementation techniques in this proposal such as term indexing, much work remains to make the systems we've described, and those we intend to implement reasonably efficient.

One rich source of papers regarding empirically successful techniques is the Vampire first-order theorem prover [Riazanov and Voronkov, 1999], a successful classical first order prover. A selection of the relevant papers for Imogen are [Riazanov and Voronkov, 2003, Riazanov and Voronkov, 2004, Voronkov, 2001a, Nieuwenhuis et al., 2001]. Techniques and statistics for redundancy elimination are given in [Gottlob and Leitsch, 1985, Tammet, 1998]. Term indexing and related optimizations are described in [Graf, 1996, Ramakrishnan et al., 2001].

Tableaux Work Bench [Abate and Goré, 2003] is a system somewhat similar to Imogen for backward tableaux calculi. It allows users to define backward calculi and experiment with proof search.

Proofs of Redundancy Criteria. It would be interesting to have a generic method of proving the completeness of redundancy elimination for the Imogen framework. Generic mechanisms for doing so are given

in [Voronkov, 2001b, Degtyarev and Voronkov, 2001b].

10.2 Future Work

There are a number of ideas that we will not have time to investigate in this thesis. However, we think them interesting enough to record as future work. Some such ideas not presented in the earlier text are given below.

Modal Logic. There are a number of calculi and implementations of theorem provers for full modal logic. [Beckert and Goré, 1997, Voronkov, 1999, Hustadt and Schmidt, 2000] are just a few of these. Since there has been so much activity in this area, there is a wide range of calculi and implementation techniques used. Therefore this would be a good domain to compare the polarized inverse method.

Other Proof Assistants. While our work in this thesis is directed at LF and Twelf, it seems possible to apply roughly the same techniques to other proof assistants based on a higher-order intuitionistic meta language with induction such as Delphin [Poswolsky, 2008], Agda [Coquand, 1998], Beluga [Pientka, 2008] and Abella [Gacek, 2008].

Formalization. Many of the results we've seen so far would would be well served by formalization. As far as we know, there are no formal proofs of the completeness of the inverse method or any of the redundancy elimination strategies we discuss. It would be comforting to have proofs of the soundness and completeness of Imogen's inner loop and the redundancy elimination strategies. While the completeness of focusing has been proved in other settings, for instance [Zeilberger, 2008], proofs for our systems would be welcome.

Acknowledgments

Our work on constraints is based on prior unpublished work with Kaustuv Chaudhuri and Uluç Saranli. We'd like to thank David Baelde for helpful discussions about inductive theorem proving during his visit to CMU. Thanks also to Michael Maher for pointing out the difficulties of subsumption in forward constraint reasoning, and Daniel Lee for informing us about the Twelf Standard ML formalization.

Part III

Appendix

A Example: List Reverse is an Involution

This example demonstrates first-order induction using the $\mathbf{M}_2^+$ loop with Imogen's intuitionistic first-order prover. We wish to prove that list reverse is an involution.

A.1 Paper Proof

Following Martin-Löf's judgments as types principle, we we define lists with the following inference rules. As usual, all judgments are schematic in their variables.

$$\frac{}{\texttt{nil} : \texttt{list}} \qquad \frac{l : \texttt{list}}{\texttt{cons}\ x\ l : \texttt{list}}$$

List reversal is defined inductively as well.

$$\frac{}{\texttt{rev}\ \texttt{nil}\ l\ l}\ \text{rev-nil} \qquad \frac{\texttt{rev}\ l\ (\texttt{cons}\ x\ l_1)\ l_2}{\texttt{rev}\ (\texttt{cons}\ x\ l)\ l_1\ l_2}\ \text{rev-cons}$$

We wish to prove the the following theorem by first-order induction.

Theorem A.1. *For any l_1, l_2, if*

$$\texttt{rev}\ l_1\ \texttt{nil}\ l_2$$

then

$$\texttt{rev}\ l_2\ \texttt{nil}\ l_1$$

To prove this theorem, we'll need a series of lemmas. The first lemma we prove by induction is that `rev` is deterministic.

Lemma A.1. *If $\mathcal{D} : \texttt{rev}\ l_1\ l_2\ l_3$ and $\mathcal{E} : \texttt{rev}\ l_1\ l_2\ l_3'$ then $l_3 = l_3'$.*

Proof. Proof by induction on the derivation $\mathcal{D}$.

Case:

$$\mathcal{D} = \frac{}{\texttt{rev}\ \texttt{nil}\ l_2\ l_2}\ \text{rev-nil}$$

Then

$$\mathcal{E} = \frac{}{\texttt{rev}\ \texttt{nil}\ l_2\ l_2}\ \text{rev-nil}$$

$l_2 = l_3$	Assumption
$l_2 = l_3'$	Assumption
$l_3 = l_3'$	Equality reasoning

Case:

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ \texttt{rev}\ l_1'\ (\texttt{cons}\ x\ l_2)\ l_3\end{array}}{\texttt{rev}\ (\texttt{cons}\ x\ l_1')\ l_2\ l_3}\ \text{rev-cons}$$

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}'\\ \texttt{rev}\ l_1'\ (\texttt{cons}\ x\ l_2)\ l_3'\end{array}}{\texttt{rev}\ (\texttt{cons}\ x\ l_1')\ l_2\ l_3'}\ \text{rev-cons}$$

$\texttt{rev}\ l_1'\ (\texttt{cons}\ x\ l_2)\ l_3$	Inversion on $\mathcal{D}$
$\texttt{rev}\ l_1'\ (\texttt{cons}\ x\ l_2)\ l_3'$	Inversion on $\mathcal{E}$
$l_3 = l_3'$	By Induction on $\mathcal{D}', \mathcal{E}'$

□

Now we give a suitable generalization of the induction hypothesis.

Lemma A.2. *For any l_1, l_2, if*

$$\begin{array}{l}\mathcal{D}_1 : \texttt{rev}\ l_1\ l_2\ l_3\\ \mathcal{D}_2 : \texttt{rev}\ l_3\ \texttt{nil}\ l_3'\\ \mathcal{D}_3 : \texttt{rev}\ l_2\ l_1\ l_3''\end{array}$$

then there is a $\mathcal{D}$ such that

$$\mathcal{D} : l_3' = l_3''$$

Proof. Induction on $\mathcal{D}_1$.

Case:

$$\mathcal{D}_1 = \dfrac{}{\texttt{rev}\ \texttt{nil}\ l_3\ l_3}\ \text{rev-nil}$$

$l_1 = \texttt{nil}$	Assumption
$l_2 = l_3$	Assumption
$\texttt{rev}\ l_3\ \texttt{nil}\ l_3'$	$\mathcal{D}_2$
$\texttt{rev}\ l_3\ \texttt{nil}\ l_3''$	$\mathcal{D}_3$ and equality
$l_3' = l_3''$	By Lemma A.1

Case:

$$\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}_1'\\ \texttt{rev}\ l_1'\ (\texttt{cons}\ x\ l_2)\ l_3\end{array}}{\texttt{rev}\ (\texttt{cons}\ x\ l_1')\ l_2\ l_3}\ \text{rev-cons}$$

$$\mathcal{D}_3' = \dfrac{\begin{array}{c}\mathcal{D}_3\\ \texttt{rev}\ l_2\ (\texttt{cons}\ x\ l_1)\ l_3''\end{array}}{\texttt{rev}\ (\texttt{cons}\ x\ l_2)\ l_1'\ l_3''}\ \text{rev-cons}$$

$l_1 = \texttt{cons}\ x, l_1'$	Assumption
$l_3' = l_3''$	By Induction with $\mathcal{D}_1', \mathcal{D}_2, \mathcal{D}_3'$

□

Finally we can prove Theorem A.1.

Proof. We note in passing that `rev` is *effective*. This means that for any l_1, l_2, there exists l_3 such that `rev` l_1 l_2 l_3. The proof is immediate.

`rev` l_1 `nil` l_2	Assumption ($\mathcal{D}_1$)
`rev` l_2 `nil` l_2'	`rev` is effective ($\mathcal{D}_2$)
`rev` `nil` l_1 l_1	Rule rev-nil ($\mathcal{D}_3$)
$l_1 = l_2'$	Lemma A.2 with $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$
`rev` l_2 `nil` l_1	Equality reasoning

□

A.2 Twelf Encoding

We can easily encode the above reasoning in Twelf.[9] In the next section, we will see how the theorem prover approaches the lemmas and theorem. First, we define lists and reversal.

List Definitions.

```
%{ List elements %}

elem: type.

%{ Lists %}

list : type.

nil : list.
cons : elem -> list -> list.

%{ Reverse %}

rev : list -> list -> list -> type.

rev/nil : rev nil L L.
rev/cons : rev (cons E L1) L2 L3
               <- rev L1 (cons E L2) L3.
```

Since equality is not defined in Twelf, we have to define it ourselves, and prove that `rev` respects equality. This is part of the "equality reasoning" of the paper proof.

[9] This is a slightly modified example due to Carsten Varming from the Twelf Wiki.

Equality.

```
%{ Equality }%

eq : list -> list -> type.

eq/ref : eq L L.

%{ Reverse respects equality %}

rev-eq : eq L1 L1' -> eq L2 L2' -> eq L3 L3'
         -> rev L1 L2 L3 -> rev L1' L2' L3' -> type.
%mode rev-eq +E1 +E2 +E3 +R1 -R2.

- : rev-eq eq/ref eq/ref eq/ref R R.

%worlds () (rev-eq _ _ _ _ _).
%total {} (rev-eq _ _ _ _ _).
```

Lemmas. Now we can show that `rev` is deterministic, Lemma A.1.

```
%{ Reverse is deterministic }%

rev-determ : rev L1 L2 L3 -> rev L1 L2 L4
             -> eq L4 L3 -> type.
%mode rev-determ +R1 +R2 -E.

- : rev-determ rev/nil _ eq/ref.
- : rev-determ (rev/cons R1) (rev/cons R2) E
         <- rev-determ R1 R2 E.

%worlds () (rev-determ _ _ _).
%total R1 (rev-determ R1 _ _).
```

Finally, there is the effectiveness proof, along with Lemma A.2 and Theorem A.1.

```
%{ Reverse is effective }%

can-rev : {L1} {L2} rev L1 L2 L3 -> type.
%mode can-rev +L1 +L2 -D.

- : can-rev nil _ rev/nil.
- : can-rev (cons X L1) L2 (rev/cons D)
            <- can-rev L1 (cons X L2) D.

%worlds () (can-rev _ _ _).
%total D (can-rev D _ _).

%{ Reverse is an involution }%
```

```
revrev-lem : rev L1 L2 L3 -> rev L3 nil L4
                -> rev L2 L1 L5 -> eq L4 L5 -> type.
%mode revrev-lem +R1 +R2 +R3 -E.

- : revrev-lem rev/nil R1 R2 E
          <- rev-determ R2 R1 E.
- : revrev-lem (rev/cons R1) R2 R3 E
          <- revrev-lem R1 R2 (rev/cons R3) E.

%worlds () (revrev-lem _ _ _ _).
%total R1 (revrev-lem R1 _ _ _).

revrev : rev L1 nil L2 -> rev L2 nil L1 -> type.
%mode revrev +R1 -R2.

- : revrev R1 R2
  <- can-rev _ nil R3
  <- revrev-lem R1 R3 rev/nil E
  <- rev-eq eq/ref eq/ref E R3 R2.

%worlds () (revrev _ _).
%total {} (revrev _ _).
```

A.3 Inductive Theorem prover

The goal of the meta-theorem prover is to avoid having to write the proofs above. We will detail the workings of the $\mathbf{M}_2^+$ loop on the proofs that require induction, Lemma A.1 and Lemma A.2. First we give names to the formulas defining `list` and `eq`.

$$\begin{aligned}
\text{rev-nil} &:= \forall l.\ \texttt{rev}\ \texttt{nil}\ l\ l \\
\text{rev-cons} &:= \forall x\ l_1\ l_2\ l_3.\ (\texttt{rev}\ l_1\ (\texttt{cons}\ x\ l_2)\ l_3 \supset \texttt{rev}\ (\texttt{cons}\ x\ l_1)\ l_2\ l_3) \\
\text{eq-refl} &:= \forall l.\ \texttt{eq}\ l\ l \\
\text{defs} &:= \text{rev-nil} \wedge \text{rev-cons} \wedge \text{eq-refl}
\end{aligned}$$

Recall that there are two distinct forms of reasoning. One is the action of the $\mathbf{M}_2^+$ loop, choosing variables to split, and deriving appeals to induction hypotheses. The second is the action of the first-order theorem prover, which is started by the filling phase of the $\mathbf{M}_2^+$ loop. We will write

$$\frac{\text{Hypotheses}}{\text{Goal}}$$

for inductive reasoning with the $\mathbf{M}_2^+$ loop. When reasoning with the first-order theorem prover, we will generate inference rules, which will be written, as usual,

$$\frac{\text{Hypotheses}}{\text{Conclusion}}$$

A.3.1 Lemma: Reverse is Deterministic

We begin with a proof of Lemma A.1.

We begin by asking the inductive prover for a proof of

$$\frac{}{\text{defs} \supset \forall l_1\ l_2\ l_3\ l_3'.\ (\texttt{rev}\ l_1\ l_2\ l_3 \supset \texttt{rev}\ l_1\ l_2\ l_3' \supset \texttt{eq}\ l_3'\ l_3)}$$

After applying the rules $\forall$-R and $\supset$-R, we have the following goal, where c_1, c_2, c_3, c_3' are new constants.

$$\frac{\text{defs},\ \texttt{rev}\ c_1\ c_2\ c_3,\ \texttt{rev}\ c_1\ c_2\ c_3'}{\texttt{eq}\ c_3'\ c_3}$$

We begin in the filling phase by trying a direct proof. The first-order theorem prover is handed the goal formula

$$\text{defs} \supset \texttt{rev}\ c_1\ c_2\ c_3 \supset \texttt{rev}\ c_1\ c_2\ c_3' \supset \texttt{eq}\ c_3'\ c_3$$

Before we generate the rules for `rev` and `eq`, we need to know the polarities of the atoms. The rules for the choices are

Positive

$$\frac{\texttt{rev nil}\ l\ l \longrightarrow}{\longrightarrow \gamma} \qquad \frac{\texttt{rev}\ (\texttt{cons}\ x\ l_1)\ l_2\ l_3 \longrightarrow \gamma}{\texttt{rev}\ l_1\ (\texttt{cons}\ x\ l_2)\ l_3 \longrightarrow \gamma} \qquad \frac{\longrightarrow \texttt{eq}\ l\ l}{\longrightarrow \gamma}$$

Negative

$$\frac{}{\longrightarrow \texttt{rev nil}\ l\ l} \qquad \frac{\longrightarrow \texttt{rev}\ l_1\ (\texttt{cons}\ x\ l_2)\ l_3}{\longrightarrow \texttt{rev}\ (\texttt{cons}\ x\ l_1)\ l_2\ l_3} \qquad \frac{}{\longrightarrow \texttt{eq}\ l\ l}$$

It is clear that we should choose to make `rev` positive for the inverse method. Otherwise we have no control over the search space. Because `rev` is expansive in its second argument, we can ignore sequents we generate with a `rev` subformula such that the second argument is not a subterm of an occurrence of `rev` in the goal. We choose to make `eq` negative. Given these choices, the inference rules are

$$\frac{\texttt{rev nil}\ l\ l \longrightarrow}{\longrightarrow \gamma}\ R_1 \qquad \frac{\texttt{rev}\ (\texttt{cons}\ x\ l_1)\ l_2\ l_3 \longrightarrow \gamma}{\texttt{rev}\ l_1\ (\texttt{cons}\ x\ l_2)\ l_3 \longrightarrow \gamma}\ R_2 \qquad \frac{}{\longrightarrow \texttt{eq}\ l\ l}\ R_3$$

After the stabilization phase, we have the goal

$$\text{defs},\ \texttt{rev}\ c_1\ c_2\ c_3,\ \texttt{rev}\ c_1\ c_2\ c_3' \longrightarrow \texttt{eq}\ c_3'\ c_3$$

The initial sequents are obtained by renaming and then unifying all left and right atomic formulas.

Left Atomic Subformulas	**Right Atomic Subformulas**
$\texttt{rev}\ \texttt{nil}\ l\ l$	$\texttt{eq}\ l_4\ l_3$
$\texttt{rev}\ (\texttt{cons}\ x\ l_1)\ l_2\ l_3$	$\texttt{rev}\ l_1\ (\texttt{cons}\ x\ l_2)\ l_3$
$\texttt{eq}\ l\ l$	
$\texttt{rev}\ c_1\ c_2\ c_3$	
$\texttt{rev}\ c_1\ c_2\ c_3'$	

Unification gives the initial sequents[10]

$$\texttt{eq}\ l\ l \longrightarrow \texttt{eq}\ l\ l$$
$$\texttt{rev}\ \texttt{nil}\ (\texttt{cons}\ x\ l_2)\ (\texttt{cons}\ x\ l_2) \longrightarrow \texttt{rev}\ \texttt{nil}\ (\texttt{cons}\ x\ l_2)\ (\texttt{cons}\ x\ l_2)$$
$$\texttt{rev}\ (\texttt{cons}\ x_1\ l_1)\ (\texttt{cons}\ x_2\ l_2)\ l_3 \longrightarrow \texttt{rev}\ (\texttt{cons}\ x_1\ l_1)\ (\texttt{cons}\ x_2\ l_2)\ l_3$$

Now forward saturating search begins. The database begins containing the initial sequents, and the conclusions of the inference rules with no premises. The database is quickly saturated. The only rule that might be applied repeatedly is R_2. But since we can delete sequents with a subformula $\texttt{rev}\ l_1\ l_2\ l_3$ where l_2 is not a subterm of the goal, we saturate after 2 steps. Note that backward search would fail immediately, since the only rule that unifies with $\texttt{eq}\ c_3\ c_3'$ is EqRefl, and unification fails.

Now the inductive prover chooses to split a variable, in this case the derivation of $\texttt{rev}\ c_1\ c_2\ c_3$. This yields two subgoals. The first has $c_1 = \texttt{nil}$ and $c_2 = c_3$. The prover, noticing that the last hypothesis could have been generated in only one way, finds that $c_2 = c_3'$ and returns the goal

$$\frac{\text{defs}}{\texttt{eq}\ c_3\ c_3}$$

This is solved by the first-order prover immediately. The second goal has $c_1 = \texttt{cons}\ x\ c_1'$, with the last rule being rev-cons. Again, the prover inverts the last hypothesis, yielding the goal

$$\frac{\text{defs},\ \texttt{rev}\ c_1\ (\texttt{cons}\ x\ c_2)\ c_3,\ \texttt{rev}\ c_1\ \texttt{cons}\ x\ c_2\ c_3'}{\texttt{eq}\ c_3'\ c_3}$$

Now the recursion phase tries to find applications of the induction hypothesis. It finds one using the two hypotheses and adds it to the goal

$$\frac{\text{defs},\ \texttt{rev}\ c_1\ (\texttt{cons}\ x\ c_2)\ c_3,\ \texttt{rev}\ c_1\ \texttt{cons}\ x\ c_2\ c_3',\ \texttt{eq}\ c_3\ c_3'}{\texttt{eq}\ c_3'\ c_3}$$

Again, the first order prover solves this instantly. While this is an almost pathologically simple example, it shows the importance of the ability of the first-order prover to fail quickly. Since the second two goals were trivial, the entire proof in this case takes about as long as it takes for the first direct attempt to fail. As we noticed, the correct choice of atom polarity is essential.

A.3.2 Lemma: Reverse Involution Lemma

This proof is similar to the last. We keep the same polarities of the atoms. We add the previous lemma to the set of hypotheses.

[10]Recall that all variables in forward sequents are implicitly quantified outside the sequent. Thus, the variable l in one sequent is distinct from the variable l in another sequent.

$$\text{lem} := \forall l_1\ l_2\ l_3\ l_3'.\ \texttt{rev}\ l_1\ l_2\ l_3 \supset \texttt{rev}\ l_1\ l_2\ l_3' \supset \texttt{eq}\ l_3'\ l_3$$

The initial goal is

$$\frac{\text{defs},\ \text{lem},\ \texttt{rev}\ c_1\ c_2\ c_3,\ \texttt{rev}\ c_3\ \texttt{nil}\ c_3',\ \texttt{rev}\ c_2\ c_1\ c_3''}{\texttt{eq}\ c_3'\ c_3''}$$

The initial filling phase fails, for the same reason as the last proof. The prover then splits on the hypothesis $\texttt{rev}\ c_1\ c_2\ c_3$ yielding subgoals

$$\frac{\text{defs},\ \text{lem},\ \texttt{rev}\ c_2\ \texttt{nil}\ c_3',\ \texttt{rev}\ c_2\ \texttt{nil}\ c_3''}{\texttt{eq}\ c_3'\ c_3''}$$

$$\frac{\text{defs},\ \text{lem},\ \texttt{rev}\ c_1\ (\texttt{cons}\ x\ c_2)\ c_3,\ \texttt{rev}\ c_3\ \texttt{nil}\ c_3',\ \texttt{rev}\ (\texttt{cons}\ x\ c_2)\ c_1\ c_3'}{\texttt{eq}\ c_3'\ c_3''}$$

The first is solved in the filling phase by using the lemma. The second requires the induction phase to generate a new appeal to the induction hypothesis

$$\frac{\text{defs},\ \text{lem},\ \texttt{rev}\ c_1\ (\texttt{cons}\ x\ c_2)\ c_3,\ \texttt{rev}\ c_3\ \texttt{nil}\ c_3',\ \texttt{rev}\ (\texttt{cons}\ x\ c_2)\ c_1\ c_3'', \texttt{eq}\ c_3'\ c_3''}{\texttt{eq}\ c_3'\ c_3''}$$

and the filling is trivial.

A.3.3 Theorem: Reverse is an Involution

The theorem now should follow from the above lemmas. In this case, the first-order theorem prover is able to find a solution during the initial filling phase.[11]

A.4 Twelf Formalization with Theorem Prover

Though we have not yet implemented it, here is a possible concrete syntax for Twelf code using the inductive theorem prover. The is roughly the concrete syntax for the existing meta-theorem prover.

```
%{ Reverse is effective }%

%theorem can-rev
   : forall {L1 : list} {L2 : list}
     exists {L3 : list} {D: rev L1 L2 L3}
     true.
%prove L1 (can-rev L1 _ _ _).

%{ Reverse respects equality }%
```

[11] The existing backward prover does not find the solution, evidence that an inverse method prover may be able to prove more meta-theorems.

```
%theorem rev-eq
   : forall* {L1 : list}{L2 : list}{L3 : list}{L4 : list}{L5 : list}{L6 : list}
     forall {D1 : eq L1 L2} {D2 : eq L3 L4} {D3 : eq L5 L6} {D4 : rev L1 L3 L5}
     exists {D5: rev L2 L4 L6}
     true.
%prove E1 (rev-eq E1 _ _ _ _).

%{ Reverse is deterministic }%

%theorem rev-determ
   : forall* {L1 : list}{L2 : list}{L3 : list}{L4 : list}
     forall {D1 : rev L1 L2 L3} {D2 : rev L1 L2 L4}
     exists {D3: eq L4 L3}
     true.
%prove D1 (rev-determ D1 _ _).

%{ Reverse is an involution }%

%theorem revrev-lem
   : forall* {L1 : list}{L2 : list}{L3 : list}{L4 : list}{L5 : list}
     forall {D1 : rev L1 L2 L3} {D2 : rev L3 nil L4} {D3 : rev L2 L1 L5}
     exists {D4: eq L5 L4}
     true.
%prove D1 (revrev-lem D1 _ _ _).

%theorem revrev
   : forall* {L1 : list}{L2 : list}
     forall {D1 : rev L1 nil L2}
     exists {D2 : rev L2 nil L1}
     true.
%prove D1 (revrev D1 _).
```

Alternatively, we could deduce the theorem form a type family, mode, and worlds[12] declaration.

```
%{ Reverse is effective }%

can-rev : {L1} {L2} rev L1 L2 L3 -> type.
%mode can-rev +L1 +L2 -D.
%worlds () (can-rev _ _ _).
%prove L1 (can-rev L1 _ _ _).

%{ Reverse respects equality }%
rev-eq : eq L1 L1' -> eq L2 L2' -> eq L3 L3'
         -> rev L1 L2 L3 -> rev L1' L2' L3' -> type.
%mode rev-eq +E1 +E2 +E3 +R1 -R2.
%worlds () (rev-eq _ _ _ _ _).
%prove E1 (rev-eq E1 _ _ _ _).
```

[12]Regular worlds don't come up in this example, but see the next examples.

```
%{ Reverse is deterministic }%

rev-determ : rev L1 L2 L3 -> rev L1 L2 L4
             -> eq L4 L3 -> type.
%mode rev-determ +R1 +R2 -E.
%worlds () (rev-determ _ _ _).
%prove D1 (rev-determ D1 _ _).

%{ Reverse is an involution }%

revrev-lem : rev L1 L2 L3 -> rev L3 nil L4
                -> rev L2 L1 L5 -> eq L4 L5 -> type.
%mode revrev-lem +R1 +R2 +R3 -E.
%worlds () (revrev-lem _ _ _ _).

revrev : rev L1 nil L2 -> rev L2 nil L1 -> type.
%mode revrev +R1 -R2.
%worlds () (revrev _ _).
%prove D1 (revrev D1 _).
```

B Example: Extrinsic Typing

This example demonstrates proof search in LF with higher-order terms. We will encode simple extrinsic typing for the untyped lambda calculus, and search for two typing derivations. These definitions will be used in the next section to demonstrate the workings of the theorem prover on an inductive proof of type preservation.

B.1 Informal Definitions

We begin by defining types and terms using LF as the specification language. Because the objects of LF include lambda expressions, we use the LF expression $\texttt{lam}\ (\lambda x.\ e\ x)$ to refer to the encoding of an object-level lambda term. Object level application is written $e_1 \cdot e_2$ to distinguish it from LF application which is denoted by juxtaposition.

$$\frac{x \in \Delta}{\vdash \texttt{tp}\ x} \qquad \frac{\vdash \texttt{tp}\ t_1 \quad \vdash \texttt{tp}\ t_2}{\vdash \texttt{tp}\ (t_1 \Rightarrow t_2)}$$

$$\frac{x \in \Gamma}{\Gamma \vdash \texttt{tm}\ x} \qquad \frac{\Gamma, x \vdash \texttt{tm}\ (e\ x)}{\Gamma \vdash \texttt{tm}\ (\texttt{lam}\ (\lambda x.\ e\ x))} \qquad \frac{\Gamma \vdash \texttt{tm}\ e_1 \quad \Gamma \vdash \texttt{tm}\ e_2}{\Gamma \vdash \texttt{tm}\ (e_1 \cdot e_2)}$$

Extrinsic typing is defined using a context for variables.

$$\frac{\Gamma, \Gamma \vdash \texttt{of}\ x\ t_1 \vdash \Gamma \vdash \texttt{of}\ (e\ x)\ t_2}{\Gamma \vdash \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2)}\ \text{of-lam} \qquad \frac{\Gamma \vdash \Gamma \vdash \texttt{of}\ e_1\ (t_2 \Rightarrow t_1) \quad \Gamma \vdash \Gamma \vdash \texttt{of}\ e_2\ t_2}{\Gamma \vdash \Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t_1}\ \text{of-app}$$

B.2 Twelf Encoding

The Twelf encoding is straightforward, using higher order abstract syntax to represent typing of higher-order expressions.

```
tp : type.

=> : tp -> tp -> tp. %infix right 10 =>.

tm : type.

@ : tm -> tm -> tm. %infix left 10 @.
lam : (tm -> tm) -> tm.

of : tm -> tp -> type.

of/lam : of (lam T) (A => B)
          <- ({x} of x A -> of (T x) B).

of/app : of (T1 @ T2) B
          <- of T1 (A => B)
          <- of T2 A.
```

B.3 LF Theorem Prover

The LF theorem prover discussed in Section 8 is able to search for derivations that witness LF types. Because the operational semantics is not obvious, it is not expected that the inverse method prover will replace the backward-chaining Prolog-style prover for Twelf query declarations. The LF theorem prover is designed to be used in tandem with the inductive $\mathbf{M}_2^+$ prover. Even so, it is interesting to examine the inverse method search procedure for higher-order terms.

B.3.1 Inference Rules

As in the first-order example of Appendix A, we must choose the polarities of the given atoms. The rules for the choices are shown below:

Positive

$$\frac{\Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2) \longrightarrow \gamma \quad \Gamma \vdash \texttt{of}\ x\ t_1 \longrightarrow \Gamma \vdash \texttt{of}\ (e\ x)\ t_2}{\longrightarrow \gamma} \qquad \frac{\Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t_1 \longrightarrow \gamma}{\Gamma \vdash \texttt{of}\ e_1\ (t_2 \Rightarrow t_1), \Gamma \vdash \texttt{of}\ e_2\ t_2 \longrightarrow \gamma}$$

Negative

$$\frac{\Gamma \vdash \texttt{of}\ x\ t_1 \longrightarrow \Gamma \vdash \texttt{of}\ (e\ x)\ t_2}{\longrightarrow \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2)} \qquad \frac{\longrightarrow \Gamma \vdash \texttt{of}\ e_1\ (t_2 \Rightarrow t_1) \quad \longrightarrow \Gamma \vdash \texttt{of}\ e_2\ t_2}{\longrightarrow \Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t_1}$$

Again, typing would be hopeless if assigned negative polarity. The inverse method would generate all typings of all terms, and all evaluations of all terms respectively. Notice however that typing is contractive in its first argument which will restrict the search space to subterms of the goal.

B.3.2 Example: Term Inference

We can use the LF theorem prover to either find the type of a term or to build a term of a given type. As a representative example, we build a term of type $a \Rightarrow b \Rightarrow a$ by solving the query

$$\Pi a_1\ a_2 : \texttt{tp}.\ \Gamma \vdash \texttt{of}\ E\ (a_1 \Rightarrow a_2 \Rightarrow a_1)$$

To keep the example succinct, we only use the rules

$$\frac{\Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2) \longrightarrow \gamma \quad \Gamma \vdash \texttt{of}\ x\ t_1 \longrightarrow \Gamma \vdash \texttt{of}\ (e\ x)\ t_2}{\longrightarrow \gamma}\ R_1 \qquad \frac{\longrightarrow \Gamma \vdash \texttt{of}\ E\ (c_1 \Rightarrow c_2 \Rightarrow c_1)}{\longrightarrow goal}\ R_2$$

The second rule is used to simulate an existential quantifier that does not exist in LF. The actual prover would generate new sequents corresponding to the rule for application. We will use the Imogen variable-rule loop so we can see the unifications that occur at each stage. We begin with a sole initial sequent that subsumes all others.[13]

$$Q_1 : \Gamma \vdash \texttt{of}\ e^1\ t^1 \longrightarrow \Gamma \vdash \texttt{of}\ e^1\ t^1$$

[13]Recall that we rename variables when necessary so we don't erroneously equate variables from different rules and sequents.

Stage 1. We match R_1 with Q_1. This yields unification equations $e^1 = \texttt{lam}\ (\lambda x.\ e\ x)$ and $t^1 = t_1 \Rightarrow t_2$ to generate the new derived rule

$$\frac{\Gamma \vdash \texttt{of}\ x\ t_1 \longrightarrow \Gamma \vdash \texttt{of}\ (e\ x)\ t_2}{\longrightarrow \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2)}\ R_3$$

Stage 2. We can match Q_1 with R_3 in two different ways, one that consumes the antecedent, and the other that doesn't. In the first case we have equations

$$x = e^1 \quad t_1 = t^1 \quad e\ x = e^1 \quad t_2 = t^1$$

yielding new sequent

$$Q_2 : \longrightarrow \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ x))\ (t^2 \Rightarrow t^2)$$

In the second case we have only the last two or the above equations, yielding

$$Q_3 :\ \Gamma \vdash \texttt{of}\ (e^3\ x^3)\ t_2^3 \longrightarrow \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e^3\ x))\ (t_1^3 \Rightarrow t_2^3)$$

Stage 3. Matching Q_3 to R_3 gives equations

$$x = e^3\ x^3 \quad t_1 = t_2^3 \quad e\ x = \texttt{lam}\ (\lambda x.\ e^3\ x) \quad t_2 = t_1^3 \Rightarrow t_2^3$$

and new sequent

$$Q_4 : \longrightarrow \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda y.\ \texttt{lam}\ (\lambda x.\ y))\ (t_2^4 \Rightarrow t_1^4 \Rightarrow t_2^4)$$

An easy application of rule R_2 subsumes the goal.

C Example: Type Preservation

This example illustrates the combination of an inverse method LF theorem prover with the induction strategy of $\mathbf{M}_2^+$. We will assume the existence of an LF theorem prover. We show the steps meta-theorem prover would take in constructing a proof of type preservation with a call-by-name evaluation relation.

C.1 Paper Proof

First define the evaluation relation as

$$\frac{}{\vdash \texttt{eval}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (\texttt{lam}\ (\lambda x.\ e\ x))}\ \text{eval-lam} \qquad \frac{\vdash \texttt{eval}\ e_1\ (\texttt{lam}\ (\lambda x.\ e_3\ x)) \quad \vdash \texttt{eval}\ (e_3\ e_2)\ e}{\vdash \texttt{eval}\ (e_1 \cdot e_2)\ e}\ \text{eval-app}$$

Theorem C.1. *For any e, e' and t, if $\Gamma \vdash \texttt{of}\ e\ t$ and $\vdash \texttt{eval}\ e\ e'$ then $\Gamma \vdash \texttt{of}\ e'\ t$.*

Proof. Let $\mathcal{D}$ be the derivation of $\vdash \texttt{eval}\ e\ e'$ and $\mathcal{E}$ the derivation of $\Gamma \vdash \texttt{of}\ e\ t$. The proof is by induction on $\mathcal{D}$.

Case:

$$\mathcal{D} = \frac{}{\vdash \texttt{eval}\ (\texttt{lam}\ (\lambda x.\ e_0\ x))\ (\texttt{lam}\ (\lambda x.\ e_0\ x))}\ \text{eval-lam}$$

Then $e = e' = \texttt{lam}\ (\lambda x.\ e_0\ x)$, so $\mathcal{E}$ is a derivation of $\Gamma \vdash \texttt{of}\ e'\ t$.

Case:

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdash \texttt{eval}\ e_1\ (\texttt{lam}\ (\lambda x.\ e_3\ x)) & \vdash \texttt{eval}\ (e_3\ e_2)\ e'\end{array}}{\vdash \texttt{eval}\ (e_1 \cdot e_2)\ e'}\ \text{eval-app}$$

$$\mathcal{E} = \frac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \vdash \Gamma \vdash \texttt{of}\ e_1\ (t_2 \Rightarrow t) & \vdash \Gamma \vdash \texttt{of}\ e_2\ t_2\end{array}}{\vdash \Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t}\ \text{of-app}$$

1. $e = e_1 \cdot e_2$	Assumption
2. $\Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e_3\ x))\ (t_2 \Rightarrow t)$	Induction with $\mathcal{D}_1, \mathcal{E}_1$
3. For any x, if $\Gamma \vdash \texttt{of}\ x\ t_2$ then $\Gamma \vdash \texttt{of}\ e_3\ x\ t$	Inversion on 2.
4. $\Gamma \vdash \texttt{of}\ (e_3\ e_2)\ t$	(3) with $\mathcal{E}_2$
5. $\Gamma \vdash \texttt{of}\ e'\ t$	Induction with (4) and $\mathcal{D}_2$

□

C.2 Twelf Encoding

Evaluation.

```
eval : tm -> tm -> type.

eval/lam : eval (lam T) (lam T).
eval/app : eval (T1 @ T2) T4
            <- eval T1 (lam T3)
            <- eval (T3 T2) T4.
```

Type Preservation. Type preservation requires a block declaration. The world can be extended by a variable `x` of type `tm` along with a hypothetical derivation `px` : $\Gamma \vdash$ `of` x `A` for some type `A`.

```
pres : of T A -> eval T T' -> of T' A -> type.
%mode pres +O1 +E -O2.

- : pres (of/lam D) eval/lam (of/lam D).
- : pres
     (of/app
        (O2 : of T2 A)
        (O1 : of T1 (A => B)))
     (eval/app
        (E2 : eval (T3 T2) T4)
        (E1 : eval T1 (lam T3)))
     (O3 : of T4 B)
     <- pres O1 E1 (of/lam (O1' : {x} (of x A -> of (T3 x) B)))
     <- pres (O1' T2 O2) E2 O3.

%block b : some {A : tp} block {x : tm} {px : of x A}.
%worlds (b) (pres _ _ _).
%total E (pres _ E _).
```

C.3 M_2^+ Theorem prover

As in the list involution example, the goal of the meta-theorem prover is to discover a proof of type preservation automatically. The LF derived rules for the two polarities are

Positive

$$\frac{\vdash \texttt{eval}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (\texttt{lam}\ (\lambda x.\ e\ x)) \longrightarrow \gamma}{\longrightarrow \gamma} \qquad \frac{\vdash \texttt{eval}\ (e_1 \cdot e_2)\ e \longrightarrow \gamma}{\vdash \texttt{eval}\ e_1\ (\texttt{lam}\ (\lambda x.\ e_3\ x)), \vdash \texttt{eval}\ (e_3\ e_2)\ e \longrightarrow \gamma}$$

Negative

$$\frac{}{\longrightarrow \vdash \texttt{eval}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (\texttt{lam}\ (\lambda x.\ e\ x))} \qquad \frac{\longrightarrow \vdash \texttt{eval}\ e_1\ (\texttt{lam}\ (\lambda x.\ e_3\ x)) \quad \longrightarrow \vdash \texttt{eval}\ (e_3\ e_2)\ e}{\longrightarrow \vdash \texttt{eval}\ (e_1 \cdot e_2)\ e}$$

Once again, evaluation should be positive. Note that unlike typing which is expansive in its first argument, neither argument of `eval` is well behaved. We thus begin any filling phase with the following inference rules:

$$\frac{\Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2) \longrightarrow \gamma \quad \Gamma \vdash \texttt{of}\ x\ t_1 \longrightarrow \Gamma \vdash \texttt{of}\ (e\ x)\ t_2}{\longrightarrow \gamma}$$

$$\frac{\Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t_1 \longrightarrow \gamma}{\Gamma \vdash \texttt{of}\ e_1\ (t_2 \Rightarrow t_1), \Gamma \vdash \texttt{of}\ e_2\ t_2 \longrightarrow \gamma}$$

$$\frac{\vdash \texttt{eval}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (\texttt{lam}\ (\lambda x.\ e\ x)) \longrightarrow \gamma}{\longrightarrow \gamma}$$

$$\frac{\vdash \texttt{eval}\ (e_1 \cdot e_2)\ e \longrightarrow \gamma}{\vdash \texttt{eval}\ e_1\ (\texttt{lam}\ (\lambda x.\ e_3\ x)), \vdash \texttt{eval}\ (e_3\ e_2)\ e \longrightarrow \gamma}$$

We begin by asking the $\mathbf{M}_2^+$ prover for a proof of

$$\forall e\ e' : \texttt{tm}.\ \forall t : \texttt{tp}.\ \forall \mathcal{D} : \Gamma \vdash \texttt{of}\ e\ t.\ \forall \mathcal{E} :\vdash \texttt{eval}\ e\ e'.\ \exists \mathcal{D}' : \Gamma \vdash \texttt{of}\ e'\ t.\ \top$$

After applying the rules Π-R and →-R, we have the following goal, where $\mathbf{e}, \mathbf{e}'$ and $\mathbf{t}$ are new constants.

$$\mathbf{e} : \texttt{tm},\ \mathbf{e}' : \texttt{tm},\ \mathbf{t} : \texttt{tp},\ \mathcal{D} : \Gamma \vdash \texttt{of}\ \mathbf{e}\ \mathbf{t},\ \mathcal{E} :\vdash \texttt{eval}\ \mathbf{e}\ \mathbf{e}' \quad \exists \mathcal{D}' : \Gamma \vdash \texttt{of}\ \mathbf{e}'\ \mathbf{t}.\ \top$$

Phase 1: Filling. We first attempt a direct proof using the LF theorem prover. After inspecting the rules and goal, we find the following initial sequents, where parameters are written in bold face.

$$\begin{gathered}
\vdash \texttt{eval}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (\texttt{lam}\ (\lambda x.\ e\ x)) \longrightarrow \vdash \texttt{eval}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (\texttt{lam}\ (\lambda x.\ e\ x)) \\
\vdash \texttt{eval}\ (e_1 \cdot e_2)\ e \longrightarrow \vdash \texttt{eval}\ (e_1 \cdot e_2)\ e \\
\vdash \texttt{eval}\ \mathbf{e}\ \mathbf{e}' \longrightarrow \vdash \texttt{eval}\ \mathbf{e}\ \mathbf{e}' \\
\Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2) \longrightarrow \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ (t_1 \Rightarrow t_2) \\
\Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t \longrightarrow \Gamma \vdash \texttt{of}\ (e_1 \cdot e_2)\ t \\
\Gamma \vdash \texttt{of}\ \mathbf{e}\ \mathbf{t} \longrightarrow \Gamma \vdash \texttt{of}\ \mathbf{e}\ \mathbf{t}
\end{gathered}$$

As we have noted, `of` is expansive in its first argument, so we only need keep those sequents where the first argument to `of` is a subterm either of $\mathbf{e}$ or $\mathbf{e}'$. Since none of the initial sequents involving `of` satisfies this requirement, none are inserted in the initial database. A cursory glance at the rules shows that the goal can never be obtained without at least one of those axioms, so the filling phase fails before attempting any inference.

Phase 1: Splitting. We then choose a variable on which to split. We assume we make the fortuitous choice of splitting on $\mathcal{E}$. As in the paper proof, this choice leaves us with two subgoals. The first after equational reasoning is

$$\mathbf{e}_1 : \texttt{tm} \Rightarrow \texttt{tm},\ \mathbf{t} : \texttt{tp},\ \mathcal{D} : \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ \mathbf{e}_1\ x))\ \mathbf{t} \quad \exists \mathcal{D}' : \Gamma \vdash \texttt{of}\ (\texttt{lam}\ (\lambda x.\ \mathbf{e}_1\ x))\ \mathbf{t}.\ \top$$

This is trivially solved by the following filling phase. The second is

$$\frac{\begin{array}{c}\mathbf{e}_1 : \mathtt{tm},\ \mathbf{e}_2 : \mathtt{tm},\ \mathbf{e}' : \mathtt{tm},\ \mathbf{e}_3 : \mathtt{tm} \Rightarrow \mathtt{tm},\\ \mathcal{E}_1 :\vdash \mathtt{eval}\ \mathbf{e}_1\ (\mathtt{lam}\ (\lambda x.\ \mathbf{e}_3\ x)),\ \mathcal{E}_2 :\vdash \mathtt{eval}\ (\mathbf{e}_3\ \mathbf{e}_2)\ \mathbf{e}',\ \mathcal{D} : \Gamma \vdash \mathtt{of}\ (\mathtt{lam}\ (\lambda x.\ \mathbf{e}_1\ x))\ \mathbf{t}\end{array}}{\exists \mathcal{D}' : \Gamma \vdash \mathtt{of}\ \mathbf{e}'\ \mathbf{t}.\ \top}$$

Noticing that $\mathcal{D}$ can only be constructed in one way, it inverts the derivation, yielding goal

$$\frac{\begin{array}{c}\mathbf{e}_1 : \mathtt{tm},\ \mathbf{e}_2 : \mathtt{tm},\ \mathbf{e}' : \mathtt{tm},\ \mathbf{e}_3 : \mathtt{tm} \Rightarrow \mathtt{tm},\\ \mathcal{E}_1 :\vdash \mathtt{eval}\ \mathbf{e}_1\ (\mathtt{lam}\ (\lambda x.\ \mathbf{e}_3\ x)),\ \mathcal{E}_2 :\vdash \mathtt{eval}\ (\mathbf{e}_3\ \mathbf{e}_2)\ \mathbf{e},\ \mathcal{D}_1 : \Gamma \vdash \mathtt{of}\ \mathbf{e}_1\ (\mathbf{t}_2 \Rightarrow \mathbf{t}_1),\ \mathcal{D}_2 : \Gamma \vdash \mathtt{of}\ \mathbf{e}_2\ \mathbf{t}_2\end{array}}{\exists \mathcal{D}' : \Gamma \vdash \mathtt{of}\ \mathbf{e}'\ \mathbf{t}.\ \top}$$

Phase 1: Recursion. Now the theorem prover tries to find all possible applications of the induction hypotheses. There is one immediate application with $\mathcal{E}_1$ and $\mathcal{D}_1$, extending the proof context with

$$\mathcal{D}_3 : \Gamma \vdash \mathtt{of}\ (\mathtt{lam}\ (\lambda x.\ \mathbf{e}_3\ x))\ (\mathbf{t}_2 \Rightarrow \mathbf{t})$$

and by inversion with

$$\mathcal{D}_3' : \Pi x : \mathtt{tm}.\ \Gamma \vdash \mathtt{of}\ x\ \mathbf{t}_2 \rightarrow \Gamma \vdash \mathtt{of}\ (\mathbf{e}_3\ x)\ \mathbf{t}$$

We also can apply the induction hypothesis to $\mathcal{E}_2$, giving

$$\mathcal{E}_2' : \forall (p : \Gamma \vdash \mathtt{of}\ \mathbf{e}_3\ \mathbf{e}_2\ \mathbf{t}).\ \exists (q : \Gamma \vdash \mathtt{of}\ \mathbf{e}'\ \mathbf{t}).\ \top.$$

Using Schürmann's technique of Skolemization [Schürmann, 2000], we can Skolemize $\mathcal{E}_2'$ to

$$\Gamma \vdash \mathtt{of}\ \mathbf{e}_3\ \mathbf{e}_2\ \mathbf{t} \rightarrow \Gamma \vdash \mathtt{of}\ \mathbf{e}'\ \mathbf{t}$$

and the next filling stage will find the solution. Since we haven't yet implemented this process, we have been purposely vague about how the recursion stage operates. It seems a saturation based strategy like the inverse method can be used for this phase, as well as for the LF theorem prover. Perhaps the recursion phase can be implemented using the LF theorem prover using additional rules corresponding to applications of the induction hypotheses.

C.4 Twelf Formalization with Theorem Prover

As in Appendix A, we give a possible syntax for the theorem declaration as it would be given in a Twelf file.

```
%theorem pres : forallG (some {A : tp} pi {x : tm} {px : of x A})
                forall* {A : tp} {T : tm} {T' : tm}
                forall {O : of T A} {E : eval T T'}
                exists {O' : of T' A}
                true.
%prove E (pres _ E _).
```

Inferring the theorem from the type family seems to be possible as well, and would interact better with the current Twelf methodology:

```
pres : of T A -> eval T T' -> of T' A -> type.
%mode pres +O1 +E -O2.
%block b : some {A : tp} block {x : tm} {px : of x A}.
%worlds (b) (pres _ _ _).
%prove E (pres _ E _).
```

References

[Abate and Goré, 2003] Abate, P. and Goré, R. (2003). The tableaux work bench. In *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 230–236. Springer-Verlag. 57

[Andreoli, 1992] Andreoli, J.-M. (1992). Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347. 9, 24, 25

[Andreoli, 2001] Andreoli, J.-M. (2001). Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1–3):131–163. 27

[Appel and Felten, 1999] Appel and Felten (1999). Proof-carrying authentication. In *SIGSAC: 6th ACM Conference on Computer and Communications Security*. ACM SIGSAC. 6

[Appel, 2001] Appel, A. (2001). Foundational Proof-Carrying Code. In *Logic in Computer Science*, pages 247–258. IEEE Computer Society. 6

[Autexier et al., 1999] Autexier, S., Hutter, D., Mantel, H., and Schairer, A. (1999). System description: inka 5.0 : A logic voyager. In Ganzinger, H., editor, *Conference on Automated Deduction*, pages 207–211. Springer. 57

[Avellone et al., 2004] Avellone, A., Fiorino, G., and Moscato, U. (2004). A new O(n log n)-space decision procedure for propositional intuitionistic logic. In *Kurt Goedel Society Collegium Logicum*, volume VIII, pages 17–33. 32

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press. 35

[Bachmair and Ganzinger, 2001] Bachmair, L. and Ganzinger, H. (2001). Resolution theorem proving. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science. 12

[Baelde, 2008] Baelde, D. (2008). *A Linear Approach to the Proof Theory of Least and Greatest Fixed Points*. PhD thesis, Ećole Polytechnique. 25, 48, 50, 57

[Baelde et al., 2007] Baelde, D., Gacek, A., Miller, D., Nadathur, G., and Tiu, A. (2007). The Bedwyr system for model checking over syntactic expressions. In Pfenning, F., editor, *Conference on Automated Deduction*, pages 391–397. Springer. 57

[Bancilhon et al., 1986] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. (1986). Magic sets and other strange ways to implement logic programs. In *ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*. 42

[Barendregt, 1984] Barendregt, H. (1984). *The Lambda Calculus, its Syntax and Semantics*. North-Holland. Second Revised Edition. 37

[Beckert and Goré, 1997] Beckert, B. and Goré, R. (1997). Free variable Tableaux for propositional modal logics. In Galmiche, D., editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 91–106. Springer. 58

[Beeri and Ramakrishnan, 1991] Beeri, C. and Ramakrishnan, R. (1991). On the power of magic. *The Journal of Logic Programming*, 10:255–300. 42

[Bundy, 2001] Bundy, A. (2001). The automation of proof by mathematical induction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science. 57

[Bundy et al., 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253. 57

[Bundy et al., 1990] Bundy, A., van Harmelen, F., Smaill, A., and Ireland, A. (1990). Extensions to the rippling-out tactic for guiding inductive proofs. In Stickel, M. E., editor, *Conference on Automated Deduction*, pages 132–146. Springer. 57

[Chaudhuri, 2006] Chaudhuri, K. (2006). *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University. Technical report CMU-CS-06-162. 9, 36, 37, 56

[Chaudhuri et al., 2008] Chaudhuri, K., Pfenning, F., and Price, G. (2008). A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning*, 40(2–3):133–177. 25, 39, 52, 56

[Comon and Lescanne, 1989] Comon, H. and Lescanne, P. (1989). Equational problems and disunification. *Journal of Symbolic Computation*, 3–4(7):371–426. 44

[Coquand, 1998] Coquand, C. (1998). The AGDA proof system homepage. http://www.cs.chalmers.se/~catarina/agda/. 58

[Degtyarev and Voronkov, 2001a] Degtyarev, A. and Voronkov, A. (2001a). Equality reasoning in sequent-based calculi. In Robinson, J. A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, pages 611–706. Elsevier and MIT Press. 47, 56

[Degtyarev and Voronkov, 2001b] Degtyarev, A. and Voronkov, A. (2001b). The inverse method. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 4, pages 179–272. Elsevier Science. 9, 12, 16, 18, 27, 35, 56, 58

[Dyckhoff, 1992] Dyckhoff, R. (1992). Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807. 32

[Gacek, 2008] Gacek, A. (2008). The Abella interactive theorem prover (system description). In Armando, A., Baumgartner, P., and Dowek, G., editors, *International Joint Conference on Automated Reasoning*, pages 154–161. Springer. 58

[Garg and Pfenning, 2006] Garg, D. and Pfenning, F. (2006). Non-interference in constructive authorization logic. In Guttman, J., editor, *Computer Security Foundations Workshop*, pages 283–293. IEEE Computer Society Press. 44

[Gentzen, 1934] Gentzen, G. (1934). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210 and 405–431. Translated in *The Collected Papers of Gerhard Gentzen* [Gentzen, 1969]. 12

[Gentzen, 1969] Gentzen, G. (1969). *The Collected Papers of Gerhard Gentzen*. North Holland. Edited by M. E. Szabo. 79

[Giese, 2001] Giese, M. (2001). Incremental Closure of Free Variable Tableaux. In *International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science, pages 545–560. Springer-Verlag. 57

[Goldfarb, 1981] Goldfarb, W. D. (1981). The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230. 11

[Gottlob and Leitsch, 1985] Gottlob, G. and Leitsch, A. (1985). On the efficiency of subsumbtion algorithms. *Journal of the ACM*, 32(2). 57

[Graf, 1996] Graf, P. (1996). *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer. 19, 57

[Hähnle, 2001] Hähnle, R. (2001). Tableaux and related methods. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 3, pages 100–178. Elsevier Science. 12

[Harper et al., 1993] Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM*, 40(1):143–184. 6, 51

[Harper and Licata, 2007] Harper, R. and Licata, D. R. (2007). Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673. 6, 51

[Howe, 1998] Howe, J. M. (1998). *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St. Andrews, Scotland. 25

[Hustadt and Schmidt, 2000] Hustadt, U. and Schmidt, R. A. (2000). MSPASS: Modal reasoning by translation and first-order resolution. In Dyckhoff, R., editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 67–71. Springer. 58

[Hutter and Sengler, 1996] Hutter, D. and Sengler, C. (1996). INKA: the next generation. In McRobbie, M. A. and Slaney, J. K., editors, *Conference on Automated Deduction*, pages 288–292. Springer. 57

[Jaffar and Lassez, 1987] Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, West Germany. 56

[Jia, 2008] Jia, L. (2008). *Linear Logic and Imperative Programming*. PhD thesis, Princeton University. 56

[Kaufmann and Moore, 1997] Kaufmann, M. and Moore, J. S. (1997). An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213. 57

[Kreitz and Pientka, 2001] Kreitz, C. and Pientka, B. (2001). Connection-driven inductive theorem proving. *Studia Logica*, 69(2):293–326. 57

[Lamarche, 1995] Lamarche, F. (1995). Games semantics for full propositional linear logic. In *Logic in Computer Science*, pages 464–473. 9, 24

[Larchey-Wendling et al., 2001] Larchey-Wendling, D., Méry, D., and Galmiche, D. (2001). STRIP: Structural sharing for efficient proof-search. In Goré, R., Leitsch, A., and Nipkow, T., editors, *International Joint Conference on Automated Reasoning*, pages 696–700. Springer. 32

[Lassez and McAloon, 1990] Lassez, J.-L. and McAloon, K. (1990). A constraint sequent calculus. In *LICS'90: Proceedings 5th IEEE Symposium on Logic in Computer Science*, pages 52–61. IEEE Computer Society Press. 57

[Lee et al., 2007] Lee, D. K., Crary, K., and Harper, R. (2007). Towards a mechanized metatheory of standard ML. In Hofmann, M. and Felleisen, M., editors, *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184. ACM. 6

[Liang and Miller, 2007] Liang, C. and Miller, D. (2007). Focusing and polarization in intuitionistic logic. In Duparc, J. and Henzinger, T. A., editors, *Computer Science Logic*, pages 451–465. Springer. 9, 25, 27

[Maslov, 1964] Maslov, S. Y. (1964). An inverse method for establishing deducibility in classical predicate calculus. *Doklady Akademii nauk SSSR*, 159:17–20. 9, 12

[McLaughlin and Pfenning, 2008] McLaughlin, S. and Pfenning, F. (2008). Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In Cervesato, I., Veith, H., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*, pages 174–181. Springer. 9, 25

[McLaughlin and Pfenning, 2009] McLaughlin, S. and Pfenning, F. (2009). Efficient intuitionistic theorem proving with the polarized inverse method. In *CADE (submitted)*. 9

[Miller, 1992] Miller, D. (1992). Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358. 11

[Mints, 1993] Mints, G. (1993). Resolution calculus for the first order linear logic. *Journal of Logic, Language, and Information*, 2(1):59–83. 12

[Nanevski et al., 2008] Nanevski, A., Pfenning, F., and Pientka, B. (2008). A contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–56. 52

[Necula, 1997] Necula, G. C. (1997). Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press. 6

[Nieuwenhuis et al., 2001] Nieuwenhuis, R., Hillenbrand, T., Riazanov, A., and Voronkov, A. (2001). On the evaluation of indexing techniques for theorem proving. In Goré, R., Leitsch, A., and Nipkow, T., editors, *International Joint Conference on Automated Reasoning*, pages 257–271. Springer. 57

[Pfenning, 1989] Pfenning, F. (1989). Elf: A language for logic definition and verified meta-programming. In *Logic in Computer Science*, pages 313–322. IEEE Computer Society Press. 6, 51

[Pfenning, 2001] Pfenning, F. (2001). Logical frameworks. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume II, chapter 17, pages 1063–1147. Elsevier Science. 51

[Pfenning and Elliott, 1988] Pfenning, F. and Elliott, C. (1988). Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press. 6

[Pfenning and Reed, 2009] Pfenning, F. and Reed, J. (2009). Embeddings of focused substructural logics. In *To appear*. 44

[Pfenning and Schürmann, 1999] Pfenning, F. and Schürmann, C. (1999). System description: Twelf : A meta-logical framework for deductive systems. In Ganzinger, H., editor, *Conference on Automated Deduction*, pages 202–206. Springer-Verlag. 6

[Pientka, 2003] Pientka, B. (2003). *Tabled higher-order logic programming*. PhD thesis, Carnegie Mellon University. CMU-CS-03-185. 51, 52, 57

[Pientka, 2008] Pientka, B. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–382. ACM. 55, 58

[Pientka et al., 2007] Pientka, B., Li, D. X., and Pompigne, F. (2007). Focusing the inverse method for LF: a preliminary report. In *International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*. 51, 52

[Poswolsky, 2008] Poswolsky, A. (2008). *Functional Programming with Logical Frameworks: The Delphin Project*. PhD thesis, Yale University. 54, 55, 58

[Ramakrishnan et al., 2001] Ramakrishnan, I. V., Sekar, R., and Voronkov, A. (2001). Term indexing. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science. 18, 19, 57

[Raths and Otten,] Raths, T. and Otten, J. The ILTP Library. http://www.iltp.de. 32, 39

[Raths et al., 2007] Raths, T., Otten, J., and Kreitz, C. (2007). The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1-3):261–271. 10, 32

[Riazanov and Voronkov, 1999] Riazanov, A. and Voronkov, A. (1999). Vampire. In Ganzinger, H., editor, *Conference on Automated Deduction*, pages 292–296. Springer. 57

[Riazanov and Voronkov, 2003] Riazanov, A. and Voronkov, A. (2003). Efficient instance retrieval with standard and relational path indexing. In Baader, F., editor, *Conference on Automated Deduction*, pages 380–396. Springer. 57

[Riazanov and Voronkov, 2004] Riazanov, A. and Voronkov, A. (2004). Efficient checking of term ordering constraints. In Basin, D. A. and Rusinowitch, M., editors, *International Joint Conference on Automated Reasoning*, pages 60–74. Springer. 57

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41. 12

[Rümmer, 2008] Rümmer, P. (2008). A constraint sequent calculus for first-order logic with linear integer arithmetic. In Cervesato, I., Veith, H., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 274–289. Springer. 57

[Sahlin et al., 1992] Sahlin, D., Franzén, T., and Haridi, S. (1992). An intuitionistic predicate logic theorem prover. *Journal of Logic and Computation*, 2(5):619–656. 32

[Saranli and Pfenning, 2007] Saranli, U. and Pfenning, F. (2007). Using constrained intuitionistic linear logic for hybrid robotic planning problems. In *Proceedings of the International Conference on Robotics and Automation*, pages 3705–3710. IEEE Computer Society Press. 56

[Schürmann, 2000] Schürmann, C. (2000). *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University. Technical report CMU-CS-00-146. 6, 11, 48, 52, 54, 57, 76

[Tammet, 1996] Tammet, T. (1996). A resolution theorem prover for intuitionistic logic. In McRobbie, M. A. and Slaney, J. K., editors, *Conference on Automated Deduction*, pages 2–16. Springer. 56

[Tammet, 1998] Tammet, T. (1998). Towards efficient subsumption. In Kirchner, C. and Kirchner, H., editors, *Conference on Automated Deduction*, pages 427–441. Springer. 57

[Troelstra and Schwichtenberg, 1996] Troelstra, A. S. and Schwichtenberg, H. (1996). *Basic Proof Theory*. Cambridge University Press. 12

[Voronkov, 1992] Voronkov, A. (1992). Theorem proving in non-standard logics based on the inverse method. In Kapur, D., editor, *Conference on Automated Deduction*, pages 648–662. Springer. 56

[Voronkov, 1995] Voronkov, A. (1995). The anatomy of vampire implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265. 42

[Voronkov, 1999] Voronkov, A. (1999). KΞ: A theorem prover for K. In Ganzinger, H., editor, *Conference on Automated Deduction*, pages 383–387. Springer. 56, 58

[Voronkov, 2000] Voronkov, A. (2000). How to optimize proof-search in modal logics: A new way of proving redundancy criteria for sequent calculi. In *Logic in Computer Science*, pages 401–412. IEEE. 56

[Voronkov, 2001a] Voronkov, A. (2001a). Algorithms, datastructures, and other issues in efficient automated deduction. In Goré, R., Leitsch, A., and Nipkow, T., editors, *International Joint Conference on Automated Reasoning*, pages 13–28. Springer. 57

[Voronkov, 2001b] Voronkov, A. (2001b). How to optimize proof-search in modal logics: new methods of proving redundancy criteria for sequent calculi. *ACM Transactions on Computational Logic*, 2(2):182–215. 56, 58

[Wallen, 1990] Wallen, L. A. (1990). *Automated Deduction in Non-Classical Logics*. MIT Press. 56

[Zeilberger, 2008] Zeilberger, N. (2008). Focusing and higher-order abstract syntax. In Necula, G. C. and Wadler, P., editors, *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–369. ACM. 25, 58