Focused Inverse Method for LF

Xi Li

This thesis is submitted for the partial fulfillment for the degree of Master of Science in Computer Science

School of Computer Science

McGill University

August 2007

ACKNOWLEDGEMENTS

First of all, on all accounts, I would like to thank my advisor Professor Brigitte Pientka for her limitless help and patience, without her, this thesis would not have been possible. At the beginning, it was her intriguing classes that sparked my interest in automated theorem proving. Her ideas and insight on the subject provided the underlying principles of this thesis. Of the many things I learned from her, the most valuable was the pursuit of perfection in her approach to research and life. I am especially appreciative of her critical perspective and endless amount of guidance and support she provided. Secondly, I would like to express my gratitude to Florent Pompigne for his help on the theoretical aspects of the thesis. Also, I would like to thank my other colleagues in the Software Evolution and Verification Lab who have given me many ideas and help: Samuli Heilala, Punit Agrawal and Issac Yuen. Thirdly I would like to thank the faculty of the School of Computer Science at McGill University for the support. In particular, I would like to express my appreciation to Professors Prakash Panangaden and Xiaowen Chang for their guidance and inspiration. Last but certainly not least, I would like to thank my parents and my girlfriend Carolyn Shen for their encouragement and patience throughout the process.

ABSTRACT

Logical frameworks allow us to specify formal systems and prove properties about them. One interesting logical framework is Twelf, a language that uses higher order abstract syntax to encode object languages into the meta language. Currently, uniform proofs have been used for describing proof search in backwards logic programming style. However, there are certain limitations to a backward system, for example, loop-detection mechanisms are required for some of the simplest problems to yield a solution. As a consequence, the search for a more effective proof search algorithm prevails and a forward system is proposed. This thesis will discuss the theoretical foundations for a forward uniform sequent calculus and the implementation of an inverse method prover for Twelf.

ABRÉGÉ

Les cadres logiques nous permettent de spécifier des systèmes formels et de prouver des propriétés à leur sujet. Un cadre logique intéressant est Twelf, un langage qui emploie la syntaxe abstraite d'ordre supérieur pour encoder des langages objet dans le méta-langage. Actuellement, nous employons des preuves uniformes pour décrire la recherche dans le style de programmation logique arrière. Cependant, il y a certaines limitations à un système arrière: des mécanismes de détection de boucle sont nécessaires pour trouver une solution à certains des problèmes les plus simples. Par conséquent, la recherche d'un algorithme plus efficace de recherche de preuve règne et un système vers l'avant est proposé. Cette thèse discutera les bases théoriques d'un calcul séquent uniforme vers l'avant et l'implantation d'un prouveur à méthode inverse pour Twelf.

Contents

Chapter 1

Introduction

A logical framework is a system used to specify logical systems, implement algorithms for object level languages and prove meta-theoretic properties about object languages. The use of logical systems is very important. With the rapid growth of the internet and the ubiquity of computer software, information is the key to success. However, from the myriad of websites on the world wide web, users are still hesitant to download and install software. One important reason for this phenomenon is the general lack of trust in the software distributed. Be the reasons malicious or just software malpractice, unsafe software can greatly cripple the user's ability to perform the tasks as intended. Using either antivirus software or safety policy verifying software, what users want is proof that certain software is safe. Proof-carrying code [12] or certified code[4] can be used to provide users with confidence that certain software is safe. Each piece of software can be specified with a safety policy before it was written. Along with the software users receive, we attach a certificate or proof term of the given safety policy encoded in some logical system. The users now only need to check the proof term to ensure safety of the program. Although proof checking plays an important role in providing safe code, the specified logical systems need to be correct or safe before any argument of safety can be made. For example, we can use methods based on operational semantics to ensure type safety of the given code. Thus, we can specify these logical systems in a logical framework and search over the systems to provide the certificate.

While proof search in logical frameworks is often viewed as logic programming, it is usually considered in a backward setting. We first try to unify the query with the head of a clause then try to satisfy the subgoals in the order they were specified. Prolog, for example, uses SLD-resolution as the algorithm. The theoretical relation between logic programming and proof search is characterized by a system called uniform proofs [8]. The uniform proof system is used for many higher order logic programming systems such as λ-Prolog[10], Twelf[14] and Isabelle[13]. The specification of these systems is based on a backward operational semantics for proof search. We differentiate between a normal phase, where we eagerly apply the invertible rules, and a focus phase, where we pick a rule and focus on it. While this provides a framework to specify and implement certain logics, the backward proof search paradigm has limitations with proof search such as looping issues in some logical systems. We use logic programming terminology such that the head of a clause is the goal of the clause that we can deduce if all subgoals of the clause are satisfied. In backward search, search is generally started with the head and try to satisfy each subgoal of the clause. Since backward search does not usually comprise of a loop detector, it is quite possible that one of the subgoals we are trying to solve is the same as the head. If we apply the same rule once more we are stuck in the situation where we cannot progress.

A tabled proof search engine[15] for Twelf has been proposed to solve problems of this type. The idea is to keep a table whose entries are previously encountered subgoals. In memoizing subcomputations, we eliminate looping behavior and reduce redundant computation. A critical disadvantage in using tabling is the overhead to store and keep track of the subgoals. In addition, computation is suspended and states of computation has to be stored. As the number of subgoals we encounter increase, the table also increases. Thus, to suspend the computation, check the table then resume the computation can take a significant amount of time. Instead of starting with a goal and satisfying its subgoals in a backward fashion, we explore logic programming in the forward direction.

Some of the best performing theorem provers are based on the forward search paradigm. Gandalf[19], for example, is a resolution-based theorem prover that outperformed many others. Inverse method provers, however, are not so prevalent. Degtyarev and Voronkov[5] provided a detailed description of the inverse method that outlined many of the important ideas such as the subformula property and saturation algorithms. One successful inverse method prover is a focused inverse method prover for linear logic by Kaustuv Chaudhuri[2]. Later, Chaudhuri, Pfenning and Price characterized forward and backward chaining in the inverse method[3]. Building on these ideas, this thesis presents a theoretical description and implementation of a forward inverse method prover for Twelf.

For this thesis,

1. We associate logic programming with proof search via uniform proofs. We then describe the syntax of dependently typed lambda calculus to represent LF in canonical form. Then based on the uniform proof system, we develop the theoretical foundations for a forward uniform proof system. In this new system, instead of starting with the head goal and satisfying each subgoal, we turn to another approach. The subformula property allows us to consider only subformulas of a system for proof search. We exploit this property and construct proofs top down starting from axioms. Also, the forward proof system is sound and complete with respect to backward uniform proofs, this means that we can prove a statement true in the forward system if and only if we can prove it in the backward setting.

2. Based on backchaining methods introduced by Chaudhuri, Pfenning and Price for linear logic[3], we implemented a focused inverse method prover for Twelf. Subsumption checking, memoizing subgoals and freezing computation are now unnecessary. For this implementation, we

begin with subformula computation, axiom generation, rule compilation and then describe two looping structures, activation of rules and activation of facts.

3. Further, we give experimental results comparing the inverse method with different looping structures against each other and against the tabling engine. Also, we give an explanation of the numbers and briefly describe further work that could be done to improve this prover.

The rest of the thesis is structured as follows: in Chapter 2, we discuss higher order abstract syntax, contextual modal types and the syntax of LF to give the reader familiarity with the background setting and then we give an example in LF to illustrate the disadvantages of backward search. Chapter 3 gives the theoretical foundations of the inverse method prover including a brief description of Horn clauses and hereditary Harrop formulas, uniform proofs, inverse method with focusing and backchaining. Chapter 4 gives a description of the implementation issues for the inverse method prover. Chapter 5 will provide experimental results and explanation for the data. Finally, we finish with future work and a conclusion. Some results of this thesis have been published in the workshop paper for LFMTP[17].

Chapter 2

Background

2.1 Logical Frameworks

Deductive systems are a useful tool in mathematics and computer science. They are used to specify or prove theories about logical systems. These systems, such as operational semantics or type systems for various logics are usually defined by judgments and inference rules. The logical framework LF was first introduced by Harper, Honsell and Plotkin[6] as a framework for defining logics. LF gives not only a uniform way of encoding a logical language but also provide the capacity of representing inference rules and proofs about the logic.

2.1.1 Higher order abstract syntax

The first step to encoding an object language is to determine the representation of its expressions. While accurate translation of the object language is desired, we do not wish to encode in a low level language where we worry about lexical analysis and parsing related issues. In essence, we would like to represent the object language via abstract syntax so we can concentrate more on the higher level properties of the system. The encoding technique of choice is *higher order abstract syntax*. Higher order abstract syntax allows us to represent object level variables by variables in the meta language. Further, we represent object level binders via λ-abstraction in the meta-language. This way, expressions with renamed bound variables in the object language are equivalent to α-equivalent expressions in the meta-language. Object level substitution is mirrored by substitution in the meta-language.

2.1.2 Dependently typed fragment of LF

LF uses representation types to uniformly encode an object language. Our dependently typed lambda-calculus formulation for LF differentiates three levels of terms: objects, types or type families and kinds. Objects are used to represent entities in the formal language; type families *classify* objects by their syntactic or operational meaning and kinds classify types. Objects provided by the logical framework LF include constants, variables, lambda abstraction and application. Here we characterize logical frameworks only in canonical form so that all terms are beta-reduced. By splitting the definition of objects into normal and neutral types, we characterize a bi-directional system to avoid definitional equality. Along with this system, we introduce the idea of hereditary substitutions. While applying a substitution to a canonical form does not necessarily produce a canonical result, the use of hereditary substitutions allows us to compute the canonical result of an ordinary substitution of one canonical form into another. For a more detailed explanation, please refer to hereditary substitutions[7] Also, we have signatures in which we declare constants and contexts to declare variables. A formal description is as follows:

$$\begin{array}{lll}
\text{kinds} & K ::= & \mathsf{type} \mid \Pi x : A.K \\
\text{atomic types} & P ::= & a \mid P\ N \\
\text{normal types} & A ::= & P \mid \Pi x : A_1.A_2 \\
\text{normal objects} & N ::= & \lambda x.N \mid R \\
\text{neutral objects} & R ::= & x \mid c \mid R\ N \\
\text{signatures} & \Sigma ::= & \cdot \mid \Sigma, a : K \mid \Sigma, c : A \\
\text{contexts} & \Gamma ::= & \cdot \mid \Gamma, x : A
\end{array}$$

In the above definition, we let a be type constants and c be object language level constants, both declared in the signature. Variables x are declared in contexts with their respective type. We assume that no more than one case of each constant is declared in the signature or context. When we add a declaration in the signature Σ, $\Sigma, c : A$ we assume that the constant c

does not appear in it. Renaming of variables and their respective binders in $\Pi x{:}A.K$, $\Pi x : A_2$ and $\lambda x.N$ is allowed. Also, we regard the non-dependent function type $A_1 \rightarrow A_2$ as $\Pi x : A_1.A_2$ where the variable x does not appear in A_2. If $c : A$ appears in the signature Σ then we write: $\Sigma(c) = A$. We write $\Sigma(a) = \mathsf{type}$ if a is a type definition that appears in the signature. For contexts, we have $\Gamma(x) = A$ if $x : A$ is in Γ. For simplicity, we use A and B to denote types, M and N for objects and P for atomic formulas of the form $a\ N_1 \ldots N_n$. The judgments of this type theory are:

Γ	$\vdash_\Sigma K \Leftarrow \mathsf{kind}$	K is a valid kind in context Γ
Γ	$\vdash_\Sigma A \Leftarrow \mathsf{type}$	A is a valid type in context Γ
Γ	$\vdash_\Sigma P \Rightarrow K$	P is atomic of kind K in context Γ
Γ	$\vdash_\Sigma N \Leftarrow A$	N is a valid normal object with type A in context Γ
Γ	$\vdash_\Sigma R \Rightarrow A$	N is a valid neutral object with type A in context Γ
	$\vdash \Sigma\ sig$	Σ is a valid signature
	$\vdash_\Sigma \Gamma\ ctx$	Γ is a valid context

where $X \Rightarrow Y$ means that we synthesize Y given X and $X \Leftarrow Y$ means we check X has a valid Y.

The inference rules for typing are as follows:

Inference rules for kinds:

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{type} \Leftarrow \mathsf{kind}} \qquad \frac{\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} \quad \Gamma, x{:}A \vdash_\Sigma B \Leftarrow \mathsf{kind}}{\Gamma \vdash_\Sigma \Pi x{:}A.B \Leftarrow \mathsf{kind}}$$

Inference rules for types:

$$\frac{}{\Gamma \vdash_\Sigma a \Rightarrow \Sigma(a)} \qquad \frac{\Gamma \vdash_\Sigma P \Rightarrow \Pi x{:}A.K \quad \Gamma \vdash_\Sigma N \Leftarrow A}{\Gamma \vdash_\Sigma P\ N \Rightarrow [N/x : A]K}$$

$$\frac{\Gamma \vdash_\Sigma P \Rightarrow \mathsf{type}}{\Gamma \vdash_\Sigma P \Leftarrow \mathsf{type}} \qquad \frac{\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} \quad \Gamma, x{:}A \vdash_\Sigma B \Leftarrow \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.B \Leftarrow \mathsf{type}}$$

Inference rules for objects

$$\frac{}{\Gamma \vdash_\Sigma c \Rightarrow \Sigma(c)} \qquad \frac{}{\Gamma \vdash_\Sigma x \Rightarrow \Gamma(x)}$$

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow \Pi x{:}A.B \quad \Gamma \vdash_\Sigma N \Leftarrow A}{\Gamma \vdash_\Sigma R\ N \Rightarrow [N/x : A]B}$$

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow P' \quad P \equiv P'}{\Gamma \vdash_\Sigma R \Leftarrow P} \qquad \frac{\Gamma,\ x{:}A \vdash_\Sigma N \Leftarrow B}{\Gamma \vdash_\Sigma \lambda x.N \Leftarrow \Pi x{:}A.B}$$

where $A \equiv A'$ represents that A and A' are α-equivalent.

Inference rules for Signatures

$$\frac{}{\vdash\ \cdot\ sig} \qquad \frac{\vdash\ \Sigma\ sig \quad \vdash_\Sigma K \Leftarrow kind}{\vdash\ \Sigma, a : K\ sig}$$

$$\frac{\vdash\ \Sigma\ sig \quad \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash\ \Sigma, c : A\ sig}$$

Inference rules for Contexts

$$\frac{}{\vdash\ \cdot\ ctx} \qquad \frac{\vdash_\Sigma \Gamma\ ctx \quad \vdash_\Sigma A \Leftarrow \mathsf{type}}{\Gamma, x{:}A\ ctx}$$

As we see from the above rules for typing, to check whether a type definition or constant c is of type A, we look up the signature Σ. To see whether a variable x has type A, we look up x in the context Γ to check for its declared type. Assuming A is a valid type, if under the assumption that a variable x has type A we can check that an object M had some type B, then the lambda-abstraction $\lambda x.M$ has function type $\Pi x{:}A.B$. If an object M has function type $\Pi x{:}A.B$ and we apply it to an object N of type A, the result $M\ N$ is of type B. Similarly, a type constant is a valid type if it was declared in the signature. A function type is valid if its input is a valid type and its output is a valid type under the assumption of the input being a valid type. Finally, valid signatures and contexts are formed by valid type declarations.

2.2 Example

Given the syntactic description of LF, we use the example of bounded polymorphic subtyping to illustrate the representation of an object language in LF.

2.2.1 Bounded polymorphic subtyping

Consider a type system where we have a universal supertype *top*, function types $T_1 \Rightarrow T_2$ and a bounded universally quantified type: $\forall\alpha \leq T_1.T_2$. This example was taken from the POPLmark challenge[1]. First, we specify the syntax for types here and wish to encode it in LF.

$$\begin{array}{lll} \text{type} & T ::= & top \mid \alpha \mid T_1 \Rightarrow T_2 \mid \forall\alpha \leq T_1.T_2 \\ \\ \text{context} & \Gamma ::= & \cdot \mid \Gamma, w{:}\alpha \leq T \end{array}$$

For the universal type, the binder α appears only in type T_2 and never in T_1. The context is used to keep track of these bindings so that variables always make sense in some context. The following is a variant of the specification of the POPLmark challenge so that we have an operational semantics for algorithmic subtyping. Now we give the judgments along with inference rules for subtyping: Judgment:

$$\Gamma \vdash S \leq T \quad S \text{ is a subtype of } T \text{ in the context } \Gamma$$

Inference rules

$$\frac{}{\Gamma \vdash T \leq \text{top}}\ \text{sa-top} \qquad \frac{\alpha \leq T \in \Gamma}{\Gamma \vdash \alpha \leq T}\ \text{sa-hyp}$$

$$\frac{}{\Gamma \vdash \alpha \leq \alpha}\ \text{sa-ref-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1 \Rightarrow S_2 \leq T_1 \Rightarrow T_2}\ \text{sa-arr}$$

$$\frac{\Gamma \vdash \alpha \leq U \quad \Gamma \vdash U \leq T}{\Gamma \vdash \alpha \leq T}\ \text{sa-tr-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma, w{:}\alpha \leq T_1 \vdash S_2 \leq T_2}{\Gamma \vdash \forall\alpha \leq S_1.S_2 \leq \forall\alpha \leq T_1.T_2}\ \text{sa-all}^{\alpha,w}$$

Since we use type variables in the reflexivity `sa-ref-tvar` and transitivity `sa-tr-tvar` rules instead of concrete types, we can use these rules algorithmically to determine a subtyping relation. Now we need an encoding of this formal system in LF so that we can discuss looping behavior that arise in backward proof search. The first main concern is how we should handle type variables. At the object language level, these variables will be introduced in only one rule: sa-all$^{\alpha,w}$. Also, in order to take advantage of higher order abstract syntax, we need to encode object level binders with binders in the meta language. This suggests that we should *not* use different kinds for explicit types and type variables. This poses the problem of how we can encode the rules with type variables since we cannot explicitly express variables in the object language in the meta-language. For example, in the rule `sa-ref-tvar`

$$\frac{\Gamma \vdash \alpha \leq U \quad \Gamma \vdash U \leq T}{\Gamma \vdash \alpha \leq T}\ \text{sa-tr-tvar}$$

object level variable α is a meta-level variable and we do not have access to it in our meta-language. We mentioned that there is only one rule that introduces variables so we will incorporate the representation of all rules that have type variables into that one rule.

First, we start by defining types and type constructors in our object in our meta language.

```
tp : type.

top : tp.

arr : tp -> tp -> tp.

all : tp -> (tp -> tp) -> tp.
```

Here type constructor `all` takes two arguments, the first being T_1 in our described rule and the second being an argument of function type (`tp` $\rightarrow$ `tp`). We generally view the non-dependent function type $\Pi x{:}A.B$ as $A \rightarrow B$. The second argument gives the binding structure that holds α in T_2. Since α does not appear in T_1 by definition, we can safely represent T_1 as an object of type T_1 in our meta language. On the other hand, T_2, will be represented by $(\lambda x.T_2 x)$ and hence a function type.

Next, we implement the subtyping relation. As mentioned earlier, type variables cannot be represented by meta-variables. We thus add the rules `sa-ref-tvar` and `sa-tr-tvar` to every occurrence of type variables in all other rules. Implementation-wise, we first declare a constant sub to denote the subtyping relation between two types `tp`.

```
sub : tp -> tp -> type.
```

then we proceed to the #top and functional inference rules:

```
sa_top : sub T top.

sa_arr : sub S2 T2 -> sub T1 S1

             -> sub (arr S1 S2) (arr T1 T2).
```

finally, we encode all type variable occurrences in the last rule for *all*.

```
sa_all : (Πa:tp.
              (ΠU.ΠV.sub U V ->  sub a U -> sub a V) ->
              sub a T1 -> sub a a ->
              sub (S2 a) (T2 a))
           -> sub T1 S1
           -> sub (all S1 (λa.(S2 a))) (all T1 (λa.(T2 a))).
```

Using a higher-order logic programming interpretation based on backchaining, we can read the clause `sa_arr` as follows: To prove the goal `sub (arr S1 S2) (arr T1 T2)`, we must prove `sub T1 S1` and then `sub S2 T2`. Similarly we can read the clause `sa_all`: To prove `sub (all S1 (λa.(S2 a))) (all T1 (λa.(T2 a)))`, we need to prove first `sub T1 S1`, and then assuming `tr:ΠU.ΠV. sub U V -> sub a U -> sub a V`, `w:sub a T1`, and `ref:sub a a`, prove that `sub (S2 a) (T2 a)` is true where `a` is a new parameter of type `tp`.

2.2.2 Depth first search

Depth first search is a backward proof search algorithm. It essentially starts with the goal that we wish to prove and tries to search backward through the inference rules. We would start with a goal, try to unify it with the head of an inference rule. If we succeed then we will try to satisfy each of the subgoals in the order specified. Conceptually, if the goal formula and any of the subgoal formulas are the same up to renaming, we cannot guarantee termination of proof search. This is because in order for proof search to terminate, the formula we search over has to decrease in size to reach the top-most rules. In this case, although we see that there is only one rule that can be applied for each of the top, functional and universal quantified types, there remains nondeterminism. For example, the left open premise of `sa-tr-tvar` is restricted to variables so we can search over only rules that have variables in the goal. This leaves us with four rules: `sa-top`, `sa-hyp`, `sa-ref-tvar` and `sa-tr-tvar`. However, it is possible that proof search over this system does not terminate since it is possible to apply the `sa-tr-tvar` rule again.

Tabled logic programming[16] is one solution to this problem. Still based on the backward search, tabling memoizes previously encountered subgoals so that we can efficiently use them later. This loop-detection mechanism solves the problem of nondeterminism. Tabling engine incorporated many ideas such as indexing of terms for efficient access and linearized terms so that unification is potentially a linear time algorithm. However, tabling still carries a substantial amount of overhead to store previously encountered goals, suspend computation, look up and then decide to continue search.

We explore a different approach. Instead of backward search, we turn to forward search. We exploit the consequence subformula property and generate axioms from the subformulas. From these axioms and inference rules, we carry out proof search in a forward way. Each rule now has a different interpretation. Previously, we satisfy each subgoal in the rule after the main goal unifies with a query. Now, we start with axioms and draw facts from previously reached goals and build our proof until we reach the query. Each inference rule now reads, if we have a proof for each of the premises, then we have a proof for the conclusion. Next we present the theoretical foundations for forward proof search and briefly discuss the issues of implementing such a prover for the Horn fragment. Then we give a description of our implementation of the forward proof search engine. Finally, we provide experimental results, insights in implementing such a prover and future improvements.

Chapter 3

Theoretical foundations

3.1 Horn clauses and Hereditary Harrop formulas

The Curry-Howard Isomorphism allows us to interpret types as propositions. Viewing types as propositions, LF types can be viewed as logical propositions. In this thesis, we mainly focus on a special subset of logical propositions : Horn clauses. In order for the horn clauses to be used for provability for sequent proofs and not for resolution refutations, we give the following definition for Horn clauses.

$$\begin{array}{lrcl} \text{Horn goals} & G & ::= & \text{true} \mid \text{D} \\ \text{Horn clauses} & D & ::= & G \rightarrow D \mid \Pi x{:}A.G \end{array}$$

Note that from this definition, universal quantifiers are not allowed to appear in goals or subgoals. Also, we disallow nested implications in the goals.

A natural extension of Horn clauses is to allow implications and universal quantifiers to appear everywhere in the formula. This extension is called Hereditary Harrop formulas.

$$\text{Hereditary Harrop formulas} \quad A \quad ::= \quad P \mid A_1 \rightarrow A_2 \mid \Pi x{:}A_1.A_2$$

3.2 Uniform Proofs

Uniform proofs provide an abstract logic programming language that specifies search behavior in logical frameworks. We regard atomic type $a\ M_1 \ldots M_n$ as an atomic proposition and denote it as P. Similarly, non-dependant function type $A_1 \rightarrow A_2$ corresponds to an implication in the meta-logic. Finally, the dependant function type $\Pi x{:}A.B$ translates to a universal quantified formula.

In a uniform sequent calculus, all inference rules are considered as either invertible or non-invertible. The proof search is thus distinguished between the uniform phase, where we apply invertible rules eagerly, and the focusing phase, where we pick a rule and focus on it. The principle behind uniform proof search is that applying all invertible rules eagerly and postponing application of all non-invertible rules lead to a uniform sequent calculus. To characterize uniform proofs, we define the following judgments:

$\Gamma \Longrightarrow A$	There is a *uniform proof* for A from the assumptions in Γ
$\Gamma \gg A \Longrightarrow P$	There is a *focused proof* for the atom P focusing on the proposition A using the assumptions in Γ

In the two judgments above, Γ is a context that keeps track of both dynamic assumptions, which can be used for proof search, and parameter assumptions, which cannot. Now we describe a uniform proof system.

$$\frac{\Gamma \gg A \Longrightarrow P \quad A \in \Gamma}{\Gamma \Longrightarrow P}\ \mathsf{choose} \qquad\qquad \frac{}{\Gamma \gg P \Longrightarrow P}\ \mathsf{hyp}$$

$$\frac{\Gamma, c{:}A_1 \Longrightarrow A_2}{\Gamma \Longrightarrow A_1 \to A_2} \to \mathsf{R} \qquad\qquad \frac{\Gamma \Longrightarrow A_1 \quad \Gamma \gg A_2 \Longrightarrow P}{\Gamma \gg A_1 \to A_2 \Longrightarrow P} \to \mathsf{L}$$

$$\frac{\Gamma, x{:}A \Longrightarrow B}{\Gamma \Longrightarrow \Pi x{:}A.B}\ \Pi\mathsf{R}$$

$$\frac{\Gamma \gg [M/x : A]B \Longrightarrow P \quad \Gamma \vdash M : A}{\Gamma \gg \Pi x{:}A.B \Longrightarrow P}\ \Pi\mathsf{L}$$

3.2.1 Substitutions

In practice, we typically do not guess the correct instantiation for the universally quantified variables in the $\Pi\mathsf{L}$ rule, but introduce a meta-variable which will be instantiated with unification later. Previously, we have been advocating the use of meta-variables as closures. Meta-variables are associated with a postponed substitution σ which is applied as soon as we know what the meta-variable stands for. A formal treatment for meta-variables based on contextual modal types can be found in[16, 11]. This allows us to formally distinguish between the ordinary bound variables introduced by $\Pi\mathsf{R}$ or a λ-abstraction and meta-variables u which are subject to instantiation. An advantage of this approach is that we localize dependencies while allowing in-place updates. Moreover, we can present all meta-variables that appear in a given term in a linear order and ensure that the types and contexts of meta-variables further to the right may mention meta-variables. When a meta-variable is introduced it is created as $u[\mathsf{id}_\Gamma]$ meaning it can depend on all the bound variables occurring in Γ. During search Γ is concrete and id_Γ will be unfolded. For example, if $\Gamma = x_1{:}A_1, x_2{:}A_2, x_3{:}A_3$ then $\mathsf{id}_\Gamma = (x_1/x_1, x_2/x_2, x_3/x_3)$. Moreover, we can easily characterize all the meta-variables occurring in a formula or sequent. The distinction between ordinary bound variables and meta-variables provides a clean basis for describing proof search. We will therefore enrich our lambda-calculus with first-class meta-variables denoted by u.

$$\begin{array}{lll} \text{Neutral Terms } M & ::= & \ldots \mid u[\sigma] \\ \text{Meta-variable context } \Delta & ::= & \cdot \mid \Delta, u{::}A[\Psi] \end{array}$$

σ denotes a postponed substitution which is applied as soon as we know what the meta-variable u stands for. The type of a meta-variable is $A[\Psi]$ denoting an object M which has type A in the context Ψ. We briefly highlight how contextual substitution into types and objects-level terms is defined to give an intuition, but refer the interested reader to [11]. We write $[\![\hat{\Psi}.M/u]\!]$ for replacing a meta-variable u with an object M. $\hat{\Psi}$ characterizes the ordinary bound variables occurring in M. This explicit listing of the bound variables occurring in M is necessary because of α-renaming issues and can be eliminated in an implementation. We only show contextual substitution into objects here. We note that there are no side-conditions necessary when substituting into λ-abstraction, since the objects M we substitute for u is closed with respect to $\hat{\Psi}$. When we encounter a meta-variable $u[\sigma]$, we first apply $[\![\hat{\Psi}.M/u]\!]$ to the substitution σ yielding σ' and then replace u with M and apply the substitution σ'. Note because of α-renaming issues we must possibly rename the domain of σ'.

$$\begin{array}{lcl}
[\![\hat{\Psi}.M/u]\!](a\ M_1 \dots M_n) & = & a\ N_1 \dots N_n \\
 & & \text{if for all } i\ [\![\hat{\Psi}.M/u]\!]M_i = N_i \\
[\![\hat{\Psi}.M/u]\!](A \to B) & = & A' \to B' \\
 & & \text{if } [\![\hat{\Psi}.M/u]\!]A = A' \text{ and } [\![\hat{\Psi}.M/u]\!]B = B' \\
[\![\hat{\Psi}.M/u]\!](\Pi x{:}A.B) & = & \Pi x{:}A'.B' \\
 & & \text{if } [\![\hat{\Psi}.M/u]\!]A = A' \text{ and } [\![\hat{\Psi}.M/u]\!]B = B'
\end{array}$$

$$\begin{array}{lcll}
[\![\hat{\Psi}.M/u]\!](\lambda y.N) & = & \lambda y.N' & \text{if } [\![\hat{\Psi}.M/u]\!]N = N' \\
[\![\hat{\Psi}.M/u]\!](u[\sigma]) & = & M' & \text{if } [\![\hat{\Psi}.M/u]\!]\sigma = \sigma' \text{ and } [\sigma'/\Psi]M = M' \\
[\![\hat{\Psi}.M/u]\!](u'[\sigma]) & = & u'[\sigma'] & \text{if } u' \neq u \text{ and } [\![\hat{\Psi}.M/u]\!]\sigma = \sigma' \\
[\![\hat{\Psi}.M/u]\!](R\ N) & = & (R\ N') & \text{if } [\![\hat{\Psi}.M/u]\!]R = R \text{ and } [\![\hat{\Psi}.M/u]\!](N) = N' \\
[\![\hat{\Psi}.M/u]\!](x) & = & x & \\
[\![\hat{\Psi}.M/u]\!](c) & = & c &
\end{array}$$

Simultaneous contextual substitution can be defined following similar principles. A simultaneous contextual substitution maps the meta-variables in its domain Δ' to another meta-variable context Δ which describes its range. More formally we can define simultaneous contextual substitutions as well-typed as follows:

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \theta : \Delta' \quad \Delta; \Psi \vdash M \Leftarrow A}{\Delta \vdash (\theta, \hat{\Psi}.M/u) : \Delta', u{::}A[\Psi]}$$

Finally, we are in the position to give a uniform calculus which introduces meta-variables in the rule $\Pi\mathsf{L}$, and delays their instantiation to the hyp rule where we rely on higher-order unification to find the correct instantiation. Since higher-order unification is undecidable in general we restrict it to the pattern fragment.

$\Delta; \Gamma \Longrightarrow A/(\theta, \Delta')$	There is a *uniform proof* for A from the assumptions in Γ where θ is a contextual substitution which instantiates the meta-variable in Δ and has range Δ'
$\Delta; \Gamma \gg A \Longrightarrow P/(\theta, \Delta')$	There is a *focused proof* for the atom P focusing on the proposition A using the assumptions in Γ where θ is a contextual substitution which instantiates the meta-variable in Δ and has range Δ'

In the rule $\Pi\mathsf{L}$ we introduce a new meta-variable $u[\mathsf{id}_\Gamma]$ of type $A[\Gamma]$. This means we introduce a meta-variable whose instantiation can depend on all the parameters occurring in Γ.

$$\frac{\Delta; \Gamma \gg A \Longrightarrow P/(\theta, \Delta') \quad A \in \Gamma}{\Delta; \Gamma \Longrightarrow P/(\theta, \Delta')}\, aR \qquad \frac{\Delta; \Gamma \vdash P' \doteq P/(\theta, \Delta')}{\Delta; \Gamma \gg P' \Longrightarrow P/(\theta, \Delta')}\, aL$$

$$\frac{\Delta; \Gamma, A_1 \Longrightarrow A_2/(\theta, \Delta')}{\Delta; \Gamma \Longrightarrow A_1 \to A_2/(\theta, \Delta')} \to R$$

$$\frac{\Delta; \Gamma \Longrightarrow A_1/(\theta_1, \Delta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \gg [\![\theta_1]\!]A_2 \Longrightarrow [\![\theta_1]\!]P/(\theta_2, \Delta_2)}{\Delta; \Gamma \gg A_1 \to A_2 \Longrightarrow P/([\![\theta_2]\!]\theta_1, \Delta_2)} \to L$$

$$\frac{\Delta; \Gamma, x{:}A \Longrightarrow B/(\theta, \Delta')}{\Delta; \Gamma \Longrightarrow \Pi x{:}A.B\ (\theta, \Delta')}\, \Pi R$$

$$\frac{\Delta, u{::}A[\Gamma]; \Gamma \gg [u[\mathsf{id}_\Gamma]/x]B \Longrightarrow P/((\theta, \hat{\Gamma}.M/u), \Delta')}{\Delta; \Gamma \gg \Pi x{:}A.B \Longrightarrow P/(\theta, \Delta')}\, \Pi L$$

3.3 Inverse method and focusing

An interesting alternative to backward proof search, is forward proof search based on the inverse method. This has potentially many advantages. While backward logic programming based depth first search is incomplete and requires backtracking, forward search provides a complete search strategy without backtracking. Similar to tabling, it also allows us to execute some specifications which were not previously executable. Next, we present a forward uniform proof system where we guess the correct instantiation. Afterwards, we describe a lifted version with meta-variables.

$\Gamma \xRightarrow{f} A$	There is a *forward uniform proof* for A from the assumptions in Γ
$\Gamma \gg A \xRightarrow{f} P$	There is a *forward focused proof* for the atom P focusing on the proposition A using the assumptions in Γ

The context Γ is now interpreted differently, in that sequents $\Gamma \xRightarrow{f} A$ and $\Gamma \gg A \xRightarrow{f} P$ assert that all assumptions in Γ as well as A, if the sequent is focused, are needed to prove the conclusion. General weakening is thus disallowed but incorporated in the rule $\mathsf{f{\to}R_2}$. Since our context Γ keeps track of dynamic assumption and parameters, we do not require it to be completely empty in the rule $\mathsf{f\text{-}ax}$. Instead we can think of it as the strongest context in which P is well-typed. Since we want to preserve that contexts

Γ are well-typed, we must make sure in the rule f$\rightarrow$L that the union of two contexts Γ_1 and Γ_2 is well-typed and preserves the present dependencies between parameter assumptions and dynamic assumptions. The rule f-drop was called in the backward uniform calculus choose. In the forward direction there is no choice but rather we must drop the formula out of the focus.

$$\frac{\Gamma \gg A \overset{f}{\Longrightarrow} P}{\Gamma \cup \{A\} \overset{f}{\Longrightarrow} P}\ \textsf{f-drop} \qquad \frac{}{\Gamma \gg P \overset{f}{\Longrightarrow} P}\ \textsf{f-ax}$$

$$\frac{\Gamma, c{:}A_1 \overset{f}{\Longrightarrow} A_2}{\Gamma \overset{f}{\Longrightarrow} A_1 \to A_2}\ \textsf{f}{\to}\textsf{R}_1 \qquad \frac{\Gamma \overset{f}{\Longrightarrow} A_2}{\Gamma \overset{f}{\Longrightarrow} A_1 \to A_2}\ \textsf{f}{\to}\textsf{R}_2$$

$$\frac{\Gamma_1 \overset{f}{\Longrightarrow} A_1 \quad \Gamma_2 \gg A_2 \overset{f}{\Longrightarrow} P}{\Gamma_1 \cup \Gamma_2 \gg A_1 \to A_2 \overset{f}{\Longrightarrow} P}\ \textsf{f}{\to}\textsf{L}$$

$$\frac{\Gamma, x{:}A \overset{f}{\Longrightarrow} B}{\Gamma \overset{f}{\Longrightarrow} \Pi x{:}A.B}\ \textsf{f}\Pi\textsf{R} \qquad \frac{\Gamma \gg [M/x : A]B \overset{f}{\Longrightarrow} P \quad \Gamma \vdash M \Leftarrow A}{\Gamma \gg \forall x.A \overset{f}{\Longrightarrow} P}\ \textsf{f}\Pi\textsf{L}$$

Next, we prove soundness and completeness of this forward uniform calculus.

3.3.1 Soundness

Theorem 3.1 (Soundness)

1. *If* $\Gamma \overset{f}{\Longrightarrow} A$ *then* $\Gamma \Longrightarrow A$
2. *If* $\Gamma \gg A \overset{f}{\Longrightarrow} P$ *then* $\Gamma \gg A \Longrightarrow P$*.*

Proof: Straightforward structural induction.

Case: $\mathcal{D} = \dfrac{\Gamma \gg A \xrightarrow{f} P}{\Gamma, A \xrightarrow{f} P}$

$\Gamma \gg A \Longrightarrow P$ by i.h. (2)

$\Gamma, c{:}A \gg A \Longrightarrow P$ by weakening $\Gamma, c{:}A \Longrightarrow P$ by rule

Case 2 $\mathcal{D} = \dfrac{\Gamma, A_1 \xrightarrow{f} A_2}{\Gamma \xrightarrow{f} A_1 \to A_2}$

$\Gamma, A_1 \Longrightarrow A_2$ by ih (1)

$\Gamma \Longrightarrow A_1 \to A_2$ by rule

Case 3 $\mathcal{D} = \dfrac{\Gamma \xrightarrow{f} A_2}{\Gamma \xrightarrow{f} A_1 \to A_2}$

$\Gamma \Longrightarrow A_2$ by ih (1)

$\Gamma, c{:}A_1 \Longrightarrow A_2$ by weakening

$\Gamma \Longrightarrow A_1 \to A_2$ by rule

□

3.3.2 Completeness

Theorem 3.2 (Completeness)

1. *If* $\Gamma \Longrightarrow A$ *then* $\Gamma' \xrightarrow{f} A$ *where* $\Gamma' \subseteq \Gamma$.

2. *If* $\Gamma \gg A \Longrightarrow P$ *then* $\Gamma' \gg A \xrightarrow{f} P$ *where* $\Gamma' \subseteq \Gamma$

Proof: Straightforward structural induction.

Case 1 $\mathcal{D} = \dfrac{\Gamma \gg A \Longrightarrow P \qquad A \in \Gamma}{\Gamma \Longrightarrow P}$

$\Gamma' \gg A \overset{f}{\Longrightarrow} P, \Gamma' \subseteq \Gamma$ by i.h. (2)

Subcase 1 $A \in \Gamma'$

$\Gamma' \subseteq \Gamma$ by assumption

$\Gamma' \cup \{A\} \subseteq \Gamma$ by set

$\Gamma' \cup \{A\} \overset{f}{\Longrightarrow} P, \Gamma' \cup \{A\} \subseteq \Gamma$ by rule

Subcase 2 $A \notin \Gamma'$

$\Gamma' \subseteq \Gamma$ and $A \in \Gamma$ by assumption

$\Gamma' \cup \{A\} \overset{f}{\Longrightarrow} P, \Gamma' \cup \{A\} \subseteq \Gamma$ by rule

Case 2 $\mathcal{D} = \dfrac{\Gamma \Longrightarrow A_1 \qquad \Gamma \gg A_2 \Longrightarrow P}{\Gamma \gg A_1 \to A_2 \Longrightarrow P}$

$\Gamma_1 \overset{f}{\Longrightarrow} A_1$ and $\Gamma_1 \subseteq \Gamma$ by ih (1)

$\Gamma_2 \gg A_2 \overset{f}{\Longrightarrow} P$ and $\Gamma_2 \subseteq \Gamma$ by ih (2)

$\Gamma_1 \cup \Gamma_2 \gg A_1 \to A_2 \overset{f}{\Longrightarrow} P$ and $\Gamma_1 \cup \Gamma_2 \subseteq \Gamma$ by rule

$\square$

3.3.3 Subformula property

The forward uniform proof system presented gives rise to a proof search method based on the inverse method central to which is the notion of subformula. We outline the notion of subformulas and present a lifted calculus.

We adapt the standard definition of subformulas to the higher-order setting where objects may contain meta-variables. The immediate free subformula of the negative occurrence of the formula $\Pi x{:}A.B$ in the context Γ is then $[u[\mathsf{id}_\Gamma]/x]B$. The immediate ground subformula of the negative occurrence of the formula $\Pi x{:}A.B$ in the context Γ is $[M/x{:}A]B$.

Signed subformulas

Definition 3.3 (Signed subformulas) *Free signed subformulas and its immediate signed subformulas are defined inductively as follows.*

signed subformula	*free signed subformula*	*immediate signed subformula*
$(A \to B)^-$	A^+, B^-	A^+, B^-
$(A \to B)^+$	A^-, B^+	A^-, B^+
$(\Pi x{:}A.B)^-$	$([u[\mathit{id}_\Gamma]/x]B)^-$	$([M/x]B)^-$
$(\Pi x{:}A.B)^+$	$([a/x]B)^+$	$([a/x]B))^+$

While it is possible to give an algorithm for computing subformulas for LF we keep this development slightly less formal. However, it is possible to formalize the subsequent development using the algorithm presented later.

Definition 3.4 (Subformula property)

1. *Every derivation of a uniform sequent* $\Gamma \Longrightarrow A$ *consists of signed ground subformulas of signed formulas in* Γ^- *and* A^+.

2. *Every derivation of a focused* $\Gamma \gg A \Longrightarrow P$ *consists of signed ground subformulas of signed formulas in* Γ^- *and* A^+.

Theorem 3.5 (Ground subformula property of uniform proofs)

1. *Let $\mathcal{D}$ be a derivation of a signed uniform sequent $\Gamma^- \Longrightarrow A^+$ then every signed uniform sequent $\Gamma_0^- \Longrightarrow A_0^+$ or signed focused sequent $\Gamma_1^- \gg A_1^- \Longrightarrow P_1$ occurring in $\mathcal{D}$ fulfills the subformula property, i.e. $[\Gamma_0^-, A_0^+] < [\Gamma^-, A^+]$ or $[\Gamma_1^-, A_1^-, P_1^+] < [\Gamma^-, A^+]$.*

2. *Let $\mathcal{D}$ be a derivation of a signed focused sequent $\Gamma^- \gg A^- \Longrightarrow P^+$ then every signed uniform sequent $\Gamma_0^- \Longrightarrow A_0^+$ or signed focused sequent $\Gamma_1^- \gg A_1^- \Longrightarrow P_1$ occurring in $\mathcal{D}$ fulfills the subformula property, i.e. $[\Gamma_0^-, A_0^+] < [\Gamma^-, A^-, P^+]$ or $[\Gamma_1^-, A_1^-, P_1^+] < [\Gamma^-, A^-, P^+]$.*

Thus when we search for a proof of a particular signed sequent $\Gamma \Longrightarrow A$ or $\Gamma \gg A \Longrightarrow P$ resp. we can restrict our search to sequents consisting of signed subformulas of $[\Gamma^-, A^+]$. When $[\Gamma^-, A^+]$ contains quantifiers, it may have an infinite number of signed subformulas, so the subformula property does not restrict the search space good enough. However any signed formula has only a finite number of free signed subformulas.

Next, we consider *free signed subformula property.* We will often represent signed subformulas of a given uniform sequent $\Gamma^- \Longrightarrow A^+$ in the form $[\![\theta]\!]\Gamma_0^- \Longrightarrow [\![\theta]\!]A_0^+$, where θ is a substitution from the meta-variables Δ to some ground instance, i.e. $\cdot \vdash \theta : \Delta$ and $\Delta; \Gamma_0^- \Longrightarrow A_0^-$. We call this representation the representation *via free signed subformula*. Similarly we often represent signed subformulas of a given focused sequent $\Gamma^- \gg A^- \Longrightarrow P^+$ in the form $[\![\theta]\!]\Gamma_0^- \gg [\![\theta]\!]A^- \Longrightarrow [\![\theta]\!]P^+$. Moreover, we often write $S = [\Gamma^-, A+]$

as an abbreviation for the sequent $\Gamma \Longrightarrow A$, and $\Delta \vdash S$ as an abbreviation for $\Delta; \Gamma \Longrightarrow A$.

Lemma 3.6 *Let $S_0 = [\Gamma_0^-, A_0^+]$, and $S_1 = [\Gamma_1^-, A_1^+]$ be free signed subformulas s.t. $\Delta_0 \vdash S_0$ and $\Delta_1 \vdash S_1$. Then $[\Gamma_1^-, A_1^+] < [\Gamma_0^-, A_0^+]$ i.e. S_1 is a signed subformula of S_0, iff $S_1 = [\![\theta]\!]S$ for some signed sequent S s.t. S is a free signed subformula of S_0, where $\Delta_1 \vdash \theta : \Delta$ and $\Delta \vdash S$ and $\Delta \cap \Delta_0 = \emptyset$.*

Every signed subformula of a closed signed formula can be obtained from a free signed subformula by applying a contextual substitution.

Free subformula property

We now reformulate the subformula property.

Corollary 3.7 (Free subformula property) *Let $\mathcal{D}$ be a derivation of a closed signed uniform sequent $S = \Gamma^- \Longrightarrow A^+$ or closed signed focused sequent $\Gamma^- \gg A^- \Longrightarrow P^+$. Every signed sequent occurring in $\mathcal{D}$ has the form $[\![\theta]\!]S_0$ for a free signed sequent S_0 of S and a substitution θ s.t. $\Delta \vdash S_0$ and $\cdot \vdash \theta : \Delta$.*

Suppose we want to check the provability of a closed signed sequent S. By the previous corollary, we can restrict signed formulas occurring in the derivation to signed sequents of the form $[\![\theta]\!]S_0$ where S_0 is a free signed sequent of S s.t. $\Delta \vdash S_0$ and $\cdot \vdash \theta : \Delta$. Since this applies to axioms as well, every axiom has the form $\Gamma \gg [\![\theta]\!]([\![\rho]\!]P) \overset{f}{\Longrightarrow} [\![\theta]\!]P'$ where P and P'

are atomic free signed subformulas of the sequent S and $[\![\theta]\!]([\![\rho]\!]P) = [\![\theta]\!]P'$, and ρ is a renaming of meta-variables occurring in P, and Γ characterizes the parameters occurring in $[\![\theta]\!]P'$ and $[\![\theta]\!]([\![\rho]\!]P)$ respectively. For any given P, P' there may be an infinite number of such axioms because of different choices for substitutions θ, but there is only a finite number of pairs of free signed sequents. We can choose a most general axiom that represents all axioms.

We will now introduce a forward calculus $\mathcal{F}^A$ for the inverse method with meta-variables. The calculus is based on the idea of representing sequents through free subformulas and using most general unifiers. Since higher-order unification is only decidable for patterns, we restrict our attention for now to this fragment.

A sequent S in the original forward calculus for closed sequents, is an instance of a sequent $[\![\theta]\!]S_0$ in the calculus $\mathcal{F}^A$ if there exists a grounding substitution ρ s.t. $[\![\rho]\!][\![\theta]\!]S_0 = S$.

Given a closed signed sequent $S = [\Gamma^-, A^+]$, for every sequent S_0 provable in $\mathcal{F}$ (the original forward system for closed sequents) where $S_0 < S$, there exists a sequent S' provable in $\mathcal{F}_A$ (the forward system with meta-variables) such that S_0 is an instance of S'. Next, we give a forward calculus with meta-variables. Unlike more standard presentation where we associate a substitution θ with each of the formulas in Γ and the conclusion A, we will associate a substitution θ with a sequent. The judgment $(\Gamma \longrightarrow A) \cdot \theta$ denotes a sequent where $[\![\theta]\!]\Gamma \longrightarrow [\![\theta]\!]A$. This will be easier to implement, and models more closely our prototype. In the hypothesis rule where we unify the assumption $[\![\rho]\!]P'$ with P we keep a context Γ which describes the parameters occurring in P' and P. As mentioned earlier, typical formulations of forward calculi require the context to be empty, since they do not keep track explicitly of the parameters introduced during proof search. Due to the dependent nature of our calculus, and the fact that we would like to preserve that all propositions are well-typed, we keep track of parameters explicitly and allow the context Γ in this hypothesis rule to be non-empty. Our intention is that Γ describes all the parameters occurring in P and P'.

$$\frac{(\Delta; \Gamma \gg A \xRightarrow{f} P) \cdot \theta}{(\Delta; \Gamma \xRightarrow{f} P) \cdot \theta} \qquad \frac{\Delta; \Gamma \vdash [\![\rho]\!]P' \doteq P/\theta}{(\Delta; \Gamma \gg [\![\rho]\!]P' \xRightarrow{f} P) \cdot \theta}$$

$$\frac{(\Delta; \Gamma \xRightarrow{f} B) \cdot \theta}{(\Delta; \Gamma \xRightarrow{f} (A \to B)) \cdot \theta} \qquad \frac{(\Delta; \Gamma, c{:}A_1 \xRightarrow{f} A_2) \cdot \theta}{(\Delta; \Gamma \xRightarrow{f} (A_1 \to A_2)) \cdot \theta}$$

$$\frac{(\Delta_1; \Gamma_1 \xRightarrow{f} A_1) \cdot \theta_1 \qquad (\Delta_2; \Gamma_2 \gg A_2 \xRightarrow{f} P) \cdot \theta_2}{(((\Delta_1 \cup \Delta_2); \Gamma_1 \cup \Gamma_2) \gg (A_1 \to A_2) \xRightarrow{f} P) \cdot [\![\theta]\!]\theta_1'}$$

$$\begin{array}{ll} \text{where} & \text{ext}(\Delta_1 \cup \Delta_2, \theta_1) = \theta_1' \\ & \text{ext}(\Delta_1 \cup \Delta_2, \theta_2) = \theta_2' \\ & \mathsf{mgu}(\theta_1', \theta_2') = \theta \end{array}$$

$$\frac{(\Delta; \Gamma, x{:}A \xRightarrow{f} B) \cdot \theta}{(\Delta; \Gamma \xRightarrow{f} (\Pi x{:}A.B) \cdot \theta}$$

$$\frac{(\Delta, u{::}A[\Gamma]; \Gamma \gg [u[\mathsf{id}_\Gamma]/x]B \xRightarrow{f} P) \cdot (\theta, \hat{\Gamma}.M/u) \quad u \text{ is new}}{(\Delta; \Gamma \gg (\Pi x{:}A.B) \xRightarrow{f} P) \cdot \theta}$$

In the implication left rule, we must union not only the assumptions in Γ_1 and in Γ_2, but we also must union the meta-variables occurring in both branches. Since meta-variables occurring in both branches of the proof, may have been instantiated differently, we must reconcile their different instantiations in θ_1 and θ_2 by unifying them. Before we can unify them we first extend them with identity substitution s.t. they share the same domain. This extension is denoted with $\text{ext}(\Delta_1 \cup \Delta_2, \theta_1) = \theta_1'$ and $\text{ext}(\Delta_1 \cup \Delta_2, \theta_2) = \theta_2'$ respectively. Proofs for the free subformula property can be found in the LFMTP workshop paper[17].

Chapter 4

Implementation

Forward uniform proofs discussed in the previous chapter is a calculus designed for forward proof search. This chapter discusses an implementation of the forward inverse method with focusing for the Horn fragment. To illustrate the issues arising in an implementation, we introduce the example of computing the n^{th} Fibonacci number. Fibonacci number calculation is a good example because it requires intensive computation and use of previously derived facts or goals. Also, by defining addition ourselves, we need to calculate each Fibonacci number by adding one to the previous Fibonacci number and try to satisfy the rules. Further, the naïve algorithm for calculating Fibonacci numbers is exponential while an algorithm that uses previously calculated results is only linear. This will lead to generation of many rules and will allow us to better understand and compare the two loops, activation of facts and activation of rules.

4.1 Example : Fibonacci

We first define natural numbers and addition:

```
nat : type.
z : nat.
s : nat -> nat.
add : nat -> nat -> nat -> type.
add_z : add z N N.
add_s : add (s N1) N2 (s N) <- add N1 N2 N.
```

Here, we represent each inference rule in the object-language as a Horn clause. Natural numbers are used via a unary representation and addition is defined by incrementing both one summand and the result by 1.

Next we define computation of Fibonacci numbers recursively.

```
fib : nat -> nat -> type.
fib_0 : fib z z.
fib_1 : fib (s z) (s z).
fib_rec : fib (s (s X)) N
            <- fib X Y
            <- fib (s X) Y'
            <- add Y Y' N.
```

The two base cases for recursion simply define our Fibonacci sequence to start from 0 and 1. The recursive step fib_rec computes `fib(X+2)` if we add `fib(X)` and `fib(X+1)`.

4.2 Computation of subformulas

By the subformula property, in a cut-free sequent calculus discussed above, we only need to consider subformulas of the goal sequent relevant to proof search. Therefore, the first step in implementing the inverse method is the computation of subformulas from the signature of an LF program. This is done via the following two judgments:

$\Delta;\Gamma \vdash_n A/(\mathcal{N},\mathcal{P})$	A occurs negative and $\mathcal{N}$ and $\mathcal{P}$ are the negative and positive subformulas of A, respectively
$\Delta;\Gamma \vdash_p A/(\mathcal{N},\mathcal{P})$	A occurs positive and $\mathcal{N}$ and $\mathcal{P}$ are the negative and positive subformulas of A, respectively

Positive subformula generation

$$\frac{}{\Delta;\Gamma \vdash_p a\ M_1 \ldots M_n/(\cdot\,, \{\Delta;\Gamma \vdash a\ M_1 \ldots M_n\})}$$

$$\frac{\Delta;\Gamma, x{:}A \vdash_p B/(\mathcal{N},\mathcal{P})}{\Delta;\Gamma \vdash_p \Pi x{:}A.B/(\mathcal{N},\mathcal{P})}$$

$$\frac{\Delta;\Gamma \vdash_n A/(\mathcal{N}_1,\mathcal{P}_1) \quad \Delta;\Gamma \vdash_p B/(\mathcal{N}_2,\mathcal{P}_2)}{\Delta;\Gamma \vdash_p A \to B/(\mathcal{N}_1 \cup \mathcal{N}_2, \mathcal{P}_1 \cup \mathcal{P}_2)}$$

Negative subformula generation

$$\frac{}{\Delta;\Gamma \vdash_n a\ M_1 \ldots M_n/(\{\Delta;\Gamma \vdash a\ M_1 \ldots M_n\}, \cdot)}$$

$$\frac{\Delta, u{::}A[\Gamma];\Gamma \vdash_n [\![u[id_\Gamma]/x]\!]B/(\mathcal{N},\mathcal{P})}{\Delta;\Gamma \vdash_n \Pi x{:}A.B/(\mathcal{N},\mathcal{P})}$$

$$\frac{\Delta;\Gamma \vdash_p A/(\mathcal{N}_1,\mathcal{P}_1) \quad \Delta;\Gamma \vdash_n B/(\mathcal{N}_2,\mathcal{P}_2)}{\Delta;\Gamma \vdash_n A \to B/(\mathcal{N}_1 \cup \mathcal{N}_2, \mathcal{P}_1 \cup \mathcal{P}_2)}$$

We see that the subformulas generated are in fact closed atomic subformulas that are described by $\Delta;\Gamma \vdash a\ M_1 \ldots M_n$. In the above procedure, Δ is the context of meta-variables. Since we are in a dependently typed setting, context Γ describes the parameters in $a\ M_1 \ldots M_n$. We omit the parameter context since there are no parameters, we generate the following subformulas:

	add_z	*add_s*
Neg	$\Delta_1 \vdash \texttt{add}\ z\ N[\cdot]\ N[\cdot]$	$\Delta_2 \vdash \texttt{add}\ (\texttt{s}\ N1[\cdot])\ N2[\cdot]\ (\texttt{s}\ N3[\cdot])$
Pos	$\cdot$	$\Delta_2 \vdash \texttt{add}\ N1[\cdot]\ N2[\cdot]\ N3[\cdot]$

where $\Delta_1 = N :: nat[\cdot]$ and $\Delta_2 = N1 :: nat[\cdot]$, $N2 :: nat[\cdot]$, $N3 :: nat[\cdot]$

	fib_0	*fib_1*	*fib_rec*
Neg	$\cdot \vdash fib\ z\ z$	$\cdot \vdash fib\ (s\ z)\ (s\ z)$	$\Delta \vdash fib\ (s\ (s\ X))\ N$
Pos	$\cdot$	$\cdot$	$\Delta \vdash fib\ X\ Y$
			$\Delta \vdash fib\ (s\ X)\ Y'$
			$\Delta \vdash add\ Y\ Y'\ N$

where $\Delta = N :: nat[\cdot], X :: nat[\cdot], Y :: nat[\cdot], Y' :: nat[\cdot]$

Since the query is atomic, we simply generate the subformula:

$\cdot,\ N :: nat[\cdot] \vdash_\Sigma fib(s\ (s\ (s\ (s\ z))))N$. Using the inverse method, usually, we generate labels for subformulas so that we do not need to store and utilize subformulas of large size.

4.3 Axiom generation

Given a set of negative subformulas $\mathcal{N}$ and a set of positive subformulas $\mathcal{P}$, we generate a *focused axiom* if a negative formula unifies with a positive formula. We can describe the algorithm more formally as follow:

Let $\Delta_n; \Gamma_n \vdash N$ be a negative subformula in $\mathcal{N}$ and $\Delta_p; \Gamma_p \vdash P$ be a positive subformula in $\mathcal{P}$. To generate an axiom, we must unify negative and positive subformulas via the following procedure:

Given a negative subformula $\Delta_n; \Gamma_n \vdash N$ and a positive subformula $\Delta_p; \Gamma_p \vdash P$ we generate axioms:

1 $\Gamma_n = \Gamma_p$

a if $\Delta_n, \Delta_p; \Gamma_n \vdash N \doteq P/(\theta, \Delta)$ s.t. $[\![\theta]\!]N = [\![\theta]\!]P$ then we generate an axiom of the form: $\Delta'; \Gamma_n \gg [\![\theta]\!]N \overset{f}{\Longrightarrow} [\![\theta]\!]N$ where $\Delta \vdash \theta(\Delta_n, \Delta_p)$

b if $\Delta_n, \Delta_p; \Gamma_n, \Gamma_p \vdash N \doteq P/(\theta, \Delta)$ s.t. $[\![\theta]\!]N = [\![\theta]\!]P$ then we generate an axiom of the form: $\Delta'; \Gamma_n, \Gamma_p \gg [\![\theta]\!]N \overset{f}{\Longrightarrow} [\![\theta]\!]N$ where $\Delta \vdash \theta(\Delta_n, \Delta_p)$

2 $\Gamma_n \neq \Gamma_p$ if $\Delta_n, \Delta_p; \Gamma_n, \Gamma_p \vdash N \doteq P/(\theta, \Delta)$ s.t. $[\![\theta]\!]N = [\![\theta]\!]P$ then we generate an axiom of the form: $\Delta'; \Gamma_n, \Gamma_p \gg [\![\theta]\!]N \xRightarrow{f} [\![\theta]\!]N$ where $\Delta \vdash \theta(\Delta_n, \Delta_p)$

Given a query, we regard it as a negative subformula and try to unify it with all positive formulas to generate axioms need for proof search. We then compute the minimal set of axioms by checking forward and backward subsumption.

For the Fibonacci example, from the signed subformulas generated in the previous step, we unify a negative and a positive subformula to generate a focused axiom.

Axioms generated from subformulas in the signature for type family *add*:

$$\begin{array}{ll} \Delta_{a1} \vdash_\Sigma add\ z\ N\ N & \text{where } \Delta_1 = N :: nat[\cdot] \\ \Delta_{a2} \vdash_\Sigma add(s\ N_1)\ N_2\ (s\ N) & \text{where } \Delta_2 = N :: nat[\cdot], N1 :: nat[\cdot], N2 :: nat[\cdot] \end{array}$$

Axioms generated from subformulas in the signature for type family fib:

$$\begin{array}{l} \cdot \vdash_\Sigma fib\ z\ z \\ \cdot \vdash_\Sigma fib\ (s\ z)\ (s\ z) \\ \Delta_{f1} \vdash_\Sigma fib(s\ (s\ X))\ N \quad \text{where } \Delta_{f1} = X :: nat[\cdot], N :: nat[\cdot] \end{array}$$

Axiom generated from the query: $\cdot,\ N :: nat[\cdot] \vdash_\Sigma fib(s\ (s\ (s\ (s\ z))))N$.

4.4 Rule compilation

Following Chaudhuri *et al*, our implementation creates big-step derived rules by chaining all the focused rules together to form a focused thread and chaining all the uniform rules together to form a uniform thread. A *focused thread* of a derivation in $\mathcal{F}^A$ is a segment of the derivation that begins with a rule application of the rule $\mathsf{f\text{-}drop}$, has as leafs uniform sequents $\Delta_i; \Gamma_i \overset{f}{\Longrightarrow} A_i$ together with one focused atomic sequent $S = \Delta_k; \Gamma_k \gg P \overset{f}{\Longrightarrow} C$, and consists only of focused sequents. A *uniform thread* of a derivation in $\mathcal{F}^A$ is a segment of the derivation that begins with a uniform sequent, and consists only of uniform derivations, until it is a uniform atomic sequent.

4.4.1 Rule generation

Formally, we describe rule generation for Horn clauses as follows:

Disregarding type and kind definitions, let γ be a clause in the signature Σ

of a given program.

Judgment (currently for Horn fragment)

$\gamma \xrightarrow{r} \Delta \vdash \Omega, F$

A focused rule is generated from a clause γ in Σ where
$\Omega = [\cdot \xrightarrow{f} P_1, \ldots, \cdot \xrightarrow{f} P_n]$ and
$F = \Sigma \gg C \Rightarrow C$ are defined in the meta-variable context Δ.

Rule generation

$$\frac{P = a\ M_1 \ldots M_n}{P \xrightarrow{r} \Delta \vdash \cdot, \Sigma \gg P \Rightarrow P)}$$

$$\frac{B \xrightarrow{r} \Delta' \vdash \Omega, F \quad \Delta' = \Delta, u :: A[\cdot]}{\Pi x{:}A.B \xrightarrow{r} \Delta' \vdash \Omega, F}$$

$$\frac{B \xrightarrow{r} \Delta' \vdash \Omega, F}{A \to B \xrightarrow{r} \Delta' \vdash [\cdot \xrightarrow{f} A, \Omega], F}$$

Assuming atoms are negative, rules generated from the above algorithm are big-step *focused* rules. Since focused axioms are only generated prior to proof search, we can construct *derived rules* to simplify the search procedure. An interesting observation is that if the bias of predicates change, we generate a different set of rules. While the rules generated now are used for proof search using the inverse method, we will have SLD resolution if we completely switch the polarities.

4.4.2 Compiling derived rules

With a set of focused axioms $\mathcal{A}$ and a set of focused rules $\mathcal{R}$, we satisfy the focused premise of each rule R $\in \mathcal{R}$ with an axiom $A \in \mathcal{A}$. This results in derived rules that model closely the inference rules given in the signature. If there is only one premises (which is focused), a uniform fact is generated instead of a derived rule. Uniform facts generated in this matter should be either constants in the signature or rules that describe facts in the signature.

4.5 Inverse Method with Fact Activation

This loop essentially follows the ideas used by K.Chaudhuri in his implementation of the inverse method for linear logic. Given an ordered set of facts $\mathcal{F}$, and an ordered set of derived rules $\mathcal{R}$, each fact is picked and used to construct all possible pre-instantiated rules by looping over the existing rules. If all premises are met, a new fact is created. Finally, each new fact is checked for subsumption and added to the set of facts, each new rule is added to the set of rules and we continue with the succeeding facts. When all new facts are subsumed our search saturates and terminates with the facts. This method causes the set of rules and facts to grow, in each iteration of fact activation, the number of rules created is potentially the total number of open premises from all the rules in the previous iteration.

4.5.1 Fact activation

To activate a fact f, with a set of rules $\mathcal{R} = \{R_1, \ldots, R_k\}$ the following procedure is performed:

Take the first rule $\Delta \vdash \Omega \Rightarrow P$, we unify f with all its premises $\Omega = \{P_1, \ldots, P_n\}$

n = 1 If f unifies with the only premise P of rule R_k, and $\theta = mgu(f, P)$, a new fact $\Delta' \vdash [\![\theta]\!] f$ is created from the conclusion of the rule. and we continue to the rest of the rules.

n > 1 If the premise P_i unifies with f where $\theta = mgu(f, P_i)$, a new pre-instantiated rule $\Delta' \vdash [\![\theta]\!]\Omega \Rightarrow [\![\theta]\!]P$ (s.t. $\Delta' \vdash \theta : \Delta$ and $[\![\theta]\!]\Omega = \xrightarrow{f}[\![\theta]\!]P_1, \ldots, \xrightarrow{f}[\![\theta]\!]P_{i-1} \ldots [\![\theta]\!]P_n$)is created and added to $\mathcal{R}$ and new rules are created by applying the remaining premises to f. In brief, we compute all possible permutations of the premises to satisfy f.

3 If the premise does not unify, the rest of the premises are applied to fact f.

4.5.2 Proof Search

After each activation of a fact $f \in \mathcal{F}$, possibly some new facts $\mathcal{F}'$ and some new rules $\mathcal{R}'$ are created. Every facts $f' \in \mathcal{F}'$ is checked for subsumption to see whether there exists a fact $f \in \mathcal{F}$ such that f' is an instance of f and $\mathcal{R}'$ is added to the end of the previous facts. Once a fact is used, it will never be used again. By activating each fact and applying it to every premise of every present rule, completeness is guaranteed.

4.6 Inverse method with Rule Activation

Similar to the search procedure that activates facts to pre-instantiate rules and produce new facts, another approach which activates rules can be taken. Both procedures construct all possible permutations of facts to satisfy premises of rules to preserve completeness.

Given a fact database $\mathcal{D}$, a new fact $f \in \mathcal{F}$ (thus $f \notin \mathcal{D}$) and a rule $R \in \mathcal{R}$ where $\mathcal{R}$ is the set of derived rules generated from the signature and $\mathcal{F}$ is the ordered set of new facts, we generate a set of intermediate rules $\mathcal{R}'$. Next, we search over this set $\mathcal{R}'$ with facts from $\mathcal{F}$. Each fact $f' \in \mathcal{D}$ will be used to instantiate premises from rule $R' \in \mathcal{R}'$. When all premised of a rule in instantiated, a new fact has been deduced.

Searching over all pre-instantiated rules generated by the new fact f and $\mathcal{R}$ will result in a list of facts $\mathcal{F}'$. F is moved to $\mathcal{D}$ and we check if any $f' \in \mathcal{F}'$ is subsumed by a fact $f_d \in \mathcal{D}$. If f' is subsumed by some f_d, then it is discarded, otherwise, it is kept for further search.

All of the rules generated by the above process are discarded and we continue the search with the next fact in $\mathcal{F}$ with the fixed number of derived rules generated from the signature $\mathcal{R}$. Proof search terminates when new facts are subsumed by previous facts.

4.6.1 Activation of a rule

Given rule R that has premises $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ and a conclusion P, a new fact $f \in \mathcal{F}$, where $\mathcal{F}$ is a list of new facts and a database of facts $\mathcal{D}$ the following procedure is performed:

1 We loop through the premises of R and $\mathcal{P}$ to instantiate each P_i with f to generate either a new fact or pre-instantiated rules to search over. The number of pre-instantiated rules is at most the number of premises of each rule. Let n be the number of premises of R.

n = 1 If rule R has only one premise P_1, a new fact is created from the conclusion of the rule $P[\theta]$, where $\theta = mgu(P_1, f)$. We then move this fact to the a new set of new facts $\mathcal{F}'$.

n > 1 If rule R has more than two premises and there exists a $\theta = mgu(p_i, f)$ such that $p_i[\theta] = f[\theta]$, then a new rule $\mathcal{R}'$ with premises $\{P_1[\theta], P_2[\theta], \ldots, P_{i-1}[\theta], P_{i+1}[\theta], \ldots, P_n[\theta]\}$ and a conclusion $P[\theta]$ is created.

2 For R and f, we have generated a list of pre-instantiated rules $\mathcal{R}'$ and a set of new facts $\mathcal{F}'$. Now, we satisfy each premise P_i of each rule R' with some fact $f_d \in \mathcal{D}$ by finding $\theta_i = mgu(f_d, P_i)$. If all premises of R' are satisfied, a new fact $c[\theta_1][\theta_2]\ldots[\theta_n]$ is created and added to $\mathcal{F}$.

4.6.2 Proof Search

We loop over a dynamically growing list of new facts. In each stage of the loop, we activate all derived rules generated from the signature. After activation of each rule R, all new rules are discarded and the fact f used for the application is moved to the database of facts $\mathcal{D}$. New facts will be created for further activation of rules. In this loop, the number of rules stored and searched over in each iteration will not change. A growing set of facts are used to satisfy premises of these rules. Proof search terminates when all new facts are subsumed.

Chapter 5

Comparison and Experimental results

Currently, a prototype of the forward inverse method for the Horn fragment has been implemented. In this section we compare our inverse method to the tabled engine. The two examples used are computation of the k^th Fibonacci number and parsing a string with n tokens. These examples allow us to realize and understand our limitations and suggest improvements. The specifications of the machine that ran these examples are: Intel Pentium 4 3.4GHz CPU, 4GB of RAM. We are running SML New Jersey 110.55 under Linux version 2.6.14-gentoo-r3. Time is measured in second(s) and ∞ represents that the run time is over 30 min and we have terminated the process.

5.1 Fibonacci example

Theoretically, to calculate each Fibonacci number, we need to calculate the previous two Fibonacci numbers. Using depth first search to find a Fibonacci number, we will have an exponential algorithm due to the recomputation of previously encountered goals. However, using the inverse method, we begin with the base cases (Fibonacci of 0 and 1) and deduce more Fibonacci numbers as we search. By constructing each Fibonacci number, starting from fib(0), fib(1), fib(2), . . ., we do not repeat subcomputations and in theory, we have a linear time algorithm. The tabled engine in Twelf provides us a loop detection mechanism. In suspending search and using memoization, we are able to reuse previously computed goals. In the table below, the first column is input of Fibonacci, the second column is the result of the calculation, column Facts is sum of facts generated from type family `add` and type family `fib`. The columns IR time and IF time are the run times via inverse method using activation of rules and activation of facts, respectively. IF rules show the number of rules generated for activation of rules. For the tabling engine, which is denoted by Tab, not all type families have to be stored in the table. The first column shows the run time for tabling, Time in Tab are the run times with all type families stored and the value in parentheses are run times with only the type family fib tabled. The last column shows the number of entries in the table for all type families tabled.

k	Facts (add + fib)	IR Time	IF Time	Tab Time
14	377 + 14	1.48	2.75	0.46 (0.08)
15	610 + 15	4.41	102.37	1.210 (0.07)
16	987 + 16	11.19	∞	3.135 (0.10)
17	1597 + 17	34.10	∞	6.861 (0.10)
18	2584 + 18	193.79	∞	139.826 (0.16)

k	Facts (add + fib)	IR rules	IF rules	Tab ♯ Entries
14	377 + 14	5	2071	403
15	610 + 15	5	3554	638
16	987 + 16	5	n/a	1017
17	1597 + 17	5	n/a	1629
18	2584 + 18	5	n/a	2618

The run times for k $<$ 14 are not interesting because they are much less than a second, taking into account the skew of data, we choose not to discuss them. The number of rules generated for IR is not listed because it is always constant for the same signature, in this case it is five. The number of rules generated in the IF loop reaches thousands for k = 14. Since the number of rules does not decrease, there is a considerable amount of overhead keeping track of these rules and searching over them. Tabling outperforms both loops even if we table all type families. Also, tabling only the type family `fib` yield a significant advantage compared to tabling all type families.

5.2 Parsing

Parsing examples allow us to mix left and right recursive programs together to model associativity of connectives such as conjunction, disjunction, and implication. Clauses for conjunction and disjunction are left associative while implications are right associative. This program is not executable via depth first search if we do not keep track of previous subcomputations. For parsing, we compare the two looping structures for the inverse method to tabling. IR and IF still represent activation of rules and facts in the inverse method and Tab stands for tabling. The time column shows the run time required for each engine, the ♯ facts columns in IR and IF give us the number of facts generated. Number of entries in the table for the tabled engine is displayed next to its run time.

	IR		IF		Tab	
tokens	time	♯facts	time	♯fact	time	♯entries
3	0.094	42	0.110	198	0.031	5
5	0.860	138	0.109	2214	0.016	6
7	1.359	138	29.828	3702	0.015	10
9	1.016	138	33.391	3846	0.032	10
11	∞		∞		0.171	18

While the number of rules generated in this example for IR is not as substantial as Fibonacci, it is still quite large compared to the fixed number of rules in IF. The examples to execute were chosen randomly. However, by construction of the inverse method, it generates all possible facts until a solution is found. Since there are many type families, we produce facts of each type family in the order the rules are structured. After inspecting the facts generated, we see that all possible combinations of literal tokens and connective tokens for each type family are generated in the corresponding order. Hence it is possible that we generate facts with n connectives for one type family A_1 and facts with only n-1 connectives for another type family A_2. If a fact in A_2 is a solution to the query, then we have generated redundant facts from type A_1. Still, IF performs better than IR but does not perform as well as tabling.

5.3 Unification failures

To understand better the inverse method, we look at the heart of proof search: unification. In the implementation chapter, we see that unification is used extensively in axiom generation, satisfying premises and instance of solution checks. In the tabled engine, occurs checks are factored out by linearized terms[18]. This way, all unification calls are on linearized terms and an efficient assignment algorithm can be used. On the other hand, the inverse method does not perform any optimizations to higher order unification. Since unification lies at the heart of proof search, we decided to measure the number of unification failures for each of the implementations and compare them to the numbers from tabling.

	IR			IF			Tab
example	Sub	IoS	solve	Sub	IoS	solve	solve
fib 14	10983	166	27989	21380	322	164129	416
fib 15	28086	256	51169	55230	502	443397	1068
fib 16	72526	401	93343	∞	∞	∞	2100
parse 3	95	42	3351	78586	630	2496	9
parse 5	6363	138	21984	1982182	2910	9873	24
parse 7	458783	906	238087	∞	∞	∞	43

We show the number of unification failures for the Fibonacci examples and parsing examples. IR and IF are still the inverse method with activation of rules and facts respectively. Columns Sub illustrate the number of checks for subsumption. IoS columns illustrate the number to check whether a fact is an instance of the solution to the query. $Solve$ give us the number of unification failures used in proof search solving the query. The symbol ∞ still represents that we have run the search for more than 30 min and have terminated the process. The results show that unification plays an important role in proof search, it is required in axiom generation, satisfying premises of rules, subsumption checking etc. There are two kinds of subsumption checking, one is where we check that a derived fact is an instance of a previously derived fact; the other is where we check whether a derived fact unifies with the query. It is not enough to just check whether a fact is an instance of the query, it is possible we derive a fact that is more general than the query. Nor it is enough to check whether the derived fact and the query are instances of each other, it is possible that we derive a fact that is partially ground and the query is partially ground but they do not share the ground variables. In this case, we still have a solution but need to check via unification. In comparison, we see that the IR loop has less unification failures than the IF loop. However, compared to tabling, both loops have a substantially more unification failures. For tabling, unification is handled by an assignment algorithm where for the inverse method, we use higher order unification. These numbers reflect the importance of unification in the inverse method. The great number of unification failures is partially due to the fact that we do not solve subgoals in a particular order. For each use of unification, if we ordered the subgoals, it is possible that the terms are partially instantiated and we do not have more general terms. These more general terms require the further use of unification to be ground. Ordering of subgoals is briefly discussed in the future works section.

Chapter 6

Conclusions

6.1 Future works

Currently, our implementation for the inverse method prover is for the Horn fragment only. Also, the performance of both loops does not compare to tabling. Further, the inverse method does not yet give a decision procedure, for example, given a string of tokens, if it can be parsed, then the prover will produce the parsed expression, but if this string of tokens cannot be parsed, the inverse prover will search produce possible token strings with larger size. This chapter will explain some of the future works that can be done.

6.1.1 Extension to HHF

As explained in the theoretical foundations chapter, hereditary Harrop formulas is an extension to Horn clauses. The depth first search engine in Twelf is implemented for hereditary Harrop formulas. The current implementation of the inverse method lacks expressive power and dynamic clauses. Here we only give a brief overview of possible extensions. The subformula generation algorithm already works for hereditary Harrop formulas, currently we have thought about axiom generation and dynamic clauses.

For axiom generation, the use of contextual modal types will facilitate both theory and practice for an inverse method prover. However, subformulas will not only have a context that keeps track of meta-variables, they will also have a context that keeps track of parameters. In the dependently typed system, the dependency of types and type variables require that the context that declares these types be an ordered context. Since the order of the dependency of types cannot be changed, this ordering of variables in the context cannot be violated. We use the notion of *block contexts* to eliminate the factoring contexts.

6.1.2 Impose ordering on subgoals

From trial and error, we find that for the inverse method, the order of the subgoals does not matter. Since only atomic subformulas are generated and used to create axioms, the order of the subgoals does not matter with respect to axiom generation. Also, when creating and using the rules generated from the signature, we satisfy the premises without discretion to order. While backward search allows us to proceed searching if we have satisfied all previous subgoals, the inverse method does not have these requirements. One of the problems with inverse method with activation of facts is the generation of dead rules. These dead rules have premises that can never be satisfied. In backward search, when the subgoals are executed in order, some terms are partially instantiated. However, in forward search, these terms are not necessarily instantiated and when we unify these terms with some facts, the new terms might not contradict possible instantiations of previous subgoals. Now for the sequence of subgoals, we can impose an ordering such that certain subgoals should only be solved after other ones. To incorporate an ordering for subgoals, we turn to mode information. Modes information give rise to the flow of information through the subgoals. In differentiating input and output arguments for a term, we allow the information to flow in a single direction toward the goal. We use this flow of information to define our ordering of subgoals.

6.2 Conclusions

In this thesis we presented theoretical foundations along with implementation details of an inverse method prover. We described a forward uniform proof calculus with hereditary substitutions then lifted it to higher order. This forward system eliminates the necessity of loop detection in a backward setting. While the inverse method prover is specific to LF, which is the underlying framework for Twelf, it seems possible to apply a similar reasoning to systems such as λProlog[9] or Isabelle[13]. Also, we provided soundness and completeness proofs for the forward uniform proof calculus. After a detailed description of implementation of the inverse method prover, we provided comparative data to show that our prover requires optimization and extension to compare to the results from tabling.

Bibliography

[1] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: the POPLmark Challenge. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Oxford, UK, August 22-25*, volume 3603 of *Lecture Notes in Computer Science(LNCS)*, pages 50–65. Springer, 2005.

[2] Kaustuv Chaudhuri. The focused inverse method for linear logic. *Technical Report CMU-CS-05-106, Carnegie Mellon University (2005)*, (CMU-CS-06-162), 2006.

[3] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. In *Proceedings of the Third International Joint Conference on Automated Reasoning, Lecture Notes in Artificial Intelligence (LNAI)*, pages 97–111, Seattle USA, 2006. Springer-Verlag.

[4] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In F. Baader, editor, *19th International Conference on Automated Deduction (CADE'03), Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 106–120. Springer, 2003.

[5] Anatoli Degtyarev and Andrei Voronkov. The inverse method. In *Handbook of Automated Reasoning*, pages 179–272. 2001.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery (J. ACM)*, 40(1):143–184, 1993.

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

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

[9] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, 1988. MIT Press.

[10] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus – a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Con-*

ference on Automated Deduction (CADE'99), pages 287–291, Trento, Italy, 1999. Springer-Verlag LNCS.

[11] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2006 (to appear).

[12] George C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, 1997.

[13] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3(3):237–258, 1986.

[14] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE'99)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.

[15] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer, 2002.

[16] Brigitte Pientka. *Tabled higher-order logic programming.* PhD thesis, Department of Computer Sciences, Carnegie Mellon University, 2003.

[17] Brigitte Pientka, Xi Li, and Florent Pompigne. Focusing the inverse method for LF: a preliminary report. *Electronic Notes for Theoretical Computer Science*, 196:95–112, 2008.

[18] Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, Florida*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer, 2003.

[19] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.