ASPECTS OF THE COMPUTATIONAL CONTENT OF PROOFS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Judith Lynne Underwood

August 1994

ASPECTS OF THE COMPUTATIONAL CONTENT OF PROOFS

Judith Lynne Underwood, Ph.D.

Cornell University 1994

In this thesis, we explore three aspects of the computational content of proofs. These are: a computational interpretation of the metatheory of intuitionistic propositional logic, an extension of this approach to intuitionistic predicate logic, and a computational interpretation of classical sequent proofs.

We begin with a study of the computational aspects of validity, provability, and completeness for intuitionistic propositional logic. We give a constructive proof of completeness of Kripke models for intuitionistic propositional calculus, such that the computational content of the proof is a form of the tableau algorithm. Since the evidence for provability we construct is actually a term in typed λ-calculus, we can interpret this result as a formal relationship between Kripke semantics and realizability semantics. We also show how a formal proof of the completeness theorem in Nuprl could be used to add the tableau decision procedure for provability to Nuprl's collection of proof techniques via reflection.

We then explore how these results could be generalized to the metatheory of intuitionistic predicate logic. To do this, we develop some machinery for representing

infinite Kripke models in type theory using co-inductive types. Although provability in intuitionistic predicate logic is undecidable, we can still prove a constructive theorem in the metatheory which has computational content. We also show that using classical logic in part of the proof does not destroy the computational content of the proof.

Finally, we examine the computational aspects of classical sequent proofs. We show how the interpretation of nonlocal control operators as the computational content of classical axioms allows us to prove a completeness result for intuitionistic predicate logic with a depth-first-search tableau procedure as the computational content of the proof. Using the nonlocal control operator *call/cc* as the the computational interpretation of the classical axiom $((\varphi \to \bot) \to \varphi) \to \varphi$, we define a method for extracting programs from classical sequent proofs in propositional logic. We give a proof-theoretic account of continuations and their behaviour in this context, and prove a number of properties of the extraction.

Biographical Sketch

Judith Underwood was born in Walla Walla, Washington in 1966. She graduated from Walla Walla High School in 1984. In 1988 she received a Bachelor of Arts degree from Oberlin College with a double major in Mathematics and Computer Science and with high honors in mathematics. In 1990 she received a Master of Science degree in Computer Science from Cornell University. She has had a number of jobs, including cafeteria server, asparagus packer, and inspector of peas.

To my parents.

Acknowledgements

> A project of this scope could not be realized without the aid of many people, or rather it *could*, but it would be dumb to do it that way when there are so many people around willing to give their aid.
>
> – Peter Schickele, "The Definitive Biography of P. D. Q. Bach (1807-1742?)"

I am indebted to my advisor, Robert Constable, for introducing me to this research area, and for much guidance and understanding.

The National Science Foundation supported me through a graduate fellowship, as did the Mathematical Sciences Institute, for whom I must say:

> "This research was supported in part by the United States Army Research Office through the Army Center of Excellence for Symbolic Methods in Algorithmic Mathematics (ACSyAM), Mathematical Sciences Institute of Cornell University. Contract DAAL 03-91-C-0027."

This research was also supported in part by EPSRC grant GR/H73103, and I thank Don Sannella for allowing me to finish my thesis under this grant.

My friends and peers in the Prl group, Bill Aitken and Paul Jackson, have always been tolerant of my ignorance of Nuprl's implementation matters, for which I am grateful.

I have been fortunate in having a wide circle of friends, including the stalwarts of the lunch and AI crowd, and especially Russell Brown and William Rucklidge. I owe particular thanks to Jim Jennings, and to my officemate James Allan, for putting up with me during the final stages of thesis production.

Outside Cornell, I thank the chancel choir at St. Paul's United Methodist Church for a twice-weekly period of transcendence, and the old Tuesday afternoon crew at Loaves & Fishes, especially Dan Ralph.

I owe inexpressible thanks to my parents and my sister, Deborah, for their unstinting support and encouragement, and for always believing in me.

Finally, I am forever grateful for the love and support of Ian Gent.

Table of Contents

Chapter 1

Introduction

There is a clear analogy between the process of writing programs and the process of proving theorems. Generally speaking, to write a program, the programmer decides what the program must do, chooses an algorithm, and breaks the algorithm down into small computational steps until the program compiles. To prove a theorem, a mathematician formulates a statement of the theorem, chooses a proof technique, and breaks the argument down into small logical steps until the proof is accepted by the referees.

This analogy can be made more formal in a number of ways. Perhaps the most fundamental is the Curry-Howard isomorphism, (presented in [How80],but connected with earlier work, e.g. [deB70]), which shows that there is an isomorphism between propositions in intuitionistic propositional calculus and types in simply typed lambda calculus. This amounts to a formal correspondence between "what the program must do" (as expressed by its type) and "what the theorem proves," for a simple programming language and a simple logic. Later, this was extended to an isomorphism between proofs in natural deduction for intuitionistic propositional calculus and closed terms in simply typed lambda calculus [Con71], which formalizes the correspondence between proofs and programs. This leads to the slogans "propositions as types" and "proofs as programs". Many proof systems have been built on this foundation, including [vD80,ML82,C$^+$86,CH88,Pol90], and many proof systems [Gor85,ORS92,Pau88] can support reasoning in this style.

In this thesis, we explore three major aspects of the correspondence between proofs and programs. First, we approach the analogy from the opposite side and consider programs as proofs. In particular, we interpret the tableau decision procedure for intuitionistic propositional calculus as a proof. The result is a constructive proof of the completeness of Kripke models for intuitionistic propositional calculus. This approach has several interesting properties, including an application to extending Nuprl with verified decision procedures.

We then consider how the result can be extended to the more general tableau method for the predicate calculus. We cannot expect to get a decision procedure, since provability in intuitionistic predicate logic is undecidable. However, we can still obtain a description of the tableau method from a more restricted proof.

These computational metatheorems can be interpreted as a means of relating the two main approaches to the semantics of intuitionistic logic. The first is a computational approach, represented here by type theory but which can take many other forms [Kle52]. The other is a model theoretic approach, particularly the model theories of Beth [Bet56] and Kripke [Kri65]. In the first half of this thesis, we show how to represent Kripke semantics in type theory. Many others have treated this comparison in different ways; see, e.g. [Hyl82,TvD88a,Lip90b,Lip90c, Lau70].

The consideration of the computational treatment of undecidable problems such as provability in intuitionistic predicate calculus leads to the study of classical logic and the work of Griffin [Gri90] and Murthy [Mur90,Mur91b,Mur91a] on the classical typing of control operators. We re-examine this work in the context of sequent calculus, and give a method for extracting computations from classical sequent proofs. This leads to a proof-theoretic account of the behavior of continuations. We finish with some normalization results for the terms extracted from sequent proofs.

The area is still rich in further work. Computer science has generated many new logics deserving of study with regard to their computational content, as well as motivating the study of other logics from the computational point of view. There are many approaches to the problem of representing the computational content of nonconstructive logics, generally associated with a different philosophical justifications. The implications of these methods for computer science have not been adequately investigated.

1.1 Basics

In the remainder of this chapter, we review some of the basic principles underlying this work. Good introductions to this material from the logical side are [NS94] and [Dra88]. A treatment with more emphasis on computation and types is [GTL89]. Kleene's book [Kle52] contains a description of realizability, a computational treatment of intuitionistic logic different from that of type theory.

For computer scientists, perhaps the most compelling justification for intuitionistic logic is its computational value. Given the initial observation that programs and proofs are similar sorts of things, it is natural to want to make this idea more formal, and determine what kinds of proofs correspond to programs and what kinds of programs these are. Computer scientists have already developed a well-understood model of computation, namely, Turing machines (or any equivalent system). Hence we would expect our programs to represent Turing computable functions. However, there are a number of other philosophical justifications for intuitionism, with motivations which may be incompatible with Turing computability. As a result, there are a number of varieties of constructive mathematics (mathematical systems built on an intuitionistic logic). Here we restrict attention to a logical core on which nearly all varieties of intuitionism agree; in chapter 3, we shall consider one extension to the logic which is used in some metatheories. Further explication of the various philosophical schools can be found in [Bee85].

In a type theoretic approach to intuitionistic logic, propositions are types, and programs of a particular type represent proofs of the corresponding proposition. The programming language for the core logic, intuitionistic propositional calculus, is based on λ-calculus, augmented with operators to deal with pairing and disjoint union types. However, the fundamental aspects of the proofs-as-programs correspondence can be described in simply typed λ-calculus, where the only operations are λ abstraction and function application.

1.1.1 Implicational calculus

We consider the following proof system, which is a subsystem of intuitionistic propositional logic. Formulas are constructed from propositional variables in a set

PropVar according to the grammar

$$\begin{array}{rcll} A & = & x & (x \in \mathit{PropVar}) \\ & | & A_1 \rightarrow A_2 & \end{array}$$

In each rule, the construction

$$\Gamma \vdash A$$

is called a *sequent*, where Γ is a set of formulas (the *hypotheses*) and A is a single formula (the *conclusion*). This sequent can be thought of as stating, "Under the set of assumptions Γ, we can conclude A."

Axiom The axioms of the system are sequents of the form $\Gamma, A \vdash A$.

→-introduction Given a sequent of the form $\Gamma, A \vdash B$, we can conclude the sequent $\Gamma \vdash A \rightarrow B$. We say the hypothesis A has been *discharged.* This rule may also be written

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$$

→-elimination (modus ponens) If we have sequents of the form $\Gamma \vdash A \rightarrow B$ and $\Gamma \vdash A$, then we can conclude the sequent $\Gamma \vdash B$. This rule is written

$$\frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

A *proof tree* is a tree formed by application of these rules to axioms, which form the leaves of the tree. A proof tree in which the final sequent is of the form

$$\vdash A$$

(i.e. where Γ is empty) is a *proof* of A. An example of a proof is

$$\dfrac{\dfrac{A, B \vdash A}{A \vdash B \to A}}{\vdash A \to (B \to A)}$$

1.1.2 Typed λ-calculus

We can interpret the formulas in the implicational calculus as describing types in simply typed λ-calculus. Proof trees will then correspond to terms of in the calculus.

More formally, we show how to build terms in the typed λ-calculus in the same way as proofs are built. We assume we have a collection *TermVar* of term variables $x, y, \ldots$, each associated with a type, i.e. a formula as described above, built from base types which are the propositional variables.

We begin with the axioms in the proof system. With each formula in a sequent

$$A_1, \ldots, A_n \vdash A_i$$

we associate a term variable of that type, i.e.

$$x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i.$$

The sequent is now read as "Under the typing assumptions $x_1 : A_1 \ldots x_n : A_n$, conclude that x_i has type A_i."

The $\to$-introduction rule corresponds to a typing rule for λ-abstractions.

$$\frac{x_1\!:\!G_1, \ldots, x_n\!:\!G_n, y\!:\!A \vdash t\!:\!B}{x_1\!:\!G_1, \ldots, x_n\!:\!G_n \vdash \lambda y.t\!:\!A\to B}$$

Finally, the $\to$-elimination rule corresponds to a typing rule for function applications.

$$\frac{x_1\!:\!G_1, \ldots, x_n\!:\!G_n \vdash t\!:\!A\to B \qquad x_1\!:\!G_1, \ldots, x_n\!:\!G_n \vdash a\!:\!A}{x_1\!:\!G_1, \ldots, x_n\!:\!G_n \vdash t(a)\!:\!B}$$

Just as a term of type $A\to B$ is a function from terms of type A to terms of type B, we see that the modus ponens rule explains a proof of $A\to B$ as a function from proofs of A to proofs of B.

In each of these cases, we read the final sequent as describing the type of the term in the conclusion, given the typing assumptions for the variables listed in the hypothesis. Note that because proofs begin with axioms, all free variables in the term in the conclusion are in the hypothesis list. A proof $\vdash A$, then, can be interpreted as constructing a term t which has type A under an empty set of typing assumptions. In other words, the term has no free variables.

For example, the proof above becomes

$$\frac{\dfrac{x\!:\!A, y\!:\!B \vdash x\!:\!A}{x\!:\!A \vdash \lambda y.x\!:\!B\to A}}{\vdash \lambda x.\lambda y.x\!:\!A\to(B\to A)}$$

Thus, this proof can be represented by the term

$$\lambda x^A.\lambda y^B.x^A$$

of type $A \rightarrow (B \rightarrow A)$. In general, we can consider a proof of φ in the implicational calculus described above as defining a closed term in the simply typed λ calculus which has type φ. Furthermore, the extraction of the term from the proof is completely automatic.

To carry this construction of terms from proofs through to intuitionistic propositional calculus, we first add rules for introduction and elimination of conjunction $\wedge$ and disjunction $\vee$ to the proof system. We then add operators to typed λ-calculus to represent these rules. The operators for $\wedge$ are a pairing operator and an unpairing operator, and the operators for $\vee$ are a tagging mechanism for describing elements of a disjoint union, and an operator for deciding which component of a disjoint union is given by an element of the disjoint union type.

Both the logic and the programming language presented here are very simple; however, these ideas can be generalized to much richer type theories including dependent types, inductive types, and type universes representing the "type of types" in a limited way. Nuprl is one such type theory; a detailed explanation of it can be found in [$\mathrm{C}^{+}86$]. An introduction to the use of Nuprl for reasoning about proofs is [CH90]; a more technical treatment is in [BC93].

1.1.3 Semantics

This extraction of programs from proofs gives us something we may call an operational semantics of proofs in intuitionistic logic. If proofs are programs, they can be reduced and normalized like programs. It is then reasonable to ask if there is a denotational semantics of proofs. In logic, however, the emphasis is mainly on *provability*, rather than on the structure of proofs. The result has been a great deal of work on mathematical representations of provability and truth, and somewhat less work on proof theory. A common approach to a mathematical representation of provability is through an algebraic model theory for a logic, that is, demonstrating a correspondence between provability in the logic and some property of an algebraic structure based on the formulas in the logic. This corresponds more closely to a denotational semantics of formulas, rather than of proofs.

For example, consider classical propositional logic. One model theory arises from the consideration of *truth sets* (studied in [Smu68]) – sets of formulas given the value *true* under a particular assignment of truth values (true or false) to variables. If a formula is in all truth sets, it is classically true, since it is true under every assignment of truth values to its variables.

Intuitionistic logic is more complex, and so are the model theories proposed for it. In chapter 2, we examine one model theory due to Kripke [Kri65], and show that it is *complete*; i.e. that every formula which is true in all Kripke models is intuitionistically provable. Furthermore, we prove this theorem in a constructive metatheory, with an emphasis on the resulting computational content. In effect, we give a computational account of the model theory, representing the "denotational semantics" in the "operational semantics".

Chapter 2

Constructive Metatheory and Decision Procedures for Intuitionistic Propositional Calculus

In this chapter, we shall present a constructive proof of the decidability of validity in the intuitionistic propositional calculus. The proof can also be interpreted as a constructive proof of completeness of Kripke models for the logic. The computational content of the proof is a form of the tableau decision procedure for provability. If a formula is valid, the algorithm produces a proof of the formula in the form of an inhabitant of the corresponding type; if not, it produces a Kripke model and a state in the model such that the formula is not forced at that state in that model. The proof is described in some detail, and in such a way that the theorem can easily be formalized in Nuprl. The chapter concludes with a discussion of an application of this technique to the development of verified decision procedures in Nuprl.

2.1 Preliminaries

In this section, we introduce the main ideas of this chapter. We begin with an outline of the motivations of this work. This leads to a discussion of the possible statements of the theorem we will prove and their various interpretations. Finally, we give some definitions necessary for the statement of the theorem, and we state the simplest version of the theorem.

2.1.1 Motivation

Since the completeness of Kripke models for intuitionistic propositional calculus is not a new result ([Kri65]) and since the tableau algorithm is also well-known (a history is given in [Fit83], tracing the algorithm to Beth [Bet56]), some explanation of the goal of this work is necessary. The most important aspect of the result is in the properties and interpretations of the proof of the main theorem (theorem 2.2.1).

First, the development of the proof was directly influenced by the goal of extracting a specific decision procedure as the computational content of the proof. Despite this, the proof itself is very natural, and resembles earlier proofs of the same result [Fit69]. The proof is expressed sufficiently rigorously that it is clear it can be formalized in Nuprl; further details of the formalization can be found (in a different context) in [ACU].

Secondly, the description of the tableau algorithm which arises from the proof is easy to understand, and in particular (and unlike many direct presentations of the algorithm) it is very easy to see that the decision procedure terminates. The algorithm has time complexity exponential in a polynomial of the length of the formula on which it is run; the problem is known to be PSPACE-complete [Sta79].

The proof can be interpreted in a number of ways. It was originally designed as a constructive completeness proof [Und90]. As such, it is unusual in that it requires that evidence be supplied in either case of the result of the decision procedure. For any formula, the algorithm produces either a term in the type corresponding to the formula, or a Kripke model in which the formula does not hold. In particular, the counterexample is created from the failed tableau proof; in a sense, they have the same "data structure". Kripke models can then be understood as evidence of unprovability of particular formulas, rather than simply some kind of abstract algebraic structure.

In the previous work [Und90], the issue of the choice of proof system for intuitionistic propositional logic was not explicitly dealt with. The Curry-Howard isomorphism was used implicitly to derive a proof from a typed lambda term t. However, other interpretations of this evidence are possible. For example, we can interpret the term t as a realizer for the formula, i.e. semantic evidence in type theory. Under this interpretation, we have a constructive theorem that for any propositional formula, either there is a Kripke countermodel for the formula or the formula is valid in type theory. This gives a constructive method for comparing Kripke semantics with other semantics. A thorough investigation of along these lines can be found in [Lip90a].

Finally, this approach leads to an application of metatheory to problems in automated theorem proving. If theorem 2.2.1 is proved within a system which allows extraction of the computational content of constructive theorems, the result is the automatic generation of a decision procedure for validity of propositional formulas in intuitionistic propositional calculus. Furthermore, if the system is strong enough to describe itself and its own proofs (as with Nuprl's reflection mechanism), then the proof of a suitable restatement of the completeness theorem with respect to the described proof system will generate a verified decision procedure for validity. This provides a principled method for extending the reasoning power of the system. The final section of this chapter will address this application in more detail.

2.1.2 Interpretations of the theorem

The original motivation for the main theorem of this chapter was to produce a constructive proof of the completeness of Kripke models for intuitionistic propositional logic. The usual statement of the completeness theorem ("If a formula is true in all Kripke models, it is provable") may be interesting in the context of proof theory, but it is computationally almost useless. When searching for a proof of a formula φ, it is generally not easier to find a proof that φ is true in all Kripke models, so that version of the theorem is not helpful. In order to obtain the tableau algorithm as the computational content of the proof of completeness, the theorem is restated in the form "For any formula, either there is a proof of the formula or there is a Kripke model in which the formula is not forced." Thus, the computational content of a proof of this statement is an algorithm which, given a formula φ as input, returns either a proof of φ or a Kripke model which fails to force φ.

To formalize this completeness theorem, we must define the notion of proof of a formula φ and countermodel for φ. In the original paper on which this chapter is based ([Und90], also in [Und93a]), the proof of φ was presented as a closed term t in typed lambda calculus, and an appeal was made to the Curry-Howard isomorphism as justification. However, a closed term t of type φ need not be considered as a proof. Depending upon how we choose to interpret t, we may use the same argument as a proof for several theorems related to completeness.

For example, we can interpret t as a semantic object which provides evidence for the validity of the formula. This approach leaves open the exact details of the semantical system, requiring only that t be evidence for the validity of φ in the semantics. Then the theorem has the form, "For any formula, either the formula is valid or there is a Kripke countermodel." Note, however, that without knowing more about the semantics we cannot conclude that if we have a Kripke model in which φ is not forced, then φ is invalid.

Similarly, from this theorem alone we cannot conclude that φ is not provable if we have a Kripke model in which φ is not forced. If the proof system is simply natural deduction for intuitionistic propositional logic, then to conclude this we need only show that Kripke models are sound, i.e. if a formula is provable, it is true in all Kripke models. The proof of this is easy and not computationally interesting, so we do not formalize it here.

For other proof systems which subsume intuitionistic propositional logic, we can still show that, for any formula φ, there is either a Kripke countermodel for φ or there is a proof of φ using only propositional rules in the system. This is the approach taken in [ACU], where the proof system is a subsystem of Nuprl's proofs.

If the proof system is stronger than propositional logic, then Kripke models may not be complete, and the theorem must be interpreted as a proof of decidability of Kripke validity. However, if we also know that the proof system is a conservative extension of intuitionistic propositional calculus, i.e. if every propositional formula provable in the system has a proof using only rules of propositional logic, then we can conclude that a countermodel is evidence of unprovability.

However, conservative extension results can be difficult to prove. It is not known whether Nuprl's type theory is a conservative extension of intuitionistic propositional logic, though it is difficult to see how it could not be. Evidence for this is found in [KMO91], which is based on a novel approach to realizability due to Läuchli [Lau70]. For the record, we state

Conjecture 1 *Nuprl's type theory is a conservative extension of intuitionistic propositional logic.*

2.1.3 Definitions and first statement of the theorem

In the remainder of this section, we consider only the completeness interpretation of the theorem, i.e. the statement "For any formula, either there is a proof of the formula or there is a Kripke model in which the formula is not forced". To give a formal statement of this, we must first formally define the mathematical objects used by the theorem. Since we wish to show how the theorem could be formalized in Nuprl, we use a logical notation for Nuprl's type theory in our definitions. However, no knowledge of the details of Nuprl's type theory will be necessary to understand the definitions. We assume we have *Atom* as the type of atomic formulas, and *Wff* as the type of formulas.

The first problem is to translate the mathematical definition of Kripke model validity into type theory. We choose a very general definition of Kripke model (from [Fit69]), and modify only slightly the usual definition of forcing. These definitions allow a completely constructive treatment of Kripke models.

In mathematical terms, a Kripke model is a triple consisting of a set, a transitive and reflexive relation on that set, and a monotone function from the set to the set of atomic formulas. The set can be thought of as "states of knowledge"; the relation, which is not quite a partial order, then describes increasing information, as the set of atomic formulas associated with each state is considered to be the atomic formulas known to be true at that state.

To formalize this, we represent the set of states as a type T, so that elements of the type are the states. Nuprl has a hierarchy of type universes, so that it is possible to give a "type of types" in a limited way. Since it is not necessary to fix the universe level precisely, the universe type is given as U_i.

The relation on the set of states is represented as a function R from pairs of states to the type *bool*, so that the value of the function is *true* when the pair is in the relation. The type of R is restricted (by use of Nuprl's set type) to only reflexive and transitive relations. The atomic formulas forced at each state are described by a function *af* ("atomic forcing") on states and atomic formulas which is true when the formula is forced at that state. The type of this function is similarly restricted so that the relation is monotone with respect to R.

Since the type of R depends on the type T representing the set of states, and since the type of *af* depends on both T and R, the type of Kripke models is a three-part dependent product.

$$
\begin{aligned}
&\mathit{Kripke\_model} = \\
&\qquad T\!:\!U_i \\
&\quad\times\quad R\!:\!\{R\!:\!T \times T \rightarrow \mathit{bool} \mid \forall a,b,c\!:\!T.R(a,a)\&(R(a,b)\&R(b,c) \rightarrow R(a,c))\} \\
&\quad\times\quad \mathit{af}\!:\!\{\mathit{af}\!:\!T \rightarrow \mathit{Atom} \rightarrow \mathit{bool} \mid \forall a\!:\!T, v\!:\!\mathit{Atom}.\mathit{af}(a,v) \rightarrow \forall b\!:\!T.R(a,b) \rightarrow \mathit{af}(b,v)\}
\end{aligned}
$$

If K has type $\mathit{Kripke\_model}$, we use the notation $K.T$, $K.R$, and $K.\mathit{af}$ to refer to the components of K. We may omit the reference to K if it is clear from context.

Given this definition of the type of Kripke models, we must describe when a formula is forced in a model. The definition of forcing is by induction on the structure of the formula. Given af, which describes which atomic formulas are forced at each state, we define the nonatomic formulas which are forced at that state by a function *forces* of type

$$\mathit{forces} : K\!:\!\mathit{Kripke\_model} \rightarrow K.T \rightarrow \mathit{Wff} \rightarrow U_i$$

To avoid the intuitionistically troublesome use of negation in the usual definition of forcing, we define at the same time (i.e. by mutual recursion) a function *notforces* of the same type. We wish these to mean, "state s in Kripke model K forces formula α if and only if $\mathit{forces}(K)(s)(\alpha)$ is an inhabited type," and "state s in Kripke model K does not force formula α if and only if $\mathit{notforces}(K)(s)(\alpha)$ is an inhabited type." When explicit currying is not necessary, we write $\mathit{forces}(K, s, \alpha)$ for $\mathit{forces}(K)(s)(\alpha)$.

In mathematical notation, these definitions are:

$$
\begin{aligned}
\mathit{forces}(K,s,\alpha) \quad = \quad & K.\mathit{af}(s,\alpha) && (\alpha \in \mathit{Atom}) \\
& \mathit{forces}(K,s,\beta) \wedge \mathit{forces}(K,s,\gamma) && (\alpha = \beta \wedge \gamma) \\
& \mathit{forces}(K,s,\beta) \vee \mathit{forces}(K,s,\gamma) && (\alpha = \beta \vee \gamma) \\
& \forall s'.sRs' \rightarrow \\
& \mathit{notforces}(K,s',\beta) \vee \mathit{forces}(K,s',\gamma) && (\alpha = \beta \rightarrow \gamma)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{notforces}(K,s,\alpha) \quad = \quad & \neg K.\mathit{af}(s,\alpha) && (\alpha \in \mathit{Atom}) \\
& \mathit{notforces}(K,s,\beta) \vee \mathit{notforces}(K,s,\gamma) && (\alpha = \beta \wedge \gamma) \\
& \mathit{notforces}(K,s,\beta) \wedge \mathit{notforces}(K,s,\gamma) && (\alpha = \beta \vee \gamma) \\
& \exists s'.sRs' \wedge \\
& \mathit{forces}(K,s',\beta) \wedge \mathit{notforces}(K,s',\gamma) && (\alpha = \beta \rightarrow \gamma)
\end{aligned}
$$

Note that if K has only one state, these rules amount to the definition of classical truth values for formulas based on the truth values for atomic formulas assigned by af. That is, if K has one state s and we assign true to atomic formulas which are forced at s and false to all other formulas, the *forces* relation actual assigns a truth value to formulas based on the classical rules. See [CH90] for details of the classical case.

In Nuprl, the mutual induction would be achieved by defining a function returning a pair p of values, so that $\mathit{forces}(K,s,\alpha) = \pi_1(p)$ and $\mathit{notforces}(K,s,\alpha) = \pi_2(p)$. The complete definition of these functions uses a *Wff* induction operator. Further details of one possible choice for this definition appear in [ACU].

Note that we can conclude $(\mathit{forces}(K,s,\alpha) \wedge \mathit{notforces}(K,s,\alpha)) \rightarrow \perp$ from these definitions. Hence, $\mathit{notforces}(K,s,\alpha)\rightarrow(\mathit{forces}(K,s,\alpha)\rightarrow\bot)$. However, we do not yet have that $\mathit{forces}(K,s,\alpha) \vee \mathit{notforces}(K,s,\alpha)$ is true constructively. We will be able to show this easily for the models which arise in the completeness proof.

Finally, a formula α is said to be *forced* in a Kripke model K if it is forced at every state in the model, i.e. if $\forall s\!:\!K.\mathit{forces}(K,s,\alpha)$. A model in which a particular formula is not forced is said to be a *countermodel* for that formula. More precisely, we say K is a countermodel for α if there exists s in K such that $\mathit{notforces}(K,s,\alpha)$ holds. If a countermodel K for a formula φ exists, we shall provide evidence for this by specifying a state s in the model such that $\mathit{notforces}(K,s,\alpha)$.

If a formula is provable, we provide a proof in the form of a typed lambda term inhabiting the corresponding type, following the Curry-Howard isomorphism. For simplicity, we use the same notation for propositions and types in this context.

We may now state the completeness theorem as we would like to use it. This ideal version of the completeness theorem is:

Theorem 2.1.1

$$
\begin{array}{l}
\forall\varphi : \mathit{Wff} \\
\quad (\varphi \\
\quad \vee \\
\quad (\exists K : \mathit{Kripke\_model}\ \exists s : K.T.\mathit{notforces}(K,\varphi,s)))
\end{array}
$$

Because we interpret this statement in type theory, the assertion φ means "there is a term of type φ". The assertion is proved by providing such a term.

We will actually prove a more general theorem (theorem 2.2.1), to which this theorem is a simple corollary. The more general statement is necessary in order to represent a stronger inductive hypothesis needed by the proof.

The proof of this theorem is designed so that its computational content is the tableau algorithm. Accordingly, the extra complexity in the inductive hypothesis can be most easily understood as representing information about the tableau. It will be helpful to review the tableau algorithm in order to understand the modifications necessary to the statement of the theorem and motivate the structure of the proof.

2.2 The tableau algorithm for intuitionistic propositional calculus

In this section, we give a general description of the tableau algorithm for intuitionistic propositional calculus, and define the structures necessary for the final statement of the completeness theorem. We then examine the tableau algorithm in more detail, in preparation for the proof of the algorithm in the next section.

In the classical tableau algorithm, given a formula φ, the algorithm first marks the formula as false, and then determining the consequences of this assumption for the truth or falsity of the subformulas of φ. For example, if $\varphi = \alpha \wedge \beta$ is false, then we must have α is false or β is false. The tableau then divides into two branches, with one branch containing the assumption that α is false, and the other the assumption that β is false. Each branch is then further developed according to the principal connectives in α and β. If every development of each of these branches leads to a contradiction (some formula is assumed both true and false), then the original formula φ must be true. On the other hand, if the development of the tableau completes and there is a branch (i.e. a single path from the original

φ to a leaf) which is not contradictory, it produces an assignment of truth values to variables under which the original formula is false.

The intuitionistic tableau algorithm is similar, but requires a notion of a state or world at which the formula is marked true or false. We refer to these worlds as *nodes* in the tableau. Formulas are thus labelled true or false at a particular node. A branch of the tableau describes a set of nodes along a single path in the development and the formulas labelled true or false at each node. A branch is closed if it contains a node at which the same formula is labelled both true and false.

To describe this, the actual statement of the theorem uses two new definitions, *Node* and *System*. A node is a pair of sets of formulas, analogous to a (multiple conclusion) sequent, and a system is a set of nodes, analogous to a (possibly incomplete) proof tree. In the proof, an element of *System* will correspond to a branch in the intuitionistic tableau, and an element of *Node* will describe which formulas are labelled true and which are labelled false at a particular state on a branch of the tableau. (Specifically, since a node is a pair of sets of formulas, the first set in the pair is the set of formulas labelled true, while the second set is the set of formulas labelled false.) A system is closed if it contains a node at which the same formula is labelled both true and false.

In the event that a countermodel is produced, it will be defined by the element of *System* corresponding to a branch of the tableau which fails to close. Nodes in this system will correspond to states in the countermodel, and will describe which formulas are forced and not forced at that state. This correspondence is made explicit by a function f which maps nodes to states in the Kripke countermodel.

To summarize the role of *Node* and *System*, we have the following table.

	φ provable	φ has a Kripke countermodel
Node	sequent	state
System	branch	model

We use the terms *Node* and *System* in the theorem because we do not know in advance if the object we are constructing is a proof or a countermodel. The theorem we will prove is stated for systems of nodes. As we intend to use it, these systems represent branches of a tableau development; however, the theorem is true for any such system.

More formally, we define

$$\begin{aligned} \mathit{Node} &= \text{list of } \mathit{Wff} \times \text{list of } \mathit{Wff} \\ \mathit{System} &= \text{list of } \mathit{Node} \end{aligned}$$

We use the list type as a convenience; these should be considered finite sets. The completeness theorem can now be stated:

Theorem 2.2.1

$$
\begin{aligned}
&\forall S : System. \\
&\quad (\exists \langle S_T, S_F\rangle \in S.(\bigwedge S_T \rightarrow \bigvee S_F)) \\
&\vee \\
&\quad (\exists K = (T, R, af) : Kripke\_model. \\
&\quad\quad \exists f : S \rightarrow T. \\
&\quad\quad\quad \forall \langle S_T, S_F\rangle \in S.(f(\langle S_T, S_F\rangle) = s \rightarrow \\
&\quad\quad\quad\quad (\forall \alpha \in S_T.forces(K, s, \alpha) \wedge \forall \alpha \in S_F.notforces(K, s, \alpha))))
\end{aligned}
$$

Here, $\bigwedge S_T$ denotes the conjunction of the formulas in S_T, and $\bigvee S_F$ denotes the disjunction of the formulas in S_F. This theorem says that for any system, either there is a node $\langle S_T, S_F\rangle$ in the system such that the conjunction of all formulas in S_T implies some formula in S_F, or, for every node $\langle S_T, S_F\rangle$, there is a Kripke model and a state in the model which forces all formulas in S_T and does not force (more precisely, *notforces*) every formula in S_F.

Once we have proved this theorem, it is easy to obtain theorem 2.1.1 as a corollary. Given a formula φ, apply the theorem to the system $S = \{\langle\emptyset, \{\varphi\}\rangle\}$. The result of applying the theorem is either a proof of φ or a model and a state in that model in which φ is not forced.

2.2.1 The algorithm

Since the proof is developed so that the computational content is the tableau algorithm, it will be helpful to study the algorithm in more detail in order to understand the structure of the proof.

The algorithm can be interpreted as a systematic search for a countermodel, such that failure to find a countermodel gives a proof. Given a formula φ, we begin the tableau with $0F\varphi$, which asserts that state 0 in some Kripke model does not force the formula φ. We then systematically specify the consequences of this assertion for the forcing of subformulas of φ according to their main connectives, adding new states and new branches if necessary. A new state is added only when we wish to describe the consequences of the assertion $pF\alpha\rightarrow\beta$, since for this to be the case we must have a state $p' \geq p$ such that $p'T\alpha$ and $p'F\beta$. If a branch ever contains the assertions $pF\alpha$ and $pT\alpha$ for some state p, then this is a contradiction and the branch is said to be *closed*. If every branch of a tableau is closed, the tableau is also said to be closed, and a cut-free sequent proof can be extracted from the result. If there is an open branch, it can be interpreted as the description of a Kripke model in which φ is not forced.

As an example, consider the following tableau for the formula $((A\rightarrow B)\rightarrow A)\rightarrow A$ (Peirce's law):

$$
\begin{array}{c}
\text{0F } ((A\rightarrow B)\rightarrow A)\rightarrow A \\
| \\
\text{00T } (A\rightarrow B)\rightarrow A \\
\text{00F } A \\
\swarrow \qquad \searrow \\
\begin{array}{cc}
\text{00T } A & \text{00F } A\rightarrow B \\
\text{X} & | \\
 & \text{000T } A \\
 & \text{000F } B
\end{array}
\end{array}
$$

The states 0, 00, and 000 are ordered by the prefix ordering, so $0 < 00 < 000$. (This choice of names for states and ordering is standard, so that for any state s, we may create any number of states greater than s but incomparable to each other, e.g. $s\hat{}0, s\hat{}1, s\hat{}2, \ldots$.) There are two branches in this tableau. The left one is closed (signified by an X) since on that branch we have both 00T A and 00F A. The right branch, however, is open, and no further development of this branch will produce a contradiction. The model which this branch describes is a three-state set $\{0, 00, 000\}$ such that the only atomic formula forced is A, forced at 000. (In fact, since states 0 and 00 force the same set of formulas, they could be collapsed into one state.) In this model, the formula $((A \rightarrow B) \rightarrow A) \rightarrow A$ is not forced at state 0, so this is a countermodel for Peirce's law.

In the tableau above, it is not obvious that further development of the right branch will not yield a contradiction. One of the disadvantages of the tableau algorithm for intuitionistic propositional logic is that the termination condition is difficult to describe. An advantage of our presentation of the algorithm is that termination is automatic, since the algorithm is extracted from an inductive proof.

2.2.2 Termination

Intuitively, the proof is by induction on the number of formulas which can be added to the system. Each step in the tableau development above either adds a new formula to an existing state on some branch or adds a new state to some branch. In the proof of theorem 2.2.1, a system corresponds to a branch of the tableau, and a node corresponds to a state in the Kripke model. Each step in the proof either adds a new formula to a node in the system, or adds a new node to the system, and this must decrease the inductive measure. When the tableau branches, we create two systems, each of which has a new formula added to some node.

To understand how the actual inductive measure works, we must look more carefully at how formulas are added to nodes and how nodes are added to systems. We will see that all formulas added to a node are subformulas of the formulas already in the node, so as long as no formula is added if it is already there, only finitely many such additions are possible. Similarly, a node is added only if it is not contained in another node in the system. Since new nodes also contain only subformulas of formulas in the original node, only finitely many nodes can be added.

Formally, there will be two inductions in the proof of the theorem, one based on the number of formulas which can be added to nodes, and one based on the number of nodes which can be added to the system. These are both measured as set differences between a given node and system and a maximum node and maximum system constructed from the formulas in the original system. However, neither the maximum node nor the maximum system is attainable if the theorem is originally applied to a system containing one node of the form $\langle \emptyset, \{\varphi\} \rangle$, so we shall also have extra conditions which will terminate the induction.

These extra conditions can be motivated by considering the tableau procedure as a search for a countermodel. At each stage, the system can be seen as a partial model, with information about what needs to be forced and not forced at each state to produce a countermodel described by the information at each node. The model is complete when, for each node $N = \langle M_1, M_2 \rangle$, N forces everything in M_1 and does not force anything in M_2, using the atoms in M_1 and M_2 to define atomic forcing. The order on the nodes in the model is defined by inclusion on the first set in each pair; since this represents the set of atomic formulas which are forced, this ensures that forcing is monotone.

For example, the tableau development above corresponds to the following sequence of systems. The original system is

$$\{\langle\emptyset, \{((A{\rightarrow}B){\rightarrow}A){\rightarrow}A\}\rangle\}.$$

This asserts that at some node, $((A{\rightarrow}B){\rightarrow}A){\rightarrow}A$ is not forced. For that to be true, there must be some node greater than or equal to this one in the model such that $(A{\rightarrow}B){\rightarrow}A$ is forced but A is not. So, we add such a node to the system, resulting in the system

$$\{\langle\emptyset, \{((A{\rightarrow}B){\rightarrow}A){\rightarrow}A\}\rangle, \langle\{(A{\rightarrow}B){\rightarrow}A\}, \{A\}\rangle\}.$$

Now, in order to have $(A{\rightarrow}B){\rightarrow}A$ forced at this new node, we must have either $A{\rightarrow}B$ is not forced at this node, or A is forced. So, we create two new systems to account for these possibilities. The first is

$$\{\langle\emptyset, \{((A{\rightarrow}B){\rightarrow}A){\rightarrow}A\}\rangle, \langle\{(A{\rightarrow}B){\rightarrow}A, A\}, \{A\}\rangle\}.$$

This system corresponds to the left branch, and is closed since A cannot be both forced and not forced. The other system is

$$\{\langle\emptyset, \{((A{\rightarrow}B){\rightarrow}A){\rightarrow}A\}\rangle, \langle\{(A{\rightarrow}B){\rightarrow}A\}, \{A, A{\rightarrow}B\}\rangle\}.$$

Again, for it to be false that $A{\rightarrow}B$ is forced in the second node, there must be a node greater than or equal to this one such that A is forced but B is not. So, we add such a node to the system, ensuring monotonicity by including the formulas forced in $\langle\{(A{\to}B){\to}A\},\{A,A{\to}B\}\rangle$ (namely $(A{\to}B){\to}A$). This produces the system

$$\{\langle\emptyset,\{((A{\to}B){\to}A){\to}A\}\rangle,\langle\{(A{\to}B){\to}A\},\{A,A{\to}B\}\rangle\}\langle\{(A{\to}B){\to}A,A\},\{B\}\rangle\}.$$

Now nothing need be done in this system, since we have, for all nodes, the formulas in the first set are forced and those in the second set are not forced.

The extra conditions which terminate the induction simply describe the situation in which no rules can be applied because their consequences are already present. Specifically, there are two conditions. One condition describes when no rules except the rule for $F\alpha{\to}\beta$ can be applied to a particular node because all consequences of the rules are present in the node, and the other condition describes when no rule for any $F\alpha{\to}\beta$ in any node can be applied because the consequences of the application of the rule are already present in the system.

2.3 Proof of the theorem

In this section we give the proof of theorem 2.2.1 in some detail. The proof is by induction, as described in the previous section, with additional conditions which describe when no further rules can be applied. We give the inductive cases first, followed by the base case. The inductive cases correspond exactly to the application of tableau rules, and the arguments for all of them are nearly identical.

We first formalize the induction measures used in the proof. As outlined above, there are two inductive measures. One measure is an upper bound for the number of formulas which can be added to a node as rules are applied. This is calculated as the size of the set difference between the node and a maximum node which contains all subformulas of the original formulas in the system. Adding a subformula of a formula in a node to that node will decrease the measure as long as the formula was not already present in the node. When no more rules apply to a node, no further formulas can be added. Similarly, the other inductive measure is a set difference between the current system and a system which contains all possible nodes made up from subformulas of formulas in the original system. This gives an upper bound on the number of nodes which can be added to the system.

More formally, for a system S define $SF(S)$ to be the set of all subformulas of formulas appearing in S. The "maximum node" based on this system is then $N_{\max}(S) = SF(S) \times SF(S)$. Define $S_{\max}(S)$ to be the set of all nodes which are subnodes of $N_{\max}(S)$. This represents the "maximum system" based on the formulas in S.

We can now define the two inductive measures used in the proof. We wish the first induction to be on number of nodes which can be added to the system. At first, it would seem that an upper bound on this number is the size of set difference between the current system S and $S_{\max}(S)$. Let

$$i_0(S) = |S_{\max}(S) - S|$$

be the first approximation to the induction measure. However, there is a difficulty with this measure. This arises from the fact that we have made the theorem more general in order to exploit the inductive structure. Because of this, we must ensure the theorem applies to all systems, not just those arising from tableau developments. In particular, we may have a system in which the application of a rule to one node results in a node which is now identical to some other node in the system. This will never be the case for nodes added to the system by the tableau algorithm; however, it must still be dealt with.

One way to do this is to take the difference between the maximum system and a system $\bar{S}$, called the *closure* of S. We define $\bar{S} = \{\bar{N} | N \in S\}$ where $\bar{N}$ is the closure of the node N under the normal tableau rules. Thus, if two nodes in S become identical after tableau rules are applied to them, they will be identical in $\bar{S}$. In [ACU], another solution is chosen which involves restricting the statement of the theorem to systems in which nodes cannot become identical.

Formally, if $N = \langle S_T, S_F \rangle$, then $\bar{N}$ is a pair $\langle \bar{S}_T, \bar{S}_F \rangle$ such that $S_T \subseteq \bar{S}_T, S_F \subseteq \bar{S}_F$ and

$$\begin{aligned}
\alpha \wedge \beta \in \bar{S}_T &\Rightarrow \alpha \in \bar{S}_T \text{ and } \beta \in \bar{S}_T \\
\alpha \wedge \beta \in \bar{S}_F &\Rightarrow \alpha \in \bar{S}_F \text{ or } \beta \in \bar{S}_F \\
\alpha \vee \beta \in \bar{S}_T &\Rightarrow T\alpha \in \bar{S}_T \text{ or } \beta \in \bar{S}_T \\
\alpha \vee \beta \in \bar{S}_F &\Rightarrow \alpha \in \bar{S}_F \text{ and } \beta \in \bar{S}_F \\
\alpha \rightarrow \beta \in \bar{S}_T &\Rightarrow \alpha \in \bar{S}_F \text{ or } \beta \in \bar{S}_T
\end{aligned}$$

Calculating $\bar{N}$ is aesthetically unpleasant, as it involves a duplication of the work done by the tableau algorithm. However, we will show later that it is not necessary for this bound to be calculated explicitly.

Now let

$$i_1(S) = |S_{\max}(S) - \bar{S}|$$

be the first induction measure. Note that when a node is added to a system S to create a system S', we shall have to verify that $i_1(S') < i_1(S)$, which involves both $S_{\max}(S)$ and $S_{\max}(S')$. However, it is easy to see that $S_{\max}(S) = S_{\max}(S')$ since any node added will contain only subformulas of formulas in the system S.

The second induction is on the number of formulas which can be added to nodes in a given system. An upper bound on this is the sum of sizes of set differences between N and the maximum node $N_{\max}(S) = SF(S) \times SF(S)$ for all nodes N in S. So, let

$$i_2(S) = \sum_{N \in S} (|SF(S) - \pi_1(N)| + |SF(S) - \pi_2(N)|)$$

be the second induction measure. Again, although we may add a formula to a node in system S to create system S', the new formula is a subformula of some formula already in S. This means that $SF(S) = SF(S')$, so we can show that $i_2(S') < i_2(S)$.

So, the final induction measure is defined as a pair

$$i(S) = \langle i_1(S), i_2(S) \rangle$$

for each system S, and these pairs are ordered lexicographically. In fact, since we can compute an upper bound for the values of i_2, we could make this a single induction on natural numbers by an appropriate pairing function. It is useful, however, to consider the inductions separately.

Although the inductive measure is reduced at each stage, in practice it may never reach zero. So, each of the inductions will terminate either when the inductive measure is zero, or when no rules apply. To ensure that the proof structure agrees with the structure of nested inductions defined by i_1 and i_2, we apply rules which reduce i_2 until no more such rules apply, then apply the rule which reduces i_1.

We first treat the cases which reduce i_2.

$\boldsymbol{T\alpha \wedge \beta}$ For any pair $\langle S_T, S_F\rangle \in S$, if $\alpha \wedge \beta \in S_T$ and ($\alpha \notin S_T$ or $\beta \notin S_T$), apply the inductive hypothesis to S', where S' is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T \cup \{\alpha, \beta\}, S_F\rangle$.

Suppose that in the system S' it is the case that $\alpha \wedge \beta \wedge (\bigwedge S_T) \to \bigvee S_F$, i.e. we have an inhabitant $f' = \lambda x_1^{\alpha} x_2^{\beta} \ldots x_k^{\alpha\wedge\beta} \ldots$. of the corresponding function type. Then we can produce an inhabitant f of $\bigwedge S_T \to \bigvee S_F$ by the following rule. Given an element a of type $\bigwedge S_T$, including an element a_k of type $\alpha \wedge \beta$, set f equal to $f'(\pi_1(a_k), \pi_2(a_k), a)$. Then, since $\alpha \wedge \beta \in S_T$, $f \in \bigwedge S_T \to \bigvee S_F$. (If some other node in S' is the one corresponding to the inhabited type, since it is also in S the theorem still holds.)

If we have a countermodel $K = (T, R, af)$ for S', and we have $f : S' \to T$ such that $f(\langle S_T \cup \{\alpha, \beta\}, S_F\rangle) = s$ for some $s \in T$ (so that $\forall \gamma \in S_T \cup \{\alpha, \beta\}.\mathit{forces}(K, s, \gamma)$), then we trivially have $\forall \gamma \in S_T.\mathit{forces}(K, s, \gamma)$. Letting $f(\langle S_T, S_F\rangle) = s$, we then have that (T, R, af) is a Kripke countermodel for S.

$\boldsymbol{F\alpha \wedge \beta}$ For any pair $\langle S_T, S_F\rangle \in S$, if $\alpha \wedge \beta \in S_F$ and ($\alpha \notin S_F$ and $\beta \notin S_F$), apply the inductive hypothesis to S_1 and S_2, where S_1 is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T, S_F \cup \{\alpha\}\rangle$ and S_2 is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T, S_F \cup \{\beta\}\rangle$.

Suppose for S_1 and S_2 it is the case that $\bigwedge S_T \to \bigvee S_F \vee \alpha$ and $\bigwedge S_T \to \bigvee S_F \vee \beta$, so we have functions f_1 and f_2 from the product type $\bigwedge S_T$ to the disjoint union types $\bigvee S_F \vee \alpha$ and $\bigvee S_F \vee \beta$. (While $\bigvee S_F$ is really a disjoint union type and should have its elements tagged to identify which disjunct they inhabit, the details of the tagging will be omitted to avoid the extra notation.) To produce a function f to $\bigvee S_F$ given an element a of type $\bigwedge S_T$, decide if $f_1(a) \in \bigvee S_F$. If so, $f(a) = f_1(a)$. Otherwise, if $f_2(a) \in \bigvee S_F$, $f(a) = f_2(a)$. If neither case holds, we must have $f_1(a) \in \alpha$ and $f_2(a) \in \beta$. Then set $f(a) = \langle f_1(a), f_2(a)\rangle$, an element of $\alpha \wedge \beta$. Since $\alpha \wedge \beta \in S_F$, f has type $\bigwedge S_T \to \bigvee S_F$.

Otherwise, we have a countermodel $K = (T, R, af)$ for S_1 or S_2. Assume without loss of generality that we have a countermodel for S_1. If we have $f : S_1 \to T$ such that $f(\langle S_T, S_F \cup \{\alpha\}\rangle) = s$ for some $s \in T$ (so that $\forall \gamma \in S_F \cup \{\alpha\}.\mathit{notforces}(K, s, \gamma)$), then we trivially have $\forall \gamma \in S_F.\mathit{notforces}(K, s, \gamma)$. Letting $f(\langle S_T, S_F\rangle) = s$, we then have that (T, R, af) is a Kripke countermodel for S.

$\boldsymbol{T\alpha \vee \beta}$ For any pair $\langle S_T, S_F\rangle \in S$, if $\alpha \vee \beta \in S_T$ and ($\alpha \notin S_T$ and $\beta \notin S_T$), apply the inductive hypothesis to S_1 and S_2, where S_1 is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T \cup \{\alpha\}, S_F\rangle$ and S_2 is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T \cup \{\beta\}, S_F\rangle$.

Suppose we have inhabitants f_1 of $\alpha \wedge (\bigwedge S_T) \to \bigvee S_F$ and f_2 of $\beta \wedge (\bigwedge S_T) \to \bigvee S_F$. Then we can produce an element of $\bigwedge S_T \to \bigvee S_F$. Since $\alpha \vee \beta$ corresponds to a disjoint union type, its inhabitants are tagged so that it is possible to decide whether a given element of $\alpha \vee \beta$ is in α or in β. Given a of type $\bigwedge S_T$, decide whether $\pi_i(a)$ (where i is the index of the element of $\bigwedge S_T$ of type $\alpha \vee \beta$) is in α or β. If $\pi_i(a) \in \alpha$, let $f(a) = f_1(\pi_i(a), a)$; if $\pi_i(a) \in \beta$, let $f(a) = f_2(\pi_i(a), a)$. In algorithmic terms, f could be written "if $\pi_i(a) \in \alpha$ then $f_1(\pi_i(a), a)$ else $f_2(\pi_i(a), a)$".

(In the term language associated with Nuprl's type theory, this would be written $decide(\pi_i(a); x.f_1(x,a), x.f_2(x,a))$.) Then f has type $\bigwedge S_T \to \bigvee S_F$.

Otherwise, we have a countermodel $K = (T, R, af)$ for S_1 or S_2. Assume that we have a countermodel for S_1. If we have $f : S_1 \to T$ such that $f(\langle S_T \cup \{\alpha\}, S_F\rangle) = s$ for some $s \in T$ (so that $\forall \gamma \in S_T \cup \{\alpha\}.forces(K, s, \gamma)$), then we trivially have $\forall \gamma \in S_T.forces(K, s, \gamma)$. Letting $f(\langle S_T, S_F\rangle) = s$, we then have that (T, R, af) is a Kripke countermodel for S.

$\boldsymbol{F\alpha \vee \beta}$ For any pair $\langle S_T, S_F\rangle \in S$, if $\alpha \vee \beta \in S_F$ and ($\alpha \notin S_F$ or $\beta \notin S_F$), apply the inductive hypothesis to S', where S' is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T, S_F \cup \{\alpha, \beta\}\rangle$.

Suppose we have an inhabitant f' of $\bigwedge S_T \to \bigvee S_F \vee \alpha \vee \beta$. Then we can produce an inhabitant f of $\bigwedge S_T \to \bigvee S_F$. Given a of type $\bigwedge S_T$, decide if $f'(a)$ is in $\bigvee S_F$. If so, let $f(a) = f'(a)$. Otherwise, $f'(a) \in \alpha$ or $f'(a) \in \beta$. If $f'(a) \in \alpha$, let $f(a)$ be $f'(a)$, tagged so that it is an element of the disjoint union $\alpha \vee \beta$. Similarly, if $f'(a) \in \beta$, let $f(a)$ be $f'(a)$, tagged as an element of $\alpha \vee \beta$. Then, since $\alpha \vee \beta \in S_F$, f has type $\bigwedge S_T \to \bigvee S_F$.

Otherwise, we have a countermodel $K = (T, R, af)$ for S'. If we have $f : S' \to T$ such that $f(\langle S_T, S_F \cup \{\alpha, \beta\}\rangle) = s$ for some $s \in T$ (so that $\forall \gamma \in S_F \cup \{\alpha, \beta\}.notforces(K, s, \gamma)$), then we trivially have $\forall \gamma \in S_F.notforces(K, s, \gamma)$. Letting $f(\langle S_T, S_F\rangle) = s$, we then have that (T, R, af) is a Kripke countermodel for S.

$\boldsymbol{T\alpha \to \beta}$ For any pair $\langle S_T, S_F\rangle \in S$, if $\alpha \to \beta \in S_T$ and ($\alpha \notin S_F$ or $\beta \notin S_T$), apply the inductive hypothesis to S_1 and S_2, where S_1 is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T, S_F \cup \{\alpha\}\rangle$ and S_2 is S with $\langle S_T, S_F\rangle$ replaced by $\langle S_T \cup \{\beta\}, S_F\rangle$.

Suppose that for S_1 and S_2, it is the case that $\bigwedge S_T \rightarrow \bigvee S_F \vee \alpha$ and $\bigwedge S_T \wedge \beta \rightarrow \bigvee S_F$, so that we have a function f_1 from $\bigwedge S_T$ to $\bigvee S_F \vee \alpha$ and a function f_2 from $\beta \wedge \bigwedge S_T$ to $\bigvee S_F$. To produce a function f from $\bigwedge S_T$ to $\bigvee S_F$, given an element a of type $\bigwedge S_T$, decide if $f_1(a) \in \bigvee S_F$. If so, $f(a) = f_1(a)$. Otherwise, $f_1(a) \in \alpha$. Now since $\alpha \rightarrow \beta \in S_T$, let a_i be the component of $\bigwedge S_T$ of type $\alpha \rightarrow \beta$. Then we can set $f(a)$ equal to $f_2(a_i(f_1(a)), a)$ (since $a_i(f_1(a))$ has type β). Then f is of type $\bigwedge S_T \rightarrow \bigvee S_F$.

Otherwise, we have a countermodel $K = (T, R, af)$ for S_1 or S_2. If K is a countermodel for S_1, and $f : S_1 \rightarrow T$ such that $f(\langle S_T, S_F \cup \{\alpha\}\rangle) = s$ for some $s \in T$ (so that $\forall\gamma \in S_F \cup \{\alpha\}.\mathit{notforces}(K, s, \gamma)$), then we trivially have $\forall\gamma \in S_F.\mathit{notforces}(K, s, \gamma)$. Letting $f(\langle S_T, S_F\rangle) = s$, we then have that (T, R, af) is a Kripke countermodel for S.

If we have a countermodel for S_2, and $f : S_2 \rightarrow T$ such that $f(\langle S_T \cup \{\beta\}, S_F\rangle) = s$ for some $s \in T$ (so that $\forall\gamma \in S_T \cup \{\beta\}.\mathit{forces}(K, s, \gamma)$), then we trivially have $\forall\gamma \in S_T.\mathit{forces}(K, s, \gamma)$. Letting $f(\langle S_T, S_F\rangle) = s$, we then have that (T, R, af) is a Kripke countermodel for S.

Finally, we treat the case which reduces the outer induction measure, i_1.

$\boldsymbol{F\alpha\rightarrow\beta}$ For any pair $\langle S_T, S_F\rangle \in S$, if $\alpha \rightarrow \beta \in S_F$ and $\forall\langle S'_T, S'_F\rangle \in S.((S_T \cup \{\alpha\} \not\subseteq S'_T) \vee (\beta \notin S'_F))$, and none of the previous cases hold, apply the inductive hypothesis to S', where S' is $S \cup \{\langle S_T \cup \{\alpha\}, \{\beta\}\rangle\}$.

If, for S', we have an inhabitant f' of $(\bigwedge S_T \wedge \alpha) \rightarrow \beta$, then the function $\lambda y^{\wedge S_T}.\lambda x^{\alpha}.f'(y, x)$ is in $\bigwedge S_T \rightarrow (\alpha \rightarrow \beta)$, so the appropriate tagged version of this function is in $\bigwedge S_T \rightarrow \bigvee S_F$.

If we have a countermodel $K = (T, R, af)$ for S', then since $S \subseteq S'$, $\langle S_T, S_F \rangle \in S$, so $f(\langle S_T, S_F \rangle) = s$ for some $s \in T$. This means we already know that $\forall \gamma \in S_T.forces(K, s, \gamma) \wedge \forall \gamma \in S_F.notforces(K, s, \gamma)$. So, we need not change f for S, except to remove $\langle S_T \cup \{\alpha\}, \{\beta\} \rangle$ from its domain. We then have that (T, R, af) is a Kripke countermodel for S.

Base case If none of the above cases hold, we have reached the base case.

Note that because none of the above cases hold, we know that, for every $\langle S_T, S_F \rangle \in S$,

$$
\begin{aligned}
\alpha \wedge \beta \in S_T &\rightarrow \alpha \in S_T \text{ and } \beta \in S_T \\
\alpha \vee \beta \in S_T &\rightarrow \alpha \in S_T \text{ or } \beta \in S_T \\
\alpha \rightarrow \beta \in S_T &\rightarrow \alpha \in S_F \text{ or } \beta \in S_T \\
\alpha \wedge \beta \in S_F &\rightarrow \alpha \in S_F \text{ or } \beta \in S_F \\
\alpha \vee \beta \in S_F &\rightarrow \alpha \in S_F \text{ and } \beta \in S_F \\
\alpha \rightarrow \beta \in S_F &\rightarrow \exists \langle S'_T, S'_F \rangle . S_T \subseteq S'_T \text{ and } \alpha \in S'_T \text{ and } \beta \in S'_F
\end{aligned}
$$

If there is an $\langle S_T, S_F \rangle \in S$ such that $\exists \alpha \in S_T \cap S_F$, then $\bigwedge S_T \rightarrow \bigvee S_F$ is inhabited by $\lambda x^{\wedge S_T}.(j, \pi_i(x))$, where i is the index of α in $\bigwedge S_T$ and j is the tag for α in the disjoint union $\bigvee S_F$.

Otherwise, $\forall \langle S_T, S_F \rangle \in S$, $S_T \cap S_F = \emptyset$. Let $T = \{N : Node | N \in S\}$, and define R, a relation on $T \times T$, by

$$R(\langle S_T, S_F \rangle, \langle S'_T, S'_F \rangle) \Leftrightarrow S_T \subseteq S'_T$$

It is easy to see that R is transitive and reflexive. Define af, a relation on $T \times Atom$ by

$$af(\langle S_T, S_F \rangle, \alpha) \Leftrightarrow \alpha \in S_T.$$

Since af is monotone with respect to R, $(T, R, af) : Kripke\_model$. Note that since $S_T \cap S_F = \emptyset, \alpha \in S_F \rightarrow \neg af(s, \alpha)$.

Define $f : S \rightarrow T$ to be the identity map from S to T. To complete the theorem, we must show that

$$\begin{aligned}
&\forall \langle S_T, S_F \rangle \in S. \\
&\quad f(\langle S_T, S_F \rangle) = s \rightarrow \\
&\qquad (\forall \alpha \in S_T.forces(K, s, \alpha) \wedge \forall \alpha \in S_F.notforces(K, s, \alpha)).
\end{aligned}$$

Since *forces* and *notforces* are defined by mutual induction on the complexity of the formula, the proofs of $(\forall \alpha \in S_T.forces(K, s, \alpha) \wedge \forall \alpha \in S_F.notforces(K, s, \alpha))$ must proceed similarly. For simplicity, however, they will be presented sequentially.

We first prove $\forall \alpha \in S_T.forces(K, s, \alpha)$, by induction on the complexity of α.

1. α is an atom. Then $forces(K, s, \alpha) = af(s, \alpha)$, true when $\alpha \in S_T$ by definition of af.

2. $\alpha = \beta \wedge \gamma$. We know $\beta \in S_T$ and $\gamma \in S_T$, so by induction hypothesis $forces(K, s, \beta)$ and $forces(K, s, \gamma)$. Therefore, $forces(K, s, \beta \wedge \gamma)$.

3. $\alpha = \beta \vee \gamma$. We know $\beta \in S_T$ or $\gamma \in S_T$, so by induction hypothesis $forces(K, s, \beta)$ or $forces(K, s, \gamma)$. Therefore, $forces(K, s, \beta \vee \gamma)$.

4. $\alpha = \beta \rightarrow \gamma$. We need that $\forall s'.sRs' \rightarrow notforces(K, s', \beta) \vee forces(K, s', \gamma)$.

Now, for all $\langle S'_T, S'_F\rangle$ such that $f(\langle S'_T, S'_F\rangle) = s'$, we know $sRs' \Leftrightarrow S_T \subseteq S'_T$, so $\beta \rightarrow \gamma \in S'_T$. We also know, then, that $\gamma \in S'_T$ or $\beta \in S'_F$, so *forces*(K, s', γ) or *notforces*(K, s', β). Therefore, *forces*$(K, s', \beta \rightarrow \gamma)$.

The proof for $\forall \alpha \in S_F.$*notforces*(K, s, α) is similar.

1. α is an atom. Since $S_T \cap S_F = \emptyset$, $\alpha \notin S_T$, so $\neg af(s, \alpha)$, i.e. *notforces*(K, s, α).

2. $\alpha = \beta \wedge \gamma$. We know $\beta \in S_F$ or $\gamma \in S_F$, so by induction hypothesis *notforces*(K, s, β) or *notforces*(K, s, γ). Therefore, *notforces*$(K, s, \beta \wedge \gamma)$.

3. $\alpha = \beta \vee \gamma$. We know $\beta \in S_F$ and $\gamma \in S_F$, so by induction hypothesis *notforces*(K, s, β) and *notforces*(K, s, γ). Therefore, *notforces*$(K, s, \beta \vee \gamma)$.

4. $\alpha = \beta \rightarrow \gamma$.

   Since $\langle S_T, S_F\rangle \in T$ and no further rules apply to the system, we must have $\exists \langle S'_T, S'_F\rangle \in T.S_T \cup \{\beta\} \subseteq S'_T \wedge \gamma \in S'_F$. Let $s' = f(\langle S'_T, S'_F\rangle)$. By induction hypothesis, *forces*(K, s', β) and *notforces*(K, s', γ). Also, since $S_T \subseteq S_T \cup \{\beta\} \subseteq S'_T$, sRs'. Therefore, $\exists s'.sRs' \wedge$ *forces*$(K, s', \beta) \wedge$ *notforces*(K, s', γ).

$\square$

Analysis of the proof

One interesting aspect of the proof of theorem 2.2.1 is that it is very easy to modify it to prove a similar theorem for classical propositional logic. To do this, we need only modify the rule for $\alpha \rightarrow \beta \in S_F$ so that it does not create a new node but instead adds α to S_T and β to S_F. Thus, no nodes are ever added to the system.

With this modification, the tableau rules the correspond precisely to Gentzen's sequent calculus for classical logic, with the rules applied from the goal backwards.

If this classical theorem is applied to a system $\{\langle\emptyset, \{\varphi\}\rangle\}$, the result is either a classical tableau proof or a Kripke countermodel consisting of precisely one state. Thus we know that φ is not forced at that state. We can use the atomic forcing relation to assign truth values to the atoms in the formula (true if forced, false if not) to produce a classical truth value assignment for which the formula is not true.

If this theorem were implemented in Nuprl, a number of points would have to be made more explicit. Further explanation of these details is found in [ACU].

First, the types *Wff* and *Atom* would have to be instantiated with actual Nuprl types. This could be done several ways, but perhaps the most flexible is to define an abstract data type of formulas, and prove the theorem with respect to this data type. Then the theorem could be applied to any instantiation of the data types, including, for instance, logical formulas built from atomic formulas which represent equations in arithmetic.

Second, in the interpretation of the theorem as a completeness proof, the type of proofs of formulas was considered to be identical to the type of terms in the language. If the type of formulas is actually described as an abstract data type, some translation is necessary to ensure that the term is in an appropriate type. Alternatively, the theorem can be proved with respect to an abstract data type of proofs including proof constructors. Instead of being expressed explicitly as a λ-term, the proof of the formula can be represented in the abstract proof type.

Finally, the induction measures would have to be defined in more detail. This involves the definition of a number of functions on nodes and systems, described in [ACU]. Note, however, that the value of the induction measure was never explicitly used in the proof. Thus, it should not be necessary to calculate it in the extracted algorithm; we merely need to know that it exists and is reduced at each stage. Furthermore, since the algorithm is naturally recursive, it would be nice if the algorithm extracted from the proof were recursive as well.

To achieve this in Nuprl, we can use a theorem ("efficient induction") for the proof by induction instead of the built in induction. Specifically, we can prove something like an induction principle for well-founded orders $<$ on systems. In this case, given the definition of the type *System*, it is sufficient to prove a theorem of the form

$$
\begin{aligned}
&\forall P : System \rightarrow U_i \\
&\quad \forall f : System \rightarrow N \\
&\qquad (\forall S : System.(\forall S' : \{S' : System | f(S') < f(S)\}.P(S')) \Rightarrow P(S)) \\
&\qquad\quad \Rightarrow (\forall S : System.S.P(S))
\end{aligned}
$$

In this theorem, P is the predicate we wish to show holds for all systems, and f is an inductive measure on systems. The use of the set type $\{S' : System.S | f(S') < f(S)\}$ guarantees that the computational content of the proof that the inductive measure of S' is less than the measure of S is not used in the extraction. This means that the numerical induction measures need not be explicitly calculated at run time as long as they were shown to have been reduced in the proof of the theorem which uses this induction theorem.

The induction principle above may be proved in Nuprl by supplying an explicit witness for its truth. In this case, the witness is essentially

$$\lambda P.\lambda f.\lambda g.(\mathbf{Y}\lambda h.\lambda x.(gxh))$$

where $\mathbf{Y} = \lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x)))$ is the usual Y combinator. This term is shown to be an inhabitant of the appropriate type using Nuprl's computation rules, which allow the $\mathbf{Y}$ combinator to be "unrolled" in order to use the inductive hypothesis.

Analysis of the algorithm

When the algorithm extracted from this proof is applied to a single formula, its running time is exponential in the length of the formula. Since the problem is PSPACE-complete, this is not surprising; however, further analysis is warranted in light of the work of Dyckhoff [Dyc92] (and also Hudelmaier [Hud92]).

The main factors which contribute to the time complexity are the number of rules applied and the number of branches created. Since any rule application may split a branch, if we have a bound of m on the number of rules applied on any branch, then the total number of rules applied is bounded by 2^m. However, we must also account for the complexity involved in deciding which rule is to be applied and in applying it.

We calculate the number of rules which may be applied on any branch as follows. First, consider the number of rules which may be applied at a single node (other than the rule for $F\alpha\!\rightarrow\!\beta$). Each rule breaks down a formula based on its principal connective and adds at least one new formula to the node. We may consider that the rules are being applied to connectives in the original formula φ, since all formulas added are subformulas of φ, and no rule is applied the same connective more than once at this node. In this light, it is clear that the length of φ bounds the number of rules which can be applied at a single node.

Similarly, we can bound the number of nodes which are added to a single system. A node is added when the rule $F\alpha \rightarrow \beta$ is applied, i.e. $\alpha \rightarrow \beta$ is in S_F in that node. All such $\alpha \rightarrow \beta$ are still subformulas of the original formula, and once the rule has been applied for a particular $\alpha \rightarrow \beta$, it is not applied again. So, the number of applications of this rule, and hence the number of nodes which can be added to a system, is bounded by the number of implications in the original formula φ. For simplicity, we again use the length of φ as a bound.

We now have the number of rules which may be applied along a branch bounded by n^2, where n is the length of φ. Accounting for branching, then, we have the number of rules applied in the total tableau development bounded by 2^{n^2}. Howver, we have not yet accounted for the complexity of determining which rule to apply. This task requires that we check, for each rule, whether or not its consequences are already present in the node or system. If they are, the rule is not applicable. As presented here, this task is very inefficient. Orderings of the formulas within the sets S_T and S_F could help, but there is still an essential difficulty in the search through all the nodes in the system to determine if the $F\alpha \rightarrow \beta$ rule is applicable.

This problem is referred to by Dyckhoff as the problem of "loop detection". He traces it to the need for monotonicity; in our context, this is reflected in the fact that S_T must be incorporated into every new node created by application of the $F\alpha \rightarrow \beta$ rule to a node $\langle S_T, S_F \rangle$. In particular, Dyckhoff shows that the difficulty lies in the fact that formulas $\alpha \rightarrow \beta$ in S_T must be reduced at each node in which they appear. To avoid this, in [Dyc92] he introduces a calculus which replaces this rule with rules accounting for the different possible forms of α. In doing so, he removes the need for explicit tests to detect repetitions in the pattern of rule application, and hence produces a more efficient algorithm. The running time is still exponential in a polynomial of n, but the multiplicative factor is reduced, and the polynomial may be smaller. Dyckhoff and Pinto have developed a related system for constructing countermodels [PD].

2.4 Applications

In this section, we consider an application of this result to the problem of adding verified decision procedures to a theorem proving system. This example is explored in depth in [ACU], which is a development of an idea in [BC93].

In using a proof development system (such as Nuprl) which is based on a large underlying logic, patterns develop in the application of proof rules. If the system is to be useful, it is highly desirable that there be a way to express these patterns as new proof methods built from sequences of fundamental rules. These methods are known as *tactics*, and were introduced in the Edinburgh LCF system [GMW79] and adopted by many other theorem provers [C^{+}86,CH88,Gor85,ORS92]. Since tactics are just instructions for applying a sequence of primitive rules, there is no loss of soundness in the system.

In addition to tactics, we may wish to add more complex decision procedures to the system. Decision procedures may achieve their results by some method other than application of primitive rules; however, we would still want to know that the result is valid in the underlying logic. In a sufficiently rich constructive type theory with an associated computation system, it is even possible to write the decision procedure in the logic itself, avoiding the necessity of reasoning about a separate programming language. The computational content of the proof in this chapter is an example of this.

Furthermore, a sufficiently rich type theory can describe models of its own terms and proofs. In Nuprl, for example, we can prove the constructive completeness proof of this chapter with respect to abstract types for formulas and proofs, and then instantiate the type with Nuprl's propositional formulas and an internal representation of Nuprl's proofs. However, there remains the problem of translating the proof in the internal model to an actual Nuprl proof. This requires a *reflection rule*, developed by Aitken and explained in his thesis [Ait94].

The reflection rule says that if we wish to prove

$$H \vdash G$$

where H is a list of hypotheses and G is the goal, then it is sufficient to prove

$$\exists P \in \ \mathit{Proof}. P \text{ proves } \ulcorner H \vdash G \urcorner$$

where *Proof* is the type of Nuprl proofs expressed within Nuprl and $\ulcorner H \vdash G \urcorner$ is a reflected representation of the original sequent.

As an example of the use of the reflection rule, in [ACU] we describe how to formalize the proof given in this chapter in order to use its computational content as verified decision procedure to Nuprl. To do this, we recast the statement in terms of abstract data types for formulas, proofs, and structures used in the proof. This step is not strictly necessary, but it makes reasoning about these types simpler and more general. Then, after proving the theorem with respect to these abstract types, we can instantiate these types with Nuprl's reflected term and proof types, to obtain a theorem along the lines of

> Given a propositional formula in the reflected term type, either there is a proof in the reflected proof type or there is a countermodel.

Finally, the reflection rule can be used to bring the reflected proof into the system itself. Since the computational content of the proof is the tableau algorithm, we will have added a particular decision procedure for intuitionistic propositional calculus to the system. Note that the reflection rule can be applied via a lemma, so that the user of the decision procedure need not know about the use of reflection.

Chapter 3

Constructive Approaches to Completeness for Intuitionistic Predicate Calculus

In this chapter, we shall explore various generalizations of the result of the preceding chapter to the intuitionistic predicate calculus. In the propositional case, the constructive content of the completeness proof was a decision procedure for provability in the calculus. We cannot expect such strong results in the predicate case, since provability in intuitionistic predicate logic is undecidable (by reduction from the undecidability of classical predicate logic – see [BJ88]). However, the theorem can be modified so that the computational content of the proof is a version of the tableau procedure for predicate logic, and different modifications to the theorem result in versions of the procedure with different properties.

We will present a completely constructive proof which has as its computational content a bounded tableau search procedure. This procedure may return a proof or a finite countermodel, or it may return a finite approximation of a potential (and possibly infinite) countermodel. In the last case, we will know that no proof was found within the given bound; however, a more extensive search may still produce a proof. Before we can state the theorem, though, we will need to introduce type theoretic machinery for defining infinite models.

Finally, we will present some variations on the theorem using some classical logic. These are designed in such a way that the proof still has the same computational structure, but that classical logic may be used for reasoning about properties of the objects created. We will then consider the possibility of a theorem which has an unbounded tableau search as its computational content.

3.1 Previous work

Much earlier work exists on the subject of a constructive model theory for intuitionistic predicate logic. This earlier work was inspired by the needs of intuitionists to find an acceptable semantics for their logic. Many of these ideas can be adapted in a computational approach to logic.

In general, there are two issues involved in describing a completeness proof constructively. First, there is the necessity of describing the models themselves constructively. Second, the statement of the theorem must be chosen so that the proof is constructive. In the case of the Kripke models for propositional logic used in the previous chapter, the first issue was not addressed because the completeness theorem showed that finite models were sufficient. This is not the case for predicate logic, however.

Model theories for intuitionistic predicate logic were developed by Beth [Bet56] and Kripke [Kri65]. Both theories employ models which are generalizations of the Kripke models described in the previous chapter, adding rules to the forcing relation for quantified formulas. These provide a sound and complete classical semantics, but cannot be shown to be complete intuitionistically. An essential difficulty is that, computationally, models can only be described by a nonterminating processes. Without further restrictions, this makes reasoning intuitionistically about validity in the models very difficult. In fact, Beth believed his models were constructively complete; unfortunately his proof was shown to contain non-constructive reasoning. Specifically, Kreisel and Dyson [DK61] showed that Beth's proof appealed to a form of Markov's principle, which states that for a decidable predicate $A(x)$, $\neg\neg\exists x.A(x)\rightarrow\exists x.A(x)$. Markov's principle is not provable in intuitionistic predicate calculus.

One solution to the problem of finding an intuitionistically valid model theory was proposed by Veldman [Vel76] and a similar approach was taken by de Swart [dS76]. Both modify Beth-Kripke models to allow *false* to be forced at some states; hence, at these states every formula is forced. These states are called "sick" or "exploding" states. It is a consequence of their proofs that, for any model, the set of exploding states in the model is undecidable.

This modification in the definition of a model means that negation can be defined in terms of a positive condition on states in the model, rather than on a negative condition. To be precise, instead of defining "s forces $\neg\alpha$" to mean "no state $s' \geq s$ in the partial order forces α", it is defined to mean "if $s' \geq s$ forces α, then s' explodes". This second statement is easier to reason about intuitionistically, and avoids the need to identify *a priori* the exploding states.

However, both Veldman and de Swart need to use the fan theorem in their completeness proofs. This "theorem" is actually an axiom adopted by some intuitionists for reasoning about properties of infinite trees. Its justification depends critically on the assumption that the sequences representing branches in the trees can be generated arbitrarily; in particular, it conflicts with the idea that branches must be recursively computable. In the realm of computer science, however, recursive functions are widely accepted to represent the computable functions; this principle is known as Church's thesis. Since Church's thesis is incompatible with the fan theorem, the fan theorem is somewhat dubious in the context of computability.

A brief discussion of the fan theorem

In the remainder of this section, we examine the fan theorem and its computational implications. The treatment is from Troelstra [TvD88b, chapter 4], abridged and modified to focus on the role of the fan theorem in the completeness proofs of Veldman and de Swart.

A *fan* is a finitely branching infinite tree. A tree is defined formally as a nonempty, decidable set of finite sequences of natural numbers closed under sequence prefixes. This can be interpreted as describing the set of paths in the tree, where vertices of the tree are labelled with natural numbers. We will use the language of paths and branches, rather than sequences, since it is more natural for computer science. A tree is infinite if every branch can be extended.

In the remainder of this section only, we use the variable α (and $\alpha_1, \alpha_2, \ldots \alpha_i$) to stand for infinite branches in a fan, and $\bar{\alpha}$ to represent a finite initial segment of α. The initial segment of α of length x will be denoted $\bar{\alpha}x$.

The fan theorem says

Theorem 3.1.1 (Fan theorem) *Given a fan F and a predicate A, if, for every infinite branch α_i of F, there is an initial segment $\bar{\alpha}_i n_i$ such that $A(\bar{\alpha}_i n_i)$, then there is an n such that $\forall i, n_i < n$. In other words, there is a uniform upper bound on the lengths of the initial segments satisfying A.*

This can be restated as

$$\forall \alpha \in F. \exists x. A(\bar{\alpha}x) \rightarrow \exists z. \forall \alpha \in F. \exists y \leq z. A(\bar{\alpha}y)$$

A complete formal statement and discussion can be found in Troelstra [TvD88b, p. 217], which includes a variation in which the predicate A is required to be decidable, and a proof that in either case we may restrict attention without loss of generality to binary fans.

The fan theorem is an intuitionistic interpretation of König's lemma, which states that if there is no finite upper bound to the lengths of paths in a tree, then that tree has an infinite path. The fan theorem is the contrapositive of this classical statement, and hence is classically true.

However, both the fan theorem and König's lemma fail if α ranges over recursive sequences only. Specifically, Kleene [Kle50] defined a recursive predicate P such that $\forall \alpha. \exists x. P(\bar{\alpha}x)$ but $\forall x. \exists \alpha. (P(\bar{\alpha}x) \wedge \forall y < x. \neg P(\bar{\alpha}y)$. Hence, there is no upper bound on the size of the prefix satisfying P, but neither is there an infinite branch.

This counterexample does not meet the hypotheses of the fan theorem if α is allowed to range over arbitrary paths and not just recursive ones. The fan theorem is one of a family of possible extensions to constructive mathematics which have various philosophical justifications. The debate on these extensions centers on the nature of the sequences α involved. These are referred to as *choice sequences*, and the restrictions laid upon them range from none (*lawless* sequences) to the restriction that they be generated by a rule of some kind (*lawlike* sequences, including recursive sequences). In a standard computational context, all sequences α are known to be recursive, so there can be no reason to accept the validity of the fan theorem.

In the completeness theorem of de Swart, the fan theorem is used to extract a proof from what is essentially a tableau construction. If we know that the fact that φ is forced in all models implies eventually every branch on the tableau rooted at $F\varphi$ will close, the fan theorem gives a bound on the length of the branches, and hence on the size of the proof. The computational content of the fan theorem in this context is an oracle which supplies a bound on the necessary proof search and guarantees that a proof will be found within that limit.

Dummett's book [Dum77] presents a detailed account of completeness issues for intuitionistic predicate calculus. He uses the fan theorem but also demonstrates that it alone is insufficient to prove completeness for standard Beth/Kripke models constructively. It is for this reason that Veldman and de Swart modified the usual definition of Beth/Kripke models to allow states which force $\perp$, and recast their completeness proofs accordingly. We will reconsider their approach in the last section of this chapter.

Later variations on the completeness theorem of Veldman and de Swart (for example, [TvD88a, chapter 13], which cites Friedman's proof [Fri75,Fri77a,Fri77b] as its source) avoid explicit use of the fan theorem by using Beth models and moving the upper bound on the search into the definition of forcing in the model. Thus, the proof that a formula is true in all models contains the bound which the fan theorem would otherwise provide. However, when the completeness theorem is treated as a decision procedure, we do not have this information.

3.2 The tableau procedure for intuitionistic predicate calculus

The goal of the remainder of this chapter is to describe the tableau procedure as the computational content of a constructive proof. Since the problem is undecidable, the tableau procedure is necessarily a partial function, so we cannot expect a decision procedure. However, there are several ways in which the undecidability can be manifested while still leaving some computational value. As in the propositional case, it is helpful to review the tableau procedure in order to motivate and explain the choices for the definition of Kripke models and the statement of the theorem.

The tableau procedure for predicate logic is a relatively small extension of that for propositional logic. The one presented here is based on that presented in [Ner89]; see also [Fit83] for a more detailed treatment. Recall that the propositional tableau can be interpreted as a search for a Kripke countermodel; should this search fail, the result is a proof. The additional rules for predicate tableaux are a reflection of the rules for forcing in Kripke models for predicate calculus.

The models we consider are essentially generalizations of the models described in the last chapter. Suppose we are given a set C including constant symbols and a set P of relation symbols of specified arity. Let $Atom$ be the atomic terms of this language. A model is a partially ordered set $(S, \leq)$, and a pair of relations af and dom, such that

- dom is a function from S to the power set of C such that for all s, s' in S, $s \leq s'$ implies $dom(s) \subseteq dom(s')$. The set $dom(s)$ will be referred to as the *domain* of s.

- af is a relation between elements s of S and atomic formulas built from elements of $dom(s)$ and the relation symbols P. This is a generalization of the atomic forcing of the previous chapter to the predicate case. As before, af must be monotone with respect to the partial order on S.

The definition of forcing is extended by the following rules:

$$
\begin{array}{lll}
s \text{ forces } \forall x.\varphi(x) & \text{iff} & \forall s' \geq s. \forall b \in dom(s').s' \text{ forces } \varphi(b) \\
s \text{ forces } \exists x.\varphi(x) & \text{iff} & \exists b \in dom(s).s \text{ forces } \varphi(b)
\end{array}
$$

As a consequence, the tableau procedure described in the previous chapter is extended to describe the consequences of asserting that a quantified formula is forced or not at a state. The additional rules are as follows:

1. If $\forall x.\varphi(x)$ is true at a state s, then assert that $\varphi(a)$ is true for all $a \in dom(s)$. As long as monotonicity of forcing is ensured as the tableau develops, this rule is sufficient.

2. If $\forall x.\varphi(x)$ is marked false at a state s, then add a new state $s' \geq s$ and a new symbol $c \in dom(s')$ and assert that $\varphi(c)$ is false at state s'. (If such a

symbol already exists in $dom(s)$, this rule need not be applied.)

3. If $\exists x.\varphi(x)$ is marked true at a state s, then add a new symbol c to $dom(s)$ and assert that $\varphi(c)$ is true at s. (Again, if such a symbol already exists in $dom(s)$, this rule need not be applied.)

4. If $\exists x.\varphi(x)$ is marked false at a state s, then assert that $\varphi(a)$ is false at s for all $a \in dom(s)$.

As an example, consider the following tableau for the formula $\forall x.\varphi(x) \vee \neg\varphi(x)$.

$$\begin{array}{c}
\emptyset\ 0\ \mathrm{F}\ \forall x.\varphi(x) \vee \neg\varphi(x) \\
| \\
\{a\}\ 00\ \mathrm{F}\ \varphi(a) \vee \neg\varphi(a) \\
\{a\}\ 00\ \mathrm{F}\ \varphi(a) \\
\{a\}\ 00\ \mathrm{F}\ \neg\varphi(a) \\
| \\
\{a\}\ 000\ \mathrm{T}\ \varphi(a)
\end{array}$$

Here, the states are 0, 00, and 000, ordered by string prefix. The sets shown before the labels describe the domains for each label, so that $dom(0)$ is $\emptyset$, and $dom(00)$ and $dom(000)$ are $\{a\}$. The second state was created and the symbol a was added to its domain to satisfy the tableau development rule for $F\forall x.\varphi(x) \vee \neg\varphi(x)$. This tableau is not closed because no formula is labelled both T and F at the same state. In fact, this tableau describes a countermodel for the formula $\forall x.\varphi(x) \vee \neg\varphi(x)$. The model has three states. The domains and atomic formulas forced at each state are defined as follows:

State	Domain	Atomic formulas forced
0	$\emptyset$	$\emptyset$
00	$\{a\}$	$\emptyset$
000	$\{a\}$	$\{\varphi(a)\}$

In this case, it can be determined that no further application of the tableau rules will result in closure. In general, however, this will not be the case. For example, consider a tableau which contains the following development:

{ a } 0T $\forall x.\varphi(x)$
{ a } 0T $\forall x.\varphi(x)\rightarrow\exists y.\psi(x,y)$
{ a } 0T $\varphi(a)\rightarrow\exists y.\psi(a,y)$

{ a } 0F $\varphi(a)$
{ a } 0T $\varphi(a)$
X

{ a } 0T $\exists y.\psi(a,y)$
{ a,b } 00T $\psi(a,b)$
{ a,b } 00T $\forall x.\varphi(x)\rightarrow\exists y.\psi(x,y)$
{ a,b } 00T $\varphi(b)\rightarrow\exists y.\psi(b,y)$

{ a,b } 00F $\varphi(b)$
{ a,b } 00T $\varphi(b)$
X

{ a,b } 00T $\exists y.\psi(b,y)$
$\vdots$

The rightmost branch of this tableau could be extended forever. Each time a new state and new constant b are added to satisfy $\exists y.\varphi(a,y)$, monotonicity requires that we also instantiate $\forall x.\varphi(x)\rightarrow\exists y.\psi(x,y)$ at the new state with this new constant. This leads to the assertion of $\exists y.\varphi(b,y)$ at the new state, which requires a further new state and new constant. So, this tableau always contains a branch to which rules can be applied, and thus never produces either a proof or a finite countermodel. However, this infinite tableau does produce an infinite countermodel for the formula $\neg(\forall x \varphi(x) \wedge \forall x \varphi(x) \rightarrow \exists y . \psi(x, y))$. Note that the countermodel, though infinite, is constructed according to a simple set of rules. Hence, the model itself is constructive.

If it were the case that every formula not provable in the intuitionistic predicate calculus had a finite countermodel, then provability would be decidable. Since this is not true, we will need some way of coping with the possibility of an infinite countermodel. The method we use for constructing such models in type theory is described in the next section.

3.3 Constructive models in type theory

In order to describe the possibly infinite tableau development in type theory, we use *co-inductive types*. Just as inductive types are the least fixed point of a type equation $T = F(T)$, co-inductive types are defined to be the greatest fixed point of a type equation. In his thesis, Mendler [Men88] described how both inductive and co-inductive types could be added to Nuprl, and co-inductive types have also been treated in other contexts [Bac89,Pit].

As with the propositional case, the same mathematical "data structure" will be used to describe both the developing tableau proof and the countermodel. However, we will not explicitly construct a term inhabiting the type corresponding to the formula; instead, we simply treat a closed tableau as a proof. (It is straightforward to extract a sequent proof from a closed tableau [Fit83].) This is not an essential modification; as in the previous chapter, we could extract a term in the appropriate type and use this as semantic evidence instead of a proof.

The general approach is similar to that in Dummett [Dum77]; however, we combine his notions of "proof tree-trunk" and "refutation tree-trunk" (finite approximations to proofs and countermodels) into a single structure. We focus on how this structure can be implemented in constructive type theory. Furthermore, we wish our theorems to have computational value, so we reject the fan theorem.

3.3.1 Introduction to co-inductive types

To understand the idea behind co-inductive types, we first consider an example of an inductive type. The type of lists of natural numbers can be defined as a recursive type, i.e.

$$NatList = Nil + Nat \times NatList$$

where $+$ denotes disjoint union, and $\times$ represents pairing.

However, there is actually more than one type satisfying the above equation. The type of lists of natural numbers is one, but the type of possibly infinite natural number streams also meets the conditions. Generally, recursive types are constructed so that the result is the least fixed point of the type equation. This can be made more explicit by the use of a least fixed point operator μ. The definition above could be rewritten

$$NatList = \mu T.Nil + Nat \times T$$

where T takes the place of $NatList$ in the right hand side of the definition.

If, instead, we want the greatest fixed point of the type equation, we use the ν

operator. So,

$$NatStream = \nu T.Nil + Nat \times T$$

is the type of possibly infinite streams of natural numbers.

To define elements of ν-types, we use a ν_ind form:

$$\nu\_ind(d; z, y.b) \in \nu T.B$$

where d is an element of some type D, y is a variable of type D, and z is a variable with function type $D \rightarrow T$. For example, a stream of natural numbers is defined:

$$\nu\_ind(0; z, y.\langle y, z(y+1)\rangle)$$

The ν_ind form is evaluated lazily, according to the rule

$$\nu\_ind(d; z, y.b) \rightarrow_1 b[\lambda x.\nu\_ind(x; z, y.b)/z, d/y]$$

so that one evaluation step of

$$\nu\_ind(0; z, y.\langle y, z(y+1)\rangle)$$

gives

$$\langle 0, \lambda x.\nu\_ind(x; z, y.\langle y, z(y+1)\rangle)(1)\rangle$$

and, in another step,

$$\langle 0, \nu\_ind(1; z, y.\langle y, z(y+1)\rangle)\rangle$$

Further details on these types are are in Mendler's thesis [Men88].

In the current case, the data structure we use to describe the type of Kripke models and tableau proofs will represent the developing tableau. The general form of the definition is that of a possibly infinite labelled binary tree

$$\nu T.S + S \times T \times T$$

where S is the type of the information contained at each stage of the developing tableau. In this case, we will not want to insist that all points on a branch have two subtrees, so we use the general form

$$\nu T.S + S \times T + S \times T \times T$$

instead. The ν_ind form will be used to define the developing tableau inductively.

3.3.2 Types for Kripke models and tableau proofs

In this section, we define the type underlying tableau proofs and Kripke model developments using the ν types described in the previous section. Since we have the general form of the definition, we need only determine precisely what information is required at each stage of the development. We then describe the tableau development itself as a function from an initial set of information to the tableau with that information at its root.

The information required at each stage includes a set of formulas tagged with their truth values. So, we will define a *partial proof tree* (*PPT*) to be a (possibly infinite) tree labelled at each vertex with these sets. However, as in the propositional case, we require more information than this structure provides. In order to distinguish between states in the Kripke model and vertices in the tree, we add another piece of information to each tagged formula. This information is a label describing the state in the model with which the formula should be associated. Let *Labels* be the type of this information. These labels must eventually describe a partial order on the set of states. Since states may be added to the set during development, we will need a function *NewLabel* which, given a label ℓ, returns a new label associated with a state greater than ℓ in the partial order and incomparable to all other states not less than ℓ. We assume $\ell_0 \in \mathit{Labels}$ is given as a starting point.

Furthermore, since we may have formulas tagged with different labels at one stage in the tableau development, we will make explicit the domain associated with each formula. So, the information required at each stage in the tableau consists of a set of formulas, each associated with a label (representing a state in the possible Kripke countermodel), a truth value, and a set of constants in the domain at that state. The information associated with each formula corresponds precisely with the information shown on each row in the example tableau developments, so we will define a type *Row* to represent it. We will refer to the collection of this information present at each vertex in the partial proof tree as a *stage descriptor (SD)*.

Now we may describe the type of our partial proof trees. Let *CSet* the type of sets of propositional constants, let *Row* be the 4-tuple

$$\mathit{CSet} \times \mathit{Labels} \times \{T, F\} \times \mathit{Wff},$$

and let SD be the type of lists of Row. The type of partial proof trees is then

$$PPT = \nu T.SD + (SD \times T) + (SD \times T \times T)$$

It will be clear from context which of the three elements of the disjoint union is intended.

We may now define a function which represents the development of a tableau. This function takes a stage descriptor and returns a description of a tableau development rooted at that stage. For each node, the function chooses a rule which applies a formula in the stage descriptor, applies it, and returns the resulting (partial) tableau. In each stage descriptor, we keep information about the entire branch leading to that stage, so that the decision of which tableau rule to apply can be made with local information. Thus, the stage descriptors are analogous to the systems used in the previous chapter. The collection of all the elements of Row associated with a particular label corresponds to a node in the previous chapter.

	Chapter 2	φ provable	φ has a Kripke countermodel
{ rows with one label }	*Node*	sequent	state
stage descriptor	*System*	branch	model

In the rules given below, when the domain and label of the elements of SD are not specified, they are assumed to be the same for all tagged formulas mentioned in the rule. Tagged formulas will be given their usual notation of $T\alpha$ or $F\alpha$. $Develop$: $SD \rightarrow PPT$ is defined using the ν_ind form. The outline of the definition is

$$Develop = \lambda S_0.\nu\_ind(S_0; z, S.\text{cases depending on } S)$$

where S_0 is an element of SD S is a variable of type SD and z is a variable of type $SD \rightarrow T$. During reduction, the variable z is the placeholder for the term representing the recursive call; for clarity, we use the variable name *develop* instead of z. The outline of the definition of *Develop* is then

$$Develop = \lambda S_0.\nu\_ind(S_0; develop, S.\text{cases depending on } S)$$

and the cases are as follows:

$\boldsymbol{T\alpha \wedge \beta}$ If $(T\alpha \wedge \beta) \in S$ and $(T\alpha \notin S$ or $T\beta \notin S)$ then return

$$\langle S, develop(S \cup \{T\alpha, T\beta\})\rangle$$

$\boldsymbol{F\alpha \wedge \beta}$ If $(F\alpha \wedge \beta) \in S$ and $(F\alpha \notin S$ and $F\beta \notin S)$ then return

$$\langle S, develop(S \cup \{F\alpha\}), develop(S \cup \{F\beta\})\rangle$$

$\boldsymbol{T\alpha \vee \beta}$ If $(T\alpha \vee \beta) \in S$ and $(T\alpha \notin S$ and $T\beta \notin S)$ then return

$$\langle S, develop(S \cup \{T\alpha\}), develop(S \cup \{T\beta\})\rangle$$

$\boldsymbol{F\alpha \vee \beta}$ If $(F\alpha \vee \beta) \in S$ and $(F\alpha \notin S$ or $F\beta \notin S)$ then return

$$\langle S, develop(S \cup \{F\alpha, F\beta\})\rangle$$

$\boldsymbol{T\alpha \to \beta}$ If $(T\alpha \to \beta) \in S$ and $(F\alpha \notin S$ or $T\beta \notin S)$ then return

$$\langle S, develop(S \cup \{F\alpha\}), develop(S \cup \{T\beta\})\rangle$$

$\boldsymbol{F\exists x.\varphi(x)\beta}$ If $\langle C, \ell, F, \exists x\varphi(x)\rangle \in S$ and $\exists c \in C$, $\langle C, \ell, F, \varphi(c)\rangle \notin S$ then return

$$\langle S, develop(S \cup \{\langle C, \ell, F, \varphi(c)\rangle | c \in C\})\rangle$$

$\boldsymbol{T\forall x.\varphi(x)\beta}$ If $\langle C, \ell, T, \forall x\varphi(x)\rangle \in S$ and $\exists c \in C$, $\langle C, \ell, T, \varphi(c)\rangle \notin S$ then return

$$\langle S, develop(S \cup \{\langle C, \ell, T, \varphi(c)\rangle | c \in C\})\rangle$$

$\boldsymbol{T\exists x.\varphi(x)\beta}$ If $\langle C, \ell, T, \exists x\varphi(x)\rangle \in S$ and $\forall a \in C$, $\langle C, \ell, T, \varphi(a)\rangle \notin S$ then let c be a new constant, let $\hat{C} = C \cup \{c\}$ and let

$$\begin{aligned} S' &= \{\langle C', \ell', V, \alpha\rangle | (\ell' \neq \ell \text{ and } \langle C', \ell', V, \alpha\rangle \in S) \\ &\quad \text{or } (\ell' = \ell \text{ and } \langle C, \ell, V, \alpha\rangle \in S \text{ and } C' = \hat{C})\} \\ &\cup \{\langle \hat{C}, \ell, T, \varphi(c)\} \end{aligned}$$

and return

$$\langle S, develop(S')\rangle$$

This procedure adds the constant c to the domain of all formulas associated with the label ℓ and adds the tagged formula $T\varphi(c)$ to the set of formulas associated with ℓ.

The next two rules are the only ones which require the addition of new labels, since they are the only ones which add states to the tableau. Recall that $NewLabel(\ell)$ returns a label greater than ℓ and incomparable to any other label not less than ℓ.

$\boldsymbol{F\alpha\rightarrow\beta}$ If $\langle C, \ell, F, \alpha\rightarrow\beta\rangle \in S$ and $\forall \ell_0 \geq \ell.\langle C, \ell_0, T, \alpha\rangle \notin S$ or $\langle C, \ell_0, F, \beta\rangle \notin S)$ then let $\ell' = NewLabel(\ell)$, let

$$S' = S \cup \{\langle C, \ell', T, \gamma\rangle | \langle C, \ell, T, \gamma\rangle \in S\} \cup \{\langle C, \ell', T, \alpha\rangle, \langle C, \ell', F, \beta\rangle\}$$

and return

$$\langle S, develop(S')\rangle$$

$\boldsymbol{F\forall x.\varphi(x)\beta}$ If $\langle C, \ell, F, \forall x\varphi(x)\rangle \in S$ and $\forall a \in C$, $langle C, \ell, F, \varphi(a)\rangle \notin S$ then let c be a new constant, let $C' = C \cup \{c\}$, let $\ell' = NewLabel(\ell)$, let

$$S' = S \cup \{\langle C', \ell', T, \gamma\rangle | \langle C, \ell, T, \gamma\rangle \in S\} \cup \{\langle C', \ell', F, \varphi(c)\rangle\}$$

and return

$$\langle S, develop(S')\rangle$$

Closure if there is a formula α such that $T\alpha \in S$ and $F\alpha \in S$, or if $T\bot \in S$, then return

$$\emptyset$$

This signifies the closure of this branch of the development.

Completion of countermodel If none of the preceding rules apply, then return

$$\langle S \rangle.$$

This branch cannot be further developed, and describes a countermodel.

The result of the application of *Develop* is a partial proof tree P which must be evaluated lazily. We assume there is an operator $\mathit{unwind}(n, P)$ which unrolls the definition of P n stages along each branch.

Note that there is a function from *Labels* to *CSet* describing the set C associated with a label ℓ. There is only one such set because whenever the set C is modified, either we have created a new label or all tuples involving the label are updated with the new set C.

We stipulate that each element of S be chosen as a candidate for the application of a rule in rotation. This means that if a rule can be applied to an element of S and no application of another rule causes that rule to be no longer applicable, then eventually the rule will be applied. The reason for this restriction will be explained in the next section.

3.3.3 Correctness of the construction

We demonstrate that the partial proof trees created by evaluation of the *Develop* function defined in the previous section are a sound representation of tableau developments.

A tableau development for φ is a partial proof tree beginning with $F\varphi$ and such that along each branch, each stage descriptor is the result of the application of some tableau rule to the previous one. Given a provable formula φ, it is not the case that every possible tableau development produces a tableau proof. However, if a formula is provable, there exists a tableau proof, that is,a terminating tableau development such that all branches are closed. Although the rules given in the definition of *Develop* are standard tableau development rules, some justification must be made for the correctness of the algorithm which results from the evaluation of *Develop*. To do this, we prove:

Theorem 3.3.1 *If φ is provable, the evaluation of $Develop(\langle \emptyset, \ell_0, F, \varphi \rangle)$ will terminate.*

Proof: If φ is provable, then there is a standard tableau proof of φ, i.e. a tableau development which is closed on all branches. Call this tableau proof $\mathcal{T}$. We need to show that a tableau (not necessarily the same tableau) is discovered by the evaluation of $T = Develop(\langle \emptyset, \ell_0, F, \varphi \rangle)$ according to the rules above. In particular, we show that the rules applied on a branch in $\mathcal{T}$ are equivalent to ones applied on some branch in T, and that every branch in T contains all consequences of rules applied on some branch in $\mathcal{T}$.

First, since all rules which can always be applied are eventually applied, if a tableau rule present in $\mathcal{T}$ is not applied, it is because the consequences of its application were already present in the formulas at that stage as a result of some other rule application. So, all rules in $\mathcal{T}$ are either applied or have their consequences subsumed by the application of another rule. In either case, the consequences of all applications of rules on some branch in $\mathcal{T}$ are present on some branch in T.

Second, note that although the development of T at a given state may have more branches than $\mathcal{T}$, every branch of T corresponds to some branch of $\mathcal{T}$ in that it contains at least the tagged formulas present on that branch. Whenever a rule creating two branches is applied in T which was not applied in $\mathcal{T}$, both new branches still contain all the information on the single branch in $\mathcal{T}$.

So, since every branch of T represents the consequences of some branch of $\mathcal{T}$, and all branches in $\mathcal{T}$ close, every branch of the development of T closes. $\Box$

In the next section we shall need the following easy lemma:

Lemma 3.3.2 *Suppose ℓ is such that no rule can be applied to any tuple of the form $\langle C, \ell, V, \alpha\rangle$ in S. Then we have the following properties (omitting C and ℓ for brevity):*

$$
\begin{aligned}
T\alpha \wedge \beta \in S &\;\Rightarrow\; T\alpha \in S \textit{ and } T\beta \in S \\
F\alpha \wedge \beta \in S &\;\Rightarrow\; F\alpha \in S \textit{ or } F\beta \in S \\
T\alpha \vee \beta \in S &\;\Rightarrow\; T\alpha \in S \textit{ or } T\beta \in S \\
F\alpha \vee \beta \in S &\;\Rightarrow\; F\alpha \in S \textit{ and } F\beta \in S \\
T\alpha \rightarrow \beta \in S &\;\Rightarrow\; F\alpha \in S \textit{ or } T\beta \in S \\
F\alpha \rightarrow \beta \in S &\;\Rightarrow\; \exists \ell', C'. \ell \leq \ell' \langle C', \ell', T, \alpha\rangle \in S \textit{ and } \langle C', \ell', T, \beta\rangle \in S \\
T\exists x.\alpha(x) \in S &\;\Rightarrow\; \exists c \in C. T\alpha(c) \in S \\
F\exists x.\alpha(x) \in S &\;\Rightarrow\; \forall c \in C. F\alpha(c) \in S \\
T\forall x.\alpha(x) \in S &\;\Rightarrow\; \forall c \in C. T\alpha(c) \in S \\
F\forall x.\alpha(x) \in S &\;\Rightarrow\; \exists \ell', C'. \ell \leq \ell' \textit{ and } \exists c \in C' \langle C', \ell', F, \alpha(c)\rangle \in S
\end{aligned}
$$

Proof: If any of the above conditions failed, the corresponding rule would be applicable. □

3.4 Constructive version of the completeness theorem

Given the problems inherent in reasoning constructively about infinite objects and undecidable properties, it is not surprising that our constructive completeness theorem is fairly weak. Its computational content is a bounded decision procedure. Given a formula φ and a bound n, the algorithm performs an n-depth tableau search for a tableau proof of φ. If all branches of the tableau close at a stage less than or equal to n, then the closed proof is returned. Otherwise, the open branches can be shown to be consistent with a prefix of a countermodel for φ.

This latter condition, while not completely satisfactory, is about the best we can hope for. Since we cannot bound the depth to which the tableau must be searched, we cannot be certain that we have found a countermodel if tableau development rules can still be applied.

Before we prove the theorem, we make the notion of a prefix of a possible countermodel for φ precise:

Definition 3.4.1 *A prefix of a possible countermodel for φ is a finite Kripke model*

$$\langle K, \leq, af, dom \rangle$$

with a minimum state $k_0 \in S$, a subset of K designated as open *states, and a pair of functions Treq and Freq from K to sets of formulas, satisfying*

1. *for all atomic formulas α and for all $k \in K$, if $\alpha \in Treq(k)$ then $af(k, \alpha)$*

2. *for all $k \in K$, $Treq(k) \cap Freq(k) = \emptyset$*

3. *for all open $k \in K$, if we assume k forces all formulas in $Treq(k)$ and does not force any formula in $Freq(k)$, then φ is not forced at k_0.*

Note that if a prefix of a possible countermodel for φ is such that there are no open $s \in S$, we must have that S is a countermodel.

We may now state the constructive completeness theorem.

Theorem 3.4.1

$\forall \varphi . \forall n : N$

$$let\ P = unwind(n, Develop(\{\langle \emptyset, \ell_0, F, \varphi \rangle\}))\ in$$

$$P\ is\ a\ closed\ tableau\ proof\ of\ \varphi\ or$$

$$P\ contains\ a\ description\ of\ a\ prefix\ of\ a\ possible\ countermodel\ for\ \varphi$$

Proof: If all branches of P are closed, it is a closed tableau proof. If not, there is at least one branch which is not closed. We use the information on this branch to produce a prefix of a possible countermodel for φ. There are only finitely many branches, so we can pick any branch which has not closed, and let S be the last stage descriptor on this branch. Let $K = \{L \in Labels | \exists C : CSet, V \in \{T, F\}, \alpha : Wff . \langle C, L, V, \alpha \rangle \in S\}$, let the order on K be the inherited order on *Labels*, let $af(k, \alpha)$ be true when α is atomic and $\exists C . \langle C, k, T, \alpha \rangle \in S$ and let $dom(k)$ be any set C such that $\langle C, k, V, \alpha \rangle \in S$ for some truth value V and formula α. (Recall that all such sets C are the same for any given k.)

Finally let $k_0 = \ell_0$, let the open states in K be the states corresponding to labels in S which match the conditions for the application of some development rule, and let $\mathit{Treq}(k) = \{\alpha | \langle \mathit{dom}(k), k, T, \alpha \rangle \in S\}$ and $\mathit{Freq}(k) = \{\alpha | \langle \mathit{dom}(k), k, F, \alpha \rangle \in S\}$ We now show that with these functions,

$$\langle K, \leq, \mathit{af}, \mathit{dom} \rangle$$

is a prefix of a possible countermodel for φ.

To complete the proof of the theorem, we need:

1. for all atomic formulas α and for all $k \in K$, if $\alpha \in \mathit{Treq}(k)$ then $\mathit{af}(k, \alpha)$. This is true by definition of af.

2. for all $k \in K$, $\mathit{Treq}(k) \cap \mathit{Freq}(k) = \emptyset$. This is true because S was not closed.

3. for all open $k \in K$, if we assume k forces all formulas in $\mathit{Treq}(k)$ and does not force any formula in $\mathit{Freq}(k)$, then φ is not forced at k_0.

   This is the only nontrivial requirement. Given the assumptions, we will actually prove the stronger statement that for all $k \in K$, k forces all formulas in $\mathit{Treq}(k)$ and no formulas in $\mathit{Freq}(k)$. Since $\varphi \in \mathit{FSet}(k_0)$, this suffices to complete the theorem.

   If k is open, then we have k forces all formulas in $\mathit{Treq}(k)$ and no formulas in $\mathit{Freq}(k)$ as a hypothesis, and we are done. So suppose k is not open. Let α be a formula in $\mathit{Treq}(k)$ or $\mathit{Freq}(k)$. The proof that α is forced is by induction on the complexity of α. If α is atomic, it is forced by definition if it is $\mathit{Treq}(k)$, and not forced if it is in $\mathit{Freq}(k)$ because $\mathit{Treq}(k) \cap \mathit{Freq}(k)$ is empty. If α is not atomic, then since k is not open, lemma 3.3.2 applies to S and k if k is treated as a label. The lemma, when interpreted in the context of the model, reduces the requirement that α is forced or not at k to a statement about smaller formulas being forced or not at k, and by the inductive hypothesis we are done.

$\square$

Although it is not a full completeness theorem, it is not vacuous. Because each rule is applied only if the information it adds is not already present, the information in the possible countermodel increases with increasing n.

We also have the following corollary.

Theorem 3.4.2 *If evaluation of Develop*$(\langle\emptyset, \ell_0, F, \varphi\rangle)$ *terminates after n steps, then the resulting finite tree either describes a tableau proof or contains a countermodel for φ.*

Proof: If evaluation of $\mathit{Develop}(L, S, C)$ terminates in n steps, apply the theorem with this n. If all branches are closed, then there is a tableau proof. If not, then there is a finite prefix of a countermodel for φ. Call this prefix $\langle K, \leq, \mathit{af}, \mathit{dom}\rangle$, as in the theorem. Since the evaluation has terminated, there are no open states in this countermodel. Hence, the assumptions of the third condition in the definition of a finite prefix of a countermodel are vacuous, and we have k_0 does not force φ.

$\square$

Unfortunately, we cannot prove that if a finite countermodel exists, the evaluation of *Develop* will terminate. A simple counterexample is the tableau for $F\neg\forall x\exists y\varphi(x,y)$. The actual countermodel has only two states, but the development continually adds constants to the domain of the second state and does not terminate. This is a problem with tableau systems for predicate logic in general, including classical logic [Smu68].

3.5 Alternative theorems

The constructive theorem in the previous section is limited by the difficulty in reasoning about the behavior of an infinite process. All structures are finite and the "prefix of possible countermodel" restricts us to finite approximations of models. If we can use some classical reasoning, however, we can prove stronger but still computational theorems.

In previous completeness proofs, the fan theorem was used to construct a proof from an assumption that all branches of the tableau eventually close. Because the formulation of the completeness proof of the previous section does not require this reasoning step, the fan theorem was not needed. The motivation for the formulation was computational; we wished to describe the tableau procedure. Thus, we would like that any stronger theorems using classical logic still have the same computational property.

One such theorem is a variation on the theorem of the last section which employs classical reasoning to prove a stronger statement about the possible countermodel for φ. It also uses the extended definition of forcing introduced by Veldman and de Swart.

The extended definition of forcing allows states which force $\perp$. The requirement for forcing a negated formula $\neg\varphi$ at a state s then becomes "for all states $s' \geq s$, if s' forces φ then s' forces $\perp$. Note that a state forcing $\neg\varphi$ under the usual definition of Kripke model still forces $\neg\varphi$, so we have not modified the properties of regular models in any way. We say a Kripke model K *explodes* if any state in the model forces $\perp$. A Kripke model K is a countermodel for φ if K forces φ implies K explodes.

Given this extended definition of forcing, we can define the notion of a prefix of a countermodel with respect these models.

Definition 3.5.1 *A prefix of a countermodel for φ is a finite Kripke model*

$$\langle K, \leq, af, dom \rangle$$

with a minimum state $k_0 \in S$, a subset of K designated as open *states, and a pair of functions Treq and Freq from K to sets of formulas, satisfying*

1. *for all atomic formulas α and for all $k \in K$, if $\alpha \in Treq(k)$ then $af(k, \alpha)$*
2. *for all $k \in K$, $Treq(k) \cap Freq(k) = \emptyset$*
3. *for all Kripke models K' containing K and satisfying, for all open k,*
   - *k forces all formulas in $Treq(k)$ in K'*
   - *k forces any formula in $Freq(k)$ in K' implies k forces $\perp$*

   then if φ is forced at k_0, K' explodes.

With this definition, we can state a not-completely-constructive but computational theorem:

Theorem 3.5.1

$$\forall \varphi . \forall n : N$$

$$let\ P = unwind(n, Develop(\{\langle \emptyset, \ell_0, F, \varphi \rangle\}))\ in$$

P is a closed tableau proof of φ or

P describes a prefix of a countermodel for $\{\varphi\}$

This theorem has the same computational content as the completely constructive one; however, its proof uses classical reasoning if the nth development is not a closed proof.

Proof: By theorem 3.4.1, we have either P is a closed tableau proof of φ or P contains a prefix of a possible countermodel for φ. We need only show that if P describes a prefix of a possible countermodel for φ, then the same description is actually a description of a prefix of a countermodel for φ as defined in definition 3.5.1.

The proofs for the first two cases of definition 3.5.1 are as in theorem 3.4.1. Using classical reasoning on the condition "k forces any formula in *Freq*(k) implies k forces $\perp$" in the third case of definition 3.5.1, we can say that either k does not force any formula in *Freq*(k) or k forces $\perp$. By the proof of theorem 3.4.1, we have that if K' and does not force any formula in *Freq*(k), then φ is not forced at k_0. Otherwise, if k forces $\perp$, then K' explodes. $\Box$

The proof of this theorem uses classical reasoning to conclude that, given a particular open branch P of the partial proof tree, either it generates a model which does not force φ, or eventually it closes. In this latter case, the "exploding model" interpretation means that any closed branch generates a model which forces $\perp$ at some state. Hence, such a model explodes. In either case, P describes a prefix of a countermodel. This means that a prefix of a potential countermodel defined in the previous section is always a prefix of a countermodel in this setting. Note that this means we will sometimes have a "countermodel" for provable formulas; however, we will only have exploding countermodels for such formulas. The undecidability arises from the fact that we cannot determine whether or not all extensions of a given prefix of a model explode.

The computational content of this theorem is still a bounded tree search procedure. Since details were not given of an evaluation order for terms in the lazy language in which *Develop* is defined, we do not know whether the branches are evaluated depth first or breadth first. However, since the same finite tableau results either way as long as the choice of rules to apply at a given stage is deterministic, the precise evaluation order is unimportant. No matter what order is chosen, the bound n prevents an unbounded search.

Another type of theorem we can prove using the same construction is

Theorem 3.5.2

$\forall\varphi.\exists P: \ PPT.$

$$\varphi \textit{ provable } \Rightarrow P \textit{ evaluates to a proof of } \varphi \textit{ and}$$

$$\varphi \textit{ not forced in all Kripke models } \Rightarrow P \textit{ evaluates to a countermodel for } \varphi$$

The computational content of this theorem lies in the construction of P, which is just $Develop(\{\langle\emptyset, \ell_0, F, \varphi\rangle\})$, as expected. Again, the proofs of the properties of P need not be constructive in order to obtain the tableau procedure as the computational content. In this case, the theorem does not give a bound on the evaluation of P. This leaves open the possibility that P is evaluated in true depth first search order, unlike the bounded versions of the theorem.

It would be interesting if a version of the theorem could be developed so that the computational content of the proof could be specified to be a pure depth first tableau search. Since the procedure can be non-terminating, this cannot be done constructively. Furthermore, since most axioms added to constructive logic are for the purpose of adding tools for reasoning about terminating procedures, we cannot expect them to be much help. Despite our apparent inability to describe it constructively, a depth first search would be still be useful because the correctness theorem guarantees it will find a proof if a proof exists.

Even if we assume that the formula is provable, we have difficulty in representing depth first search. This arises from the fact that we cannot use the fan theorem to deduce constructively that a branch closes after a finite time from the fact that it cannot be infinite. The only way we can obtain this bound is by evaluating the branch until it closes. In order to reason about this evaluation, we will need a logical tool which, computationally interpreted, allows us to switch from one branch to another if and when evaluation of one branch terminates. Since termination is undecidable, we need a computational method for reasoning about undecidable propositions. In other words, we need a computational interpretation of classical reasoning. This is the subject of the next three chapters.

Chapter 4

Introduction to Call/cc as a Classically Typed Operator

In the next three chapters, we present a method for extracting programs from classical sequent proofs in propositional logic. The term language is typed lambda calculus augmented by a nonlocal control operator. The control operator we use is essentially Scheme's *call/cc* (call-with-current-continuation). In this chapter, we review previous work and describe in general how the control operator represents classical reasoning. We also give an example of its computational use by returning to the tableau procedure of the previous chapter. Details of the extraction of programs from classical sequent proofs are presented in chapter 5, as well as a proof theoretic explanation of the behaviour of the control operator. In chapter 6, we prove some normalization properties of the extraction and use these to give a direct proof of Friedman's conservative extension theorem.

4.1 Previous work

A number of different approaches to the problem of extracting computational content from classical proofs have been developed in the past five years. Perhaps the most directly relevant to computer science is the work related to the typing on nonlocal control operators.

Since the discovery of a classical typing for the control operator $\mathcal{C}$ by Griffin [Gri90], foundational work has been done in expressing these computations as programs in a typed lambda calculus extended by nonlocal control operators. Using the principle of the Curry-Howard isomorphism, if we augment the typed lambda calculus with an operator having the type of a classical axiom (in particular, an axiom strong enough to prove all of classical logic when added to intuitionistic logic), we will be able to extract programs from classical proofs. The question is then whether these programs actually represent a sensible computation.

Murthy answered this question in part in his thesis [Mur90], where he showed that a classical proof of a Π^0_2 sentence φ can be interpreted as a program in an extended lambda calculus which meets the specification φ. He also demonstrated that translations from classical to intuitionistic logic correspond to continuation-passing style translations of programs with control operators to programs without. In [Mur91b], Murthy showed that this interpretation gives a computational meaning to Friedman's conservative extension theorem for classical over constructive arithmetic for Π^0_2 sentences (in [Fri78]).

Barbanera and Berardi have also worked in this area, and have proved a strong normalization result for a language including the control operator $\mathcal{C}$ in [BB92]. Unfortunately, they have some substantial restrictions on the types allowed in the language. Specifically, their language forbids types with strict subtypes of the form $\neg\neg P$, and forbids $\perp$ from appearing on the left hand side of an implication.

Other work in extraction of computational content from classical proofs has been done using different systems of computation. The system of Parigot [Par92] does not use control operators at all. Furthermore, his computation system is confluent; i.e. it can present exactly one piece of evidence from a classical proof. Since classical proofs can contain different pieces of evidence for a single proposition, we feel that this system does not represent all the information in a classical proof.

Perhaps more promising is the work of Girard [Gir91a]. However, although his system LC hints at computational properties of classical proofs, it is unclear exactly what computations result.

4.2 Call/cc as a classical operator

In order to extend the "proofs as programs" principle to classical proofs, we must find a way of representing classical reasoning computationally. To do this, we must examine how classical reasoning differs from intuitionistic reasoning. The law of the excluded middle, i.e. the axiom $P \vee \neg P$, is often considered the essence of the difference between classical and intuitionistic logic. However, when we began this work, we discovered that proof theoretically this is not the case. By examining the rules for both intuitionistic and classical sequent calculus from a computational point of view, it becomes clear that the only rule which is substantially different is the rule for arrow-introduction on the right.

In Gentzen's original presentation of the sequent calculus [Gen69], the only difference between the calculus for intuitionistic logic and that for classical logic is the number of formulas allowed on the right side of the sequent. For intuitionistic logic, at most one formula is allowed. For classical logic, any finite number of formulas is permitted. So, while the intuitionistic interpretation of

$$\Gamma \vdash A$$

is "Under the set of assumptions Γ we can prove the formula A," the classical sequent

$$\Gamma \vdash \Delta$$

is interpreted as "If all formulas in Γ are true, then some formula A in Δ must be true." In classical sequent calculus, then, there are multiple possible conclusions of the sequent and it is not necessary to demonstrate which one has been proved. In other words, the formulas in Δ form an implicit classical disjunction.

However, close inspection of the classical rules reveals that most can be interpreted constructively. That is, given a rule

$$\frac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'}$$

if we have a constructive proof of the hypothesis sequent, i.e. under the set of assumptions Γ we can prove *constructively* the disjunction of the formulas in Δ, then under hypotheses Γ' we can prove, also constructively, the disjunction of the formulas in Δ'.

This is true despite the fact that we may have more than one formula on the right side of the turnstile. For, if a sequent $\Gamma \vdash \Delta$ is constructively provable, given proofs of the formulas in Γ we will be able to determine which formula in Δ is proved. This is usually enough to allow us to determine which formula on the left of the conclusion sequent is proved.

For example, consider the rule for $\vee$ introduction on the left. Since many people think of the difference between classical and intuitionistic logic as resting in the behaviour of disjunction, it is somewhat surprising that the classical rule for disjunction is really constructive. The rule is

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}$$

If we have a constructive proof of $\Gamma \vdash A, \Delta$, then we have a function from elements of the types in Γ to the disjoint union type $A \vee \bigvee \Delta$. Thus, if we apply this function to particular elements of the types in Γ, we can determine the type of the result within the disjunction.

Given this function, it is easy to construct a function with the type corresponding to the conclusion of the sequent. The only modification is that if the result of the first function is of type A within the disjunction, it is injected to the type $A \vee B$.

Even the structural rules can be interpreted constructively in this way. The only rule which cannot be constructively interpreted is the rule of arrow introduction on the left:

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \rightarrow B, \Delta}$$

Since we may have satisfied the hypothesis sequent by proving something in Δ, and we may have used the assumption A, we cannot be certain that we can still prove anything in Δ. Nor can we prove $A \rightarrow B$, since we may not have proved B in the hypothesis sequent. It is only in the computational content of this sequent that the *call/cc* operator appears. *Call/cc* can be typed with the classical tautology

$$((\alpha \rightarrow \bot) \rightarrow \alpha) \rightarrow \alpha,$$

which is a form of Peirce's law. This formula is not provable constructively. Furthermore, when it is added to constructive propositional logic as an axiom, the result is classical propositional logic.

We will give evaluation rules for *call/cc* in the next chapter. Here, we will explain the general behaviour of the operator in this setting by returning to the rule for arrow introduction. For simplicity, let us consider the case where Γ is empty, so the rule is

$$\frac{A \vdash B, \Delta}{\vdash A \rightarrow B, \Delta}$$

Suppose we are given a constructive witness for the premise of the rule, i.e. a function f of type $A \rightarrow B \vee \bigvee \Delta$. If we treat disjunction constructively, in order to provide a witness to $A \rightarrow B \vee \bigvee \Delta$ we must produce a term which is a witness to either $A \rightarrow B$ or to $\bigvee \Delta$. However, we do not have enough information to provide this witness without a witness for A. To solve this dilemma, we use *call/cc* to define a term which, when evaluated, first appears to provide a witness $w_{A \rightarrow B}$ to $A \rightarrow B$. However, this term also contains the current continuation, i.e. a description of the state of the computation at the point where this decision is made. Should the computation proceed in such a way that this witness $w_{A \rightarrow B}$ is applied to a term a type A, the term then applies f (the witness to the premise of the rule) to a obtain a term in $B \vee \bigvee \Delta$. If this term is actually in B, then it is returned and the computation proceeds. In this case, the term $w_{A\to B}$ given for $A\to B$ actually behaves as a normal witness for $A\to B$, as expected. However, if the result of applying the witness to the premise has the type of a formula in $\bigvee\Delta$, we apply the continuation which was captured at the time the decision between $A\to B\vee\bigvee\Delta$ was evaluated to return to this point in the computation with new information – namely, the inhabitant of $\bigvee\Delta$.

To take a more concrete example, consider the law of the excluded middle, or $A\vee\neg A$. Since $\neg A = A\to\bot$, the proof of this in Gentzen's sequent calculus involves the following use of arrow introduction:

$$\frac{A\vdash\bot, A}{\vdash A\to\bot, A}$$

The evaluation of the term inhabiting $A\to\bot\vee A$ captures the current context and produces an alleged inhabitant of $A\to\bot$. Should this function ever be applied, however, it must be applied to something of type A. At this point, the continuation is applied to return the computation to the context in which the decision $A\to\bot\vee A$ was made, and the computation continues with the newly supplied evidence for A instead of $A\to\bot$.

The previous example demonstrates that the evaluation of terms containing *call/cc* is generally only useful when they are in the context of a term which is expected to produce data. Although the term for $A\to\bot\vee A$ can be evaluated on its own, it cannot be evaluated to a form which does not contain continuations. However, when it is used in a term which produces, say, a natural number, the term performs useful computation.

4.3 Application to the tableau procedure

Returning to the tableau procedure of the previous chapter, we can see that a control operator will provide the necessary means of switching from one tableau branch to another when the first branch closes, without the need for explicit backtracking. In fact, control operators were introduced to functional languages for reasons of efficiency. The use of continuations can avoid the need to unwind deep layers of recursive calls when a result is found. This is especially useful in the context of the inductive types described in the previous chapter; we need not explicitly keep track of previous stages in the tableau in order to return to them and evaluate another branch.

More concretely, we can classically prove a theorem of the form

Theorem 4.3.1

$$\begin{array}{l}\forall\varphi.\exists P:\ PPT.\\ \qquad P\ \textit{evaluates to a proof of}\ \varphi\quad \textit{or}\\ \qquad P\ \textit{evaluates to a countermodel for}\ \varphi\end{array}$$

Proof outline: Without delving deeply into the logical structure of the proof, we give an argument that there is a classical proof of this theorem which corresponds computationally to a depth-first search of the tableau using control operators. The control operator is used to "decide" the undecidable disjunction "either there is not a proof, or there is a proof". Under the supposition that there is not a proof, the development describes the construction of a countermodel. Since the countermodel may be infinite, the construction actually describes a process which, in the limit, produces a countermodel. However, should a contradiction arise on one branch, that branch of the development actually represents a proof. In this case, another branch is explored, or, if no other branch is left, we have a proof of φ.

There are two ways the proof of this theorem can be structured.

Proof 1. The first proof involves reasoning about evaluation of the *Develop* function defined in chapter 3. This technique corresponds more closely with the classical theorems stated at the end of the previous chapter, with the classical reasoning wrapped around the constructive *Develop* function.

For this proof, the control operator is really needed only in two places. First, at the outset, so that we can claim immediately that either there is no proof or there is a proof. The result should be a claim that the process of evaluating $Develop(\{F\varphi\})$ constructs a countermodel in its limit, and the computational content of this proof should consist of this evaluation.

Second, at any point where the tableau branches, we need to repeat the reasoning for the two branches. For the first branch, either there is no proof or there is a proof. As at the initial step, the result of this decision should be a claim that evaluating the first branch produces a countermodel, along with the creation of a continuation at this decision. If it turns out a contradiction is found, then there is a proof for the first branch. At this point, the continuation returns control to this point and the second branch is conjectured to produce a countermodel.

Proof 2. The second method corresponds with the technique used in chapter 2, in which the algorithm is extracted directly from the proof. In this case, the proof technique still relies on classical logic but the reasoning is used in the definition of

Develop, so that the *Develop* function which is extracted from the proof is different from that defined above. The proof is technically by ν-induction in order to have a ν_ind form extraction, but uses classical logic to prove every case.

Specifically, the proof follows the same cases as the definition of *Develop*, and in every case we use classical logic to decide whether or not the branch or branches created evaluate to a proof or not. Again, initially the evaluation of the control operator will produce a purported countermodel; should this lead to a contradiction, the continuation will be applied and the computation will return a proof.

The computational content of this proof is another representation of the *Develop* function. During evaluation of this *Develop*, a control operator is evaluated at every stage. This uniformity is due to the use of classical logic to make a decision in every case of the induction. In the first proof, these decisions need only be made when the tableau branches, so these extra control operators are avoided.

In either case, we want the *Develop* tree to be evaluated eagerly, since we do not want a result which contains possible continuation applications. This means that if the resulting program were actually evaluated, there would be three possible outcomes:

1. The development halts and produces a proof.

2. The development halts and produces a finite countermodel.

3. The development does not halt, and in the limit produces a countermodel.

Unlike the constructive theorems which were restricted to reasoning about finite prefixes, we must actually prove in this case that, in the limit, the procedure described produces a countermodel. A proof of this for a similar procedure is found in [Fit69]. Essentially, we can use the procedure to define a sequence of approximations, then (classically) take the union of these approximations as the completed countermodel.

Chapter 5

Extraction from Classical Sequent Proofs

In this chapter, we present the details of our method for extracting programs from classical sequent proofs in propositional logic. First, we define term language we use to express the computations involved. Then, we give an outline of the method of extraction of a term from a sequent proof. Next, we extend the computation system by the addition of typed constants, and we define the reduction rules for program terms built from terms extracted from proofs and these constants. These rules are actually described in the form of a term rewriting machine. In the final section, we describe reduction as proof transformation, leading to a technique for representing continuations as partial proofs. We also examine the correspondence between reduction and cut elimination.

5.1 Introduction

This chapter presents a method for extracting programs from classical sequent proofs in propositional logic. The term language is typed lambda calculus augmented by a nonlocal control operator. The control operator used here is Scheme's *call/cc* (call-with-current-continuation). An advantage of using *call/cc* is that a subject reduction theorem can be proved very easily for this language. Furthermore, we can prove this theorem without resorting to translations on the types involved.

One unusual aspect of this work is the choice of the classical sequent calculus for propositional logic as the proof system. Terms are extracted directly from propositional sequent proofs, rather than via a translation to natural deduction proofs. A technique for representing continuations as partial proofs is developed, and reduction is described as an operation on proofs. This partial proof interpretation sheds different light on the problem of understanding continuations.

The work began as an attempt by the author to understand exactly how cut elimination as reduction compares to normalization of natural deduction proofs, which was inspired in part by Girard's work [Gir91b] on constructive interpretations of classical logic, and in part by considering an extension of the results on the tableau algorithm for intuitionistic propositional calculus ([Und90]) developed in previous chapters.

5.2 Term language for classical logic

Each rule of the classical sequent calculus will correspond to the construction of a term in a term language based on typed lambda calculus plus *call/cc*, a nonlocal control operator. Classical typings for control operators were discovered by Griffin and developed by Murthy ([Gri90,Mur90]). In particular, *call/cc* can be given the type $(\neg P \rightarrow P) \rightarrow P$, a classical tautology which is a form of Peirce's law. The results given here could also be expressed in a term language which uses the control operator $\mathcal{C}$ and the abort operator $\mathcal{A}$ with their operational semantics as described in [FFKD87], at the expense of increased complexity of the proofs. To make this translation, we use the definition

$$call/cc(M) = (\mathcal{C}\lambda k.k(Mk))$$

as in [FFKD87]; this definition also justifies the typing of *call/cc* if we consider the usual typing of $\mathcal{C}$ as $\neg(\neg P) \rightarrow P$.

The basic term language, with the exception of *call/cc*, is a subset of the Nuprl term language [C+86]. We use logical notation for types, and for each type, we describe how to build terms which inhabitant that type from inhabitants of its constituent types. Such terms are called *canonical* because they are not themselves reducible (though their subterms may be).

The types, and some of the terms, are defined as follows:

- Propositional letters $(p, q, r \ldots)$ are types, and their canonical inhabitants are variables (`x, y, z,` ...) of the appropriate type.

- If P_1 and P_2 are types, then $P_1 \wedge P_2$ is a type. If $\mathtt{f}_1$ and $\mathtt{f}_2$ are terms of type P_1 and P_2 respectively, then $\texttt{pair(f}_1\texttt{,f}_2\texttt{)}$ is a canonical inhabitant of $P_1 \wedge P_2$..

- If P_1 and P_2 are types, then $P_1 \vee P_2$ is a type. If $\texttt{f}_1$ and $\texttt{f}_2$ are terms of type P_1 and P_2 respectively, then $\texttt{inl(f}_1\texttt{)}$ and $\texttt{inr(f}_2\texttt{)}$ are canonical inhabitants of $P_1 \vee P_2$.

- If P_1 and P_2 are types, then $P_1 \rightarrow P_2$ is a type, and $\lambda\texttt{x.f}_2$ is a canonical inhabitant of it if $\texttt{x}$ is a variable of type P_1.

- $\perp$ is a type, and it has no canonical inhabitants.

  The type $P \rightarrow \perp$ will be abbreviated $\neg P$.

Also in the language are the following term constructors:

- If $\texttt{t}$ is a term of type $(\neg P) \rightarrow P$, then $\texttt{call/cc(t)}$ is a term of type P.

- If $\texttt{t}$ is a term of type $\perp$, then $\texttt{any}^P\texttt{(t)}$ is a term of type P. The superscript will be omitted when it is deducible from context.

- If $\texttt{t}$ is a term of type $P_1 \vee P_2$, and $\texttt{t}_1$ is a term of type P with free variable $\texttt{u}$ of type P_1, and $\texttt{t}_2$ is a term of type P with free variable $\texttt{v}$ of type P_2, then $\texttt{decide(t; u.t}_1\texttt{; v.t}_2\texttt{)}$ is a term of type P.

- If $\texttt{t}$ is a term of type $P_1 \wedge P_2$, and $\texttt{t}_1$is a term of type P with free variables $\texttt{u}$ of type P_1 and $\texttt{v}$ of type P_2, then $\texttt{spread(t;u,v.t}_1\texttt{)}$ is a term of type P.

- If $\texttt{f}$ is a term of type $P_1 \rightarrow P_2$, and $\texttt{t}$ is a term of type P_1, then $\texttt{apply(f;t)}$ is a term of type P_2. This term will often be abbreviated $\texttt{f(t)}$.

Some abbreviations will be helpful in discussing multiple disjunctions, as $A_1 \vee \ldots \vee A_n$. We shall parse this as $A_1 \vee (A_2 \vee (\ldots (A_{n-1} \vee A_n)))$, and we shall use $\texttt{in}_i\texttt{(x)}$ as an abbreviation for the appropriate sequence of `inl` and `inr` applications such that if $\texttt{x} : A_i$, then $\texttt{in}_i\texttt{(x)} : A_1 \vee \ldots \vee A_n$. To generalize the `decide` term constructor to handle multiple disjunctions, assume we have terms $\texttt{t}_1 \ldots \texttt{t}_n$ of type P such that each $\texttt{t}_i$ has free variable $\texttt{u}_i$ of type A_i, and we have `t` of type $A_1 \vee \ldots \vee A_n$. Then the term $\texttt{decide}_n\texttt{(t;u}_1\texttt{.t}_1\texttt{; ...; u}_n\texttt{.t}_n\texttt{)}$ has type P, and is considered to be an abbreviation of the appropriate sequence of ordinary `decide` constructions.

5.3 Classical sequent proof terms

For each rule of the classical sequent calculus, we describe how the computational content of the hypothesis of the sequent is transformed to computational content of the conclusion. In general, a sequent $A_1, \ldots, A_m \vdash B_1, \ldots, B_n$ will represent the type $A_1 \rightarrow (A_2 \rightarrow \ldots (A_m \rightarrow B_1 \vee \ldots \vee B_n))$, and so its computational content will be a term of that type. Thus, we will assign a variable to each formula on the left of the turnstile, and the inhabitant of the (implicit) disjunction on the right will be constructed from these variables and the term constructors. The computational content of the sequent, then, is the closed λ-term for the function involved.

Extracting terms from proofs

The rest of this section describes how to construct a term representing the computational content of a sequent proof. The term so constructed is called a *proof term.*

The computational content of an axiom

$$\mathtt{x} : A \vdash \mathtt{x} : A$$

is the term $\lambda\mathtt{x}\,.\,\mathtt{x}$, the identity function. A more general form of this axiom is

$$\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n \vdash \mathtt{x}_i : A_i$$

and its computational content is the term $\lambda\mathtt{x}_1 \ldots \lambda\mathtt{x}_n\,.\,\mathtt{x}_i$.

Now assume that we are given functions representing the computational content of the sequents which are the premises of each sequent rule. We use these functions to construct the computational content of the conclusion. The notation is extended so that $\mathtt{f} : \Gamma \vdash \Delta$ means that if $\Gamma = A_1, \ldots, A_n$ and $\Delta = B_1, \ldots, B_n$, $\mathtt{f}$ is a function with type $(A_1 \rightarrow (A_2 \ldots (A_n \rightarrow (B_1 \vee \ldots \vee B_n))))$. No ambiguity will arise, since the type is read this way only if there is no term explicitly shown for the formula on the right side of the turnstile.

Since sequent rules may involve lists of formulas $\Gamma = A_1, \ldots, A_n$ which are carried from hypothesis to conclusion without modification, we will often abbreviate the list $\mathtt{g}_1 : A_1, \ldots, \mathtt{g}_n : A_n$ as simply $\mathtt{g} : \Gamma$. Thus, in the terms below, $\lambda\mathtt{g}\,.\,\mathtt{t}$ is to be considered an abbreviation for $\lambda\mathtt{g}_1 \ldots \lambda\mathtt{g}_n\,.\,\mathtt{t}$. We also abbreviate $((((\mathtt{f}(\mathtt{g}_1))\mathtt{g}_2)\ldots)\mathtt{g}_n)$ as $\mathtt{f}(\mathtt{g})$, and we may write repeated applications $(\mathtt{f}(\mathtt{x}))\mathtt{y}$ as $\mathtt{f}(\mathtt{x},\mathtt{y})$ to avoid proliferation of parentheses.

The simplest cases are when there really is only one formula on the right of the turnstile. The computational content corresponds exactly with the standard interpretation. The rules in this case are the following:

$$\frac{\mathtt{f}_1 : \Gamma \vdash A \qquad \mathtt{f}_2 : \Gamma \vdash B}{\lambda\mathtt{g}.\mathtt{pair}(\mathtt{f}_1(\mathtt{g}), \mathtt{f}_2(\mathtt{g})) : \Gamma \vdash A \wedge B}$$

$$\frac{\mathtt{f} : \Gamma \vdash A}{\lambda\mathtt{g}.\mathtt{inl}(\mathtt{f}(\mathtt{g})) : \Gamma \vdash A \vee B} \qquad \frac{\mathtt{f} : \Gamma \vdash B}{\lambda\mathtt{g}.\mathtt{inr}(\mathtt{f}(\mathtt{g})) : \Gamma \vdash A \vee B}$$

$$\frac{\mathtt{f}_1 : \Gamma \vdash A \qquad \mathtt{f}_2 : \Gamma, B \vdash C}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{f}_2(\mathtt{g}, \mathtt{x}(\mathtt{f}_1(\mathtt{g}))) : \Gamma, A \rightarrow B \vdash C} \quad \frac{\mathtt{f} : \Gamma, A \vdash B}{\lambda\mathtt{g}.(\lambda\mathtt{x}.\mathtt{f}(\mathtt{g}, \mathtt{x})) : \Gamma \vdash A \rightarrow B}$$

There are also cases in which the rule is not dependent on the number of formulas in Δ.

$$\frac{\mathtt{f} : \Gamma, A \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{spread}(\mathtt{x}; \mathtt{u}, \mathtt{v}.\mathtt{f}(\mathtt{g}, \mathtt{u})) : \Gamma, A \wedge B \vdash \Delta}$$

$$\frac{\mathtt{f} : \Gamma, B \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{spread}(\mathtt{x}; \mathtt{u}, \mathtt{v}.\mathtt{f}(\mathtt{g}, \mathtt{v})) : \Gamma, A \wedge B \vdash \Delta}$$

$$\frac{\mathtt{f}_1 : \Gamma, A \vdash \Delta \qquad \mathtt{f}_2 : \Gamma, B \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{decide}(\mathtt{x}; \mathtt{u}.\mathtt{f}_1(\mathtt{g}, \mathtt{u}); \mathtt{v}.\mathtt{f}_2(\mathtt{g}, \mathtt{v})) : \Gamma, A \vee B \vdash \Delta}$$

Finally, there are the cases in which the fact that there are multiple formulas on the right of some sequent is important. The terms extracted from these cases are more complicated because the conclusion has a disjunction type.

$$\frac{\mathtt{f}_1 : \Gamma \vdash A, \Delta \qquad \mathtt{f}_2 : \Gamma \vdash B, \Delta}{\begin{array}{c}\lambda\mathtt{g}.\mathtt{decide}(\mathtt{f}_1(\mathtt{g}); \mathtt{u}.\mathtt{decide}(\mathtt{f}_2(\mathtt{g}); \mathtt{w}.\mathtt{inl}(\mathtt{pair}(\mathtt{u}, \mathtt{w})); \mathtt{x}.\mathtt{inr}(\mathtt{x})); \mathtt{v}.\mathtt{inr}(\mathtt{v})) : \\ \Gamma \vdash A \wedge B, \Delta\end{array}}$$

$$\frac{\mathtt{f} : \Gamma \vdash A, \Delta}{\lambda\mathtt{g}.\mathtt{decide}(\mathtt{f}(\mathtt{g}); \mathtt{u}.\mathtt{inl}(\mathtt{inl}(\mathtt{u})); \mathtt{v}.\mathtt{inr}(\mathtt{v})) : \Gamma \vdash A \vee B, \Delta}$$

$$\frac{\mathtt{f} : \Gamma \vdash B, \Delta}{\lambda\mathtt{g}.\mathtt{decide}(\mathtt{f}(\mathtt{g}); \mathtt{u}.\mathtt{inl}(\mathtt{inr}(\mathtt{u})); \mathtt{v}.\mathtt{inr}(\mathtt{v})) : \Gamma \vdash A \vee B, \Delta}$$

$$\frac{\mathtt{f}_1 : \Gamma \vdash A, \Delta \qquad \mathtt{f}_2 : \Gamma, B \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{decide}(\mathtt{f}_1(\mathtt{g}); \mathtt{u}.\mathtt{f}_2(\mathtt{g}, \mathtt{x}(\mathtt{u})); \mathtt{v}.\mathtt{v}) : \Gamma, A \rightarrow B \vdash \Delta}$$

The case of arrow introduction on the right where there is more than one formula on the right is the only one where assuming the hypothesis is constructive does not imply the conclusion is constructive. It is here we must introduce *call/cc*.

$$\frac{\mathtt{f} : \Gamma, A \vdash B, \Delta}{\lambda\mathtt{g}.\mathtt{call/cc}(\lambda\mathtt{h}.\mathtt{inl}(\lambda\mathtt{x}.\mathtt{decide}((\mathtt{f}(\mathtt{g}))\mathtt{x}; \mathtt{u}.\mathtt{u}; \mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v})))))) : \Gamma \vdash A \rightarrow B, \Delta}$$

To see the intuition behind this term, we consider how it will be used in computation. Suppose Γ is empty, so that we have

$$\mathtt{call/cc}(\lambda\mathtt{h}.\mathtt{inl}(\lambda\mathtt{x}.\mathtt{decide}(\mathtt{f}(\mathtt{x}); \mathtt{u}.\mathtt{u}; \mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v}))))))$$

as a closed term of type $(A \rightarrow B) \vee \Gamma$. If we use this term in a computation of a piece of printable data (i.e. a number, a list, etc.), we must perform a case split to eliminate the disjunction. When the term is evaluated, `h` is bound to the current continuation and we have

$$\texttt{inl}(\lambda\texttt{x}.\texttt{decide}(\texttt{f}(\texttt{x}); \texttt{u}.\texttt{u}; \texttt{v}.\texttt{any}(\texttt{h}(\texttt{inr}(\texttt{v})))))$$

as the term of type $(A{\rightarrow}B) \vee \Gamma$. So, in the case split, we take the branch for $A{\rightarrow}B$ with the function

$$\lambda\texttt{x}.\texttt{decide}(\texttt{f}(\texttt{x}); \texttt{u}.\texttt{u}; \texttt{v}.\texttt{any}(\texttt{h}(\texttt{inr}(\texttt{v}))))$$

as our purported inhabitant of the type $A{\rightarrow}B$. Now suppose this function is applied to a term `a` of type A. Applying the function `f` from the hypothesis of the sequent to `a`, we have an element of $B \vee \Delta$. Should this actually be of type B, then we return that value. If not, we have an inhabitant `v` of Δ. We then return the computation to the stage at which we took the case split for $(A{\rightarrow}B) \vee \Delta$ by applying the continuation `h` to our new evidence for $(A{\rightarrow}B) \vee \Delta$, namely, `inr(v)` where `v` is the inhabitant of Δ produced by `f(a)`. Thus, the continuation operator allows the computation to switch paths when the computation produces further evidence which determines which of the disjuncts is proved. This description will be made more concrete in the next section when the computation rules are defined.

The structural rules also have term extractions, though their computational content is uninteresting.

$$\frac{\mathtt{f} : \Gamma \vdash \Delta}{\lambda\mathtt{g}.\mathtt{inr}(\mathtt{f}(\mathtt{g})) : \Gamma \vdash A, \Delta} \qquad \frac{\mathtt{f} : \Gamma \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{f}(\mathtt{g}) : \Gamma, A \vdash \Delta}$$

$$\frac{\mathtt{f} : \Gamma \vdash A, A, \Delta}{\lambda\mathtt{g}.\mathtt{decide}(\mathtt{f}(\mathtt{g}); \mathtt{u}.\mathtt{inl}(\mathtt{u}); \mathtt{v}.\mathtt{v}) : \Gamma \vdash A, \Delta} \qquad \frac{\mathtt{f} : \Gamma, A, A \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{f}(\mathtt{g}, \mathtt{x}, \mathtt{x}) : \Gamma, A \vdash \Delta}$$

The exchange rules also require treatment, though their computational content is trivial.

$$\frac{\mathtt{f} : \Gamma \vdash A_1, \ldots, A_i, A_{i+1}, \ldots, A_n}{\begin{array}{c}\lambda\mathtt{g}.\mathtt{decide}_n(\mathtt{f}(\mathtt{g}); \mathtt{u}_1.\mathtt{in}_1(\mathtt{u}_1); \ldots; \mathtt{u}_i.\mathtt{in}_{i+1}(\mathtt{u}_i); \mathtt{u}_{i+1}.\mathtt{in}_i(\mathtt{u}_{i+1}); \ldots; \mathtt{u}_n.\mathtt{in}_n(\mathtt{u}_n)) : \\ \Gamma \vdash A_1, \ldots, A_{i+1}, A_i, \ldots, A_n\end{array}}$$

$$\frac{\mathtt{f} : A_1, \ldots, A_i, A_{i+1}, \ldots, A_n \vdash \Delta}{\lambda\mathtt{x}_1 \ldots \lambda\mathtt{x}_{i+1}\lambda\mathtt{x}_i \ldots \lambda\mathtt{x}_n.\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_i, \mathtt{x}_{i+1}, \ldots, \mathtt{x}_n) : A_1, \ldots, A_{i+1}, A_i, \ldots, A_n \vdash \Delta}$$

If we consider terms to represent natural deduction proofs, then this extraction method implicitly gives us a translation from proofs in classical sequent calculus to proofs in natural deduction plus $(\neg P \to P) \to P$. This translation minimizes the number of *call/cc*'s in the translated term, in the sense that there is only one for each use of the classical arrow introduction rule.

5.4 Reduction of terms

In this section we describe a more general term language and present machine rewrite rules for computation in this language. These rewrite rules will define the operational semantics of the language, and it is with respect to this semantics that we will prove that reduction preserves type.

We wish to be able to use these classical proof terms in a general context; for example, we may want to use a classical proof term to represent the propositional logic content of a proof about the integers. To do this, we allow typed nonlogical constants in the term language. We cannot give reduction rules for these constants, since they may represent information about some external theory. We instead require that reductions involving these constants are terminating and type correct, so that constants obey the standard constructive mathematical semantics. For example, if we have a function constant `f` of type $A \rightarrow B$, if we apply `f` to `a` of type A, we are guaranteed that the result `b` is of type B. Furthermore we must be able to treat `b` as we would any ordinary member of type B, so that if $B = B_0 \vee B_1$, we must be able to reduce the term $\mathtt{decide(b;u.t_1;v.t_2)}$. So we actually require that *all* constants which may appear during reduction (not just the constants present at the beginning) have the property that reductions involving them are terminating and type correct.

Given these assumptions on the behavior of the constants under reduction, we define the program term language to be type-correct terms built from constants and proof terms by any of the term constructors described above, except for `call/cc`. Thus, we know that all instances of `call/cc` in the program are from proof terms, and so we know the context in which they occur.

Since we may be using these proof terms to represent the logical content of a proof in a larger theory, we shall call the result an *applied proof term.* Formally, we define an applied proof term as follows:

- Proof terms are applied proof terms. The type of a proof term is the type corresponding to the formula which was proved.
- Typed constants are applied proof terms.
- Typed variables are applied proof terms.
- If $\mathtt{t}_1$ and $\mathtt{t}_2$ are applied proof terms of type P_1 and P_2 respectively, then $\mathtt{pair(t_1,\ t_2)}$ is an applied proof term of type $P_1 \wedge P_2$, and $\mathtt{inl(t_1)}$ and $\mathtt{inr(t_2)}$ are applied proof terms of type $P_1 \vee P_2$.
- If `x` is a variable of type P_1 and `t` is an applied proof term of type P_2, then $\lambda\mathtt{x.t}$ is an applied proof term of type $P_1 \rightarrow P_2$.
- If `t` is an applied proof term of type $\perp$, then $\mathtt{any}^P\mathtt{(t)}$ is an applied proof term of type P. (The superscript will be omitted when it is deducible from context.)
- If `t` is an applied proof term of type $P_1 \vee P_2$, and $\mathtt{t}_1$ is an applied proof term of type P with free variable `u` of type P_1, and $\mathtt{t}_2$ is an applied proof term of type P with free variable `v` of type P_2, then $\mathtt{decide(t;\ u.t_1;\ v.t_2)}$ is an applied proof term of type P.
- If `t` is an applied proof term of type $P_1 \wedge P_2$, and $\mathtt{t}_1$ is an applied proof term of type P with free variables `u` of type P_1 and `v` of type P_2, then $\mathtt{spread(t;u,v.t_1)}$ is an applied proof term of type P.
- If `f` is an applied proof term of type $P_1 \rightarrow P_2$, and `t` is an applied proof term of type P_1, then `apply(f;t)` is an applied proof term of type P_2. This term will often be abbreviated `f(t)`.

An applied proof term is called a *program term* if it has no free variables.

Following the example of [FFKD87], we present the operational semantics as a set of rules representing the transition function of a term rewriting machine. We use their notion of a continuation point to represent the continuation object; however, since we do not fix an evaluation order, we shall use a more general notion of evaluation context (the analogous "applicative contexts" defined in [FFKD87] are defined for a particular evaluation order). An evaluation context is a term with a hole in it, such that the hole is not within the scope of any binding operators (including those in the body of spread and decide terms). A program not in normal form can be split into an evaluation context and a redex. We use the notation $E[R]$ for such a program, where $E[]$ is the evaluation context and R is the redex. A continuation point is an evaluation context tagged with $\mathbf{p}$. If $E[R]$ is a program of type φ and R is of type α, then $\langle \mathbf{p}, E[] \rangle$ is a continuation point which may be applied to any term of type α to produce a program of type φ.

So, the reduction rules for terms are:

$$
\begin{array}{rcl}
E[(\lambda \mathtt{x.t})\mathtt{a}] & \mapsto & E[\mathtt{t}[\mathtt{x} := \mathtt{a}]] \\
E[\mathtt{spread}(\mathtt{pair}(\mathtt{a},\mathtt{b});\mathtt{u},\mathtt{v.t})] & \mapsto & E[\mathtt{t}[\mathtt{u} := \mathtt{a}, \mathtt{v} := \mathtt{b}]] \\
E[\mathtt{decide}(\mathtt{inl}(\mathtt{a});\mathtt{u.t}_1;\mathtt{v.t}_2)] & \mapsto & E[\mathtt{t}_1[\mathtt{u} := \mathtt{a}]] \\
E[\mathtt{decide}(\mathtt{inr}(\mathtt{b});\mathtt{u.t}_1;\mathtt{v.t}_2)] & \mapsto & E[\mathtt{t}_2[\mathtt{v} := \mathtt{b}]] \\
E[\mathtt{call/cc}(\lambda \mathtt{k.t})] & \mapsto & E[\mathtt{t}[\mathtt{k} := \langle \mathbf{p}, E[] \rangle]] \\
E[(\langle \mathbf{p}, E_0[] \rangle)\mathtt{a}] & \mapsto & E_0[\mathtt{a}]
\end{array}
$$

(plus the assumptions described above about reductions involving constants.) In section 5.5.3, we shall introduce another notation for continuation points inspired by the proof theoretic analysis.

The rule for `call/cc` described above, though not fully general, is sufficient since all of the terms extracted from proofs will have the property that the `call/cc` operator will only be applied to terms of the form λ`k.t`.

Note that from these rules alone we can conclude that reduction of type correct programs preserves the type of the whole program.

Theorem 5.4.1 *If* `t` *is a well-typed program term of type* φ*, and* `t` *reduces to* `t'`*, then* `t'` *has type* φ*.*

Proof: The only cases to verify are the rules for reductions of `call/cc` terms, and the application of continuation points. For the case of `call/cc`, if we have $E[$`call/cc(`λ`k.t)`$]$ is of type φ and `call/cc(`λ`k.t)` has type α, we have that λ`k.t` has type $\neg\alpha\rightarrow\alpha$, since the program is well-typed. Internally, then, the continuation point $\langle \mathbf{p}, E[]\rangle$ which is substituted for `k` will have type $\alpha\rightarrow\perp$, but in fact $E[]$ represents a program of type φ with a hole in it for a term of type α. (If we had A-translated the whole program as in [Mur90], this difference between the apparent type of the continuation within the program and the actual type of the continuation in computation would not appear.) So, the term `t[k :=` $\langle \mathbf{p}, E[]\rangle$`]` has type α, so the reduction of `call/cc(`λ`k.t)` preserves the type of the whole term.

For the case of the application of continuation points, we showed above that if `call/cc(`λ`k.t)` has type α, then $E[]$ represents a program of type φ with a hole in it for a term of type α. Since its internal type is $\alpha\rightarrow\perp$, though, we know that if it is applied, it must be applied to a term `a` of type α, since the program is type correct. This reduction results in $E[\mathtt{a}]$, which is a program of type φ because $E[\mathtt{call/cc(t)}]$ was of type φ. $\square$

5.5 Proof-theoretic interpretation of reduction

In this section, we give an interpretation of reduction in an extended sequent calculus. The proof terms we extracted in section proofterms can be interpreted as natural deduction proofs; in effect, that section can be seen as defining a translation from sequent proofs to natural deduction proofs, with the addition of a rule deducing φ from $\neg\varphi\rightarrow\varphi$. We first define an extension to sequent calculus which allows representation of natural deduction style elimination rules. We also define a map from program terms to the extended sequent proof structure. Then, we describe the proofs corresponding to the continuation terms which may arise during execution, including the behavior of such terms when they are reduced.

5.5.1 Expanded sequent calculus

Since terms in the term language correspond more closely to natural deduction proofs than to sequent proofs, in order to reason about properties of terms as properties of proofs we shall extend the notion of proof with additional rules. The result is a hybrid between natural deduction and sequent systems, with the natural deduction steps appearing as intermediate steps in the sequent proof. Reductions will generally correspond to some preliminary rearranging of the proof, followed by a sequence of cut eliminations. The procedure of substitution corresponds to a sequence of cut elimination steps.

We add to the normal collection of sequent rules (including cut) some rules corresponding to elimination of connectives in natural deduction. We shall then have that every redex corresponds to a natural deduction rule or a cut, although not every cut will correspond to a redex.

A decide redex corresponds to a rule of the form:

$$\frac{\vdash \mathtt{c} : A \vee B \qquad \mathtt{u} : A \vdash \mathtt{f}_1 : P \qquad \mathtt{v} : B \vdash \mathtt{f}_2 : P}{\vdash \mathtt{decide}(\mathtt{c}; \mathtt{u}.\mathtt{f}_1; \mathtt{v}.\mathtt{f}_2) : P}$$

This may also appear as

$$\frac{\vdash \mathtt{c} : A, B \qquad \mathtt{u} : A \vdash \mathtt{f}_1 : P \qquad \mathtt{v} : B \vdash \mathtt{f}_2 : P}{\vdash \mathtt{decide}(\mathtt{c}; \mathtt{u}.\mathtt{f}_1; \mathtt{v}.\mathtt{f}_2) : P}$$

because of the implicit disjunction on the right of the turnstile. [1]

Similarly, a spread which may be reduced corresponds to a rule of the form:

$$\frac{\vdash \mathtt{c} : A \wedge B \qquad \mathtt{u} : A, \mathtt{v} : B \vdash \mathtt{f} : P}{\vdash \mathtt{spread}(\mathtt{c}; \mathtt{u}, \mathtt{v}.\mathtt{f}) : P}$$

Finally, the $\rightarrow$-elimination rule corresponds to application:

$$\frac{\vdash \mathtt{a} : A \qquad \vdash \mathtt{f} : A \rightarrow B}{\vdash \mathtt{f}(\mathtt{a}) : B}$$

If `f(a)` is actually a redex, this will reduce to an ordinary cut rule.

[1] In the rest of this chapter, all of the implicit disjunctions occurring on the right side of the turnstile are now considered to be explicit. This allows us to treat the proof as a pseudo-constructive proof with only one conclusion of each sequent.

5.5.2 The map from terms to proofs

Since we wish to reason about properties of terms by reasoning about the proofs from which they are derived, we need to associate a proof tree with each program term. Furthermore, it is helpful if the proof tree structure reflects the term structure, so we need to augment the sequent proof with some redundant information corresponding to the syntax of the term associated with each sequent rule.

First, we describe the modifications needed to make the original sequent proof correspond to the term structure. The abbreviation `f(g)`, used above for repeated applications, requires additional explanation here if each application redex is to correspond to a single cut. In what follows we shall use a notation

$$\frac{\mathtt{g} : \Gamma \vdash \mathtt{g} : \Gamma \qquad \mathtt{f} : \Gamma \vdash \Delta}{\mathtt{g} : \Gamma \vdash \mathtt{f}(\mathtt{g}) : \Delta}$$

for repeated application, where $\mathtt{g} : \Gamma$ is a list $\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n$, and the above cut is an abbreviation for the sequence of cuts

$$\frac{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n \vdash \mathtt{x}_n : A_n \qquad \dfrac{\begin{array}{c}\dfrac{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n \vdash \mathtt{x}_1 : A_1 \qquad \mathtt{f} : A_1, \ldots, A_n \vdash \Delta}{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n, \mathtt{f}(\mathtt{x}_1) : A_2, \ldots, A_n \vdash \Delta} \\ \vdots \end{array}}{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n, \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_{n-1}) : A_n \vdash \Delta}}{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n \vdash \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) : \Delta}$$

where after the first rule, the term $\mathtt{f}(\mathtt{x}_1)$ has type $A_2 \rightarrow (A_3 \rightarrow (\ldots \rightarrow (A_n \rightarrow \Delta)))$, etc. This simply describes the repeated application of `f` to each of the new variables $\mathtt{x}_1 \ldots \mathtt{x}_n$, where each cut corresponds to one application.

We give a few examples of the remaining cases. In general, the structure of the term associated with the conclusion of the rule governs the structure of the proof into which the rule expands. The effect of these transformations is mainly to make function applications and other operations which are implicit in the term explicit in the proof.

For example, the sequent

$$\frac{\mathtt{f} : \Gamma, A \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{spread}(\mathtt{x};\mathtt{u},\mathtt{v}.\mathtt{f}(\mathtt{g},\mathtt{u})) : \Gamma, A \wedge B \vdash \Delta}$$

becomes

$$\frac{\mathtt{x} : A \wedge B \vdash \mathtt{x} : A \wedge B \qquad \dfrac{\mathtt{g} : \Gamma, \mathtt{u} : A, \mathtt{v} : B \vdash \mathtt{g} : \Gamma, \mathtt{u} : A \qquad \mathtt{f} : \Gamma, A \vdash \Delta}{\mathtt{g} : \Gamma, \mathtt{u} : A, \mathtt{v} : B \vdash (\mathtt{f}(\mathtt{g}))\mathtt{u} : \Delta}}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{spread}(\mathtt{x};\mathtt{u},\mathtt{v}.(\mathtt{f}(\mathtt{g}))\mathtt{u})) : \Gamma, A \wedge B \vdash \Delta}$$

As another example, the rule

$$\frac{\mathtt{f} : \Gamma, B \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{spread}(\mathtt{x};\mathtt{u},\mathtt{v}.\mathtt{f}(\mathtt{g},\mathtt{v})) : \Gamma, A \wedge B \vdash \Delta}$$

becomes

$$\frac{\dfrac{\mathtt{g} : \Gamma, \mathtt{u} : A, \mathtt{v} : B \vdash \mathtt{g} : \Gamma, \mathtt{v} : B \qquad \mathtt{f} : \Gamma, B \vdash \Delta}{\mathtt{g} : \Gamma, \mathtt{u} : A, \mathtt{v} : B \vdash (\mathtt{f}(\mathtt{g}))\mathtt{v} : \Delta}}{\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{spread}(\mathtt{x};\mathtt{u},\mathtt{v}.(\mathtt{f}(\mathtt{g}))\mathtt{u})) : \Gamma, A \wedge B \vdash \Delta}$$

There is one case where the expansion is not obvious.

$$\frac{\mathtt{f} : \Gamma, A \vdash B, \Delta}{\lambda\mathtt{g}.\mathtt{call/cc}(\lambda\mathtt{h}.\mathtt{inl}(\lambda\mathtt{x}.\mathtt{decide}((\mathtt{f}(\mathtt{g}))\mathtt{x};\mathtt{u}.\mathtt{u};\mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v})))))) : \Gamma \vdash A \rightarrow B, \Delta}$$

The expansion must mirror the structure of the term, but it is not clear how the call/cc and the associated variable h should be treated. Since the evaluation of a call/cc binds the variable to the current continuation, we shall make this binding explicit by adding an extra cut with a placeholder representing the eventual continuation. We shall call this placeholder `h` as well, and represent it as a constant. The last step in the derivation is then:

$$\frac{\vdash \mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot \qquad \mathtt{g} : \Gamma, \mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot \vdash t : A \rightarrow B, \Delta}{\lambda \mathtt{g}.\mathtt{call/cc}(\lambda \mathtt{h}.t) : \Gamma \vdash A \rightarrow B, \Delta}$$

where

$$t = \mathtt{inl}(\lambda \mathtt{x}.\mathtt{decide}((\mathtt{f}(\mathtt{g}))\mathtt{x}; \mathtt{u}.\mathtt{u}; \mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v})))))$$

In the derivation of the body of the call/cc, the variable `h` remains an assumption along the branch of the proof corresponding to the second branch in the `decide`:

$$\frac{\dfrac{\mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot \vdash \mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot \qquad \dfrac{\mathtt{v} : \Delta \vdash \mathtt{v} : \Delta}{\mathtt{v} : \Delta \vdash \mathtt{inr}(\mathtt{v}) : (A \rightarrow B) \vee \Delta}}{\mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot, \mathtt{v} : \Delta \vdash \mathtt{h}(\mathtt{inr}(\mathtt{v})) : \bot}}{\mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot, \mathtt{v} : \Delta \vdash \mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v}))) : B}$$

The remainder of the derivation of the body of the call/cc is:

$$\frac{\dfrac{\dfrac{\dfrac{\mathtt{g} : \Gamma, \mathtt{x} : A \vdash \mathtt{g} : \Gamma, \mathtt{x} : A \qquad \mathtt{f} : \Gamma, A \vdash B, \Delta}{\mathtt{g} : \Gamma, \mathtt{x} : A \vdash (\mathtt{f}(\mathtt{g}))\mathtt{x} : B, \Delta} \qquad \mathtt{u} : B \vdash \mathtt{u} : B \qquad \textit{(see above)}}{\mathtt{g} : \Gamma, \mathtt{x} : A, \mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot \vdash \mathtt{decide}((\mathtt{f}(\mathtt{g}))\mathtt{x}; \mathtt{u}.\mathtt{u}; \mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v})))) : B}}{\mathtt{g} : \Gamma, \mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot \vdash \lambda \mathtt{x}.\mathtt{decide}((\mathtt{f}(\mathtt{g}))\mathtt{x}; \mathtt{u}.\mathtt{u}; \mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v})))) : A \rightarrow B}}{\mathtt{g} : \Gamma, \mathtt{h} : ((A \rightarrow B) \vee \Delta) \rightarrow \bot \vdash t : A \rightarrow B, \Delta}$$

We have described how to augment the sequent proof with the information needed to model the structure of the proof term. We must now associate similar extended sequent calculus trees with the rest of the term in a similar manner. As previously described, the program term is constructed from terms extracted from sequent proofs, variables, and typed constants, using the usual term constructors. The sequent tree corresponding to the program term is constructed using the type information and the structure of the part of the program term not described by the sequent proof. In essence, we construct a proof of the formula corresponding to the type of the whole program, from assumptions corresponding to the types of the constants.

To describe computation with constants, we associate with each constant a sequent tree of the following form, according to its type. For a constant c of type P, the sequent tree is defined inductively as follows. We shall define a map $[\bullet]$ from sequents to sequent trees, and the tree associated with the constant c: P will be the tree $[\vdash P]$.

$$[\Gamma \vdash P] \quad = \quad \Gamma \vdash P \qquad \text{if } P \text{ is atomic}$$

$$[\Gamma \vdash A \wedge B] \quad = \quad \frac{[\Gamma \vdash A] \qquad [\Gamma \vdash B]}{\Gamma \vdash A \wedge B}$$

$$[\Gamma \vdash A \rightarrow B] \quad = \quad \frac{[\Gamma, A \vdash B]}{\Gamma \vdash A \rightarrow B}$$

$$[\Gamma \vdash \neg A] \quad = \quad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}$$

Finally, we define

$$[\Gamma \vdash A \vee B] \quad = \quad \frac{[\Gamma \vdash A]}{\Gamma \vdash A \vee B} \quad \text{or} \quad \frac{[\Gamma \vdash B]}{\Gamma \vdash A \vee B}$$

though we cannot know which until the a cut with $\Gamma \vdash A \vee B$ is actually eliminated. For example, this case arises if we have a constant c of type $A \rightarrow (B \vee C)$. We cannot know the type of the result until c has been applied; since we assume that the evaluation of constant functions terminates and is constructively type correct, we will be able to decide what type the result of a particular application of c actually has.

The above definitions correspond to a tableau proof development except that formulas on the left of the turnstile are not broken down. This is because formulas on the left are arguments to the constant function, and the function must be applied to arguments of the appropriate type.

At this stage, we have sequent trees associated with constant terms and with the term t_0 extracted from the original sequent proof. We now define a map from program terms built from these constants and proof terms to sequent trees. The map will be denoted $\{\bullet\}_D$, where D is a list of the typed bound variables at any point in the term.

$$\{t_0\}_D \quad = \quad \text{the augmented sequent proof associated with } t_0\text{, if } t_0 \text{ is a proof term}$$

$$\{\mathtt{c} : P\}_D \quad = \quad [\vdash P]$$

$$\{\mathtt{pair}(\mathtt{x},\mathtt{y}) : A \wedge B\}_D \quad = \quad \frac{\{\mathtt{x} : A\}_D \qquad \{\mathtt{y} : B\}_D}{D \vdash \mathtt{pair}(\mathtt{x},\mathtt{y}) : A \wedge B}$$

$$\{\mathtt{inl}(\mathtt{x}) : A \vee B\}_D \quad = \quad \frac{\{\mathtt{x} : A\}_D}{D \vdash \mathtt{inl}(\mathtt{x}) : A \vee B}$$

$$\{\mathtt{inr}(\mathtt{x}) : A \vee B\}_D \quad = \quad \frac{\{\mathtt{x} : B\}_D}{D \vdash \mathtt{inr}(\mathtt{x}) : A \vee B}$$

$$\{\lambda\mathtt{x}.\mathtt{t} : A \rightarrow B\}_D \quad = \quad \frac{\{\mathtt{t} : B\}_{\mathtt{x}:A,D}}{D \vdash \lambda\mathtt{x}.\mathtt{t} : A \rightarrow B}$$

$$\{\mathtt{decide}(\mathtt{d};\mathtt{u}.\mathtt{t}_1;\mathtt{v}.\mathtt{t}_2) : P\}_D \quad = \quad \frac{\{\mathtt{d} : A \vee B\}_D \quad \{\mathtt{t}_1 : P\}_{\mathtt{u}:A,D} \quad \{\mathtt{t}_2 : P\}_{\mathtt{v}:B,D}}{D \vdash \mathtt{decide}(\mathtt{d};\mathtt{u}.\mathtt{t}_1;\mathtt{v}.\mathtt{t}_2) : P}$$

$$\{\mathtt{spread}(\mathtt{p};\mathtt{u},\mathtt{v}.\mathtt{t}) : P\}_D \quad = \quad \frac{\{\mathtt{p} : A \wedge B\}_D \qquad \{\mathtt{t} : P\}_{\mathtt{u}:A,\mathtt{v}:B,D}}{D \vdash \mathtt{spread}(\mathtt{p};\mathtt{u},\mathtt{v}.\mathtt{t}) : P}$$

$$\{\mathtt{apply}(\mathtt{f};\mathtt{x}) : B\}_D \quad = \quad \frac{\{\mathtt{x} : A\}_D \qquad \{\mathtt{f} : A \rightarrow B\}_D}{D \vdash \mathtt{apply}(\mathtt{f};\mathtt{x}) : B}$$

$$\{\mathtt{any}(\mathtt{x}) : A\}_D \quad = \quad \frac{\{\mathtt{x} : \bot\}_D}{D \vdash \mathtt{any}(\mathtt{x}) : A}$$

5.5.3 Reduction as proof transformation

In this section, we describe the correspondence between reduction steps in the program term and cut elimination steps in the proof.

Most reduction steps in the program will correspond to an initial step which creates a cut, followed by a sequence of cut elimination steps. This is because substitution of a term for a variable involves passing through the proof tree until the place where the variable is introduced (as part of an axiom sequent) is found, and substituting a proof for that axiom. Note that this corresponds more closely with the actual complexity of substitution, since the proof tree corresponds closely to the syntactical structure of the term. Most reduction steps follow a pattern of setting up a simple cut and eliminating it, in a sequence of cut elimination steps.

In the remainder of this section, we describe, for each kind of reduction, how reduction transforms the proof.

Note that the first premise sequent of a redex must be a sequent with no hypotheses, since we do not allow reductions within the scope of a binding operator.

Simple cuts.

Simple cuts arise from the reduction of any of the more complex rules. The elimination of a cut

$$\frac{\vdash \mathtt{a} : A \qquad \mathtt{x} : A \vdash \mathtt{f} : B}{\vdash (\lambda \mathtt{x}.\mathtt{f})\mathtt{a} : B}$$

corresponds to β-reduction, so we must describe how the argument $\mathtt{a}{:}A$ comes to be substituted for $\mathtt{x}$ in $\mathtt{f}$. Since the structure of the term corresponds so closely with the structure of the proof tree, in order for the substitution of the argument for the bound variable to take place, the argument must be propagated towards the leaves of the tree to the place where the variable is introduced in an axiom sequent. Observe that this is almost immediate in the terms which come directly from the sequent proof, because these terms are constructed via application from the term(s) representing the hypothesis. For general terms, however, this is not true, and it may take several cut elimination steps for the argument to reach the places where the variable was introduced. So we must pass the cut with $\mathtt{a}{:}A$ towards the leaves of the sequent tree until it reaches the point where the variable $\mathtt{x}$ was introduced as an axiom.

The procedure is described by induction on the structure of the proof of $\mathtt{x} : A \vdash \mathtt{f} : B$. To eliminate a cut with an axiom sequent, as

$$\frac{\vdash \mathtt{a} : A \qquad \mathtt{x} : A, \mathtt{g} : \Gamma \vdash \mathtt{x} : A}{\Gamma \vdash A}$$

(Γ may of course be empty), replace this cut with

$$\Gamma \vdash \mathtt{a} : A.$$

To eliminate a cut with a sequent not an axiom, suppose that $\Gamma_0 \vdash \Delta_0$ and $\Gamma_1 \vdash \Delta_1$ are the hypotheses of the sequent rule resulting in $\mathtt{x} : A, \mathtt{g} : \Gamma \vdash \mathtt{d} : \Delta$, so that we have

$$\frac{\Gamma_0 \vdash \Delta_0 \qquad \Gamma_1 \vdash \Delta_1}{\mathtt{x} : A, \mathtt{g} : \Gamma \vdash \mathtt{d} : \Delta}$$

First, replace the conclusion of the cut rule with with $\mathtt{g} : \Gamma \vdash \mathtt{d} : \Delta$. If $\mathtt{x}{:}\ A$ is not in either Γ_0 or Γ_1, then we are done. Otherwise, continue the process recursively with the cuts

$$\frac{\vdash \mathtt{a} : A \qquad \Gamma_0 \vdash \Delta_0}{\Gamma_0 - \mathtt{x} : A \vdash \Delta_0} \quad \text{and} \quad \frac{\vdash \mathtt{a} : A \qquad \Gamma_1 \vdash \Delta_1}{\Gamma_1 - \mathtt{x} : A \vdash \Delta_1}$$

(assuming both are applicable, i.e. $\mathtt{x} : A$ is in both Γ_0 and Γ_1).

In other words, we replace the cut

$$\frac{\vdash \mathtt{a} : A \qquad \dfrac{\Gamma_0 \vdash \Delta_0 \qquad \Gamma_1 \vdash \Delta_1}{\mathtt{x} : A, \mathtt{g} : \Gamma \vdash \mathtt{d} : \Delta}}{\Gamma \vdash \Delta}$$

with

$$\frac{\dfrac{\vdash \mathtt{a} : A \qquad \Gamma_0 \vdash \Delta_0}{\Gamma_0 - \mathtt{x} : A \vdash \Delta_0} \qquad \dfrac{\vdash \mathtt{a} : A \qquad \Gamma_1 \vdash \Delta_1}{\Gamma_1 - \mathtt{x} : A \vdash \Delta_1}}{\mathtt{g} : \Gamma \vdash \mathtt{d} : \Delta}$$

and continue the cut elimination process with these new cuts until an axiom is reached. We assume the substitution occurs in `d` when the cut elimination process reaches the axioms.

Note that although we seem to be introducing new sequents $\Gamma_0 - \mathtt{x} : A \vdash \Delta_0$ and $\Gamma_1 - \mathtt{x} : A \vdash \Delta_1$, in fact these are eliminated at the next stage in the process. In general, the elimination of a cut removes its conclusion sequent and replaces it with another sequent proving the same formula.

Reduction of `decide`

There are two cases, which reduce in essentially the same way.

1. $\mathtt{decide(inl(a);u.t_1;v.t_2)}$

The redex $\mathtt{decide(inl(a);u.t_1;v.t_2)}$ corresponds to a proof segment of the form

$$\dfrac{\dfrac{\vdash \mathtt{a} : A}{\vdash \mathtt{inl(a)} : A \vee B} \qquad \mathtt{u} : A \vdash \mathtt{t}_1 : P \qquad \mathtt{v} : B \vdash \mathtt{t}_2 : P}{\vdash \mathtt{decide(inl(a);u.t_1;v.t_2)} : P}$$

This first reduces to

$$\dfrac{\vdash \mathtt{a} : A \qquad \mathtt{u} : A \vdash \mathtt{t}_1 : P}{\vdash P}$$

and then the process described above for simple cuts is performed.

2. $\mathtt{decide(inr(b);u.t_1;v.t_2)}$

The redex $\mathtt{decide(inr(b);u.t_1;v.t_2)}$ reduces similarly to the previous case.

$$\dfrac{\dfrac{\vdash \mathtt{b} : B}{\vdash \mathtt{inr(b)} : A \vee B} \qquad \mathtt{u} : A \vdash \mathtt{t}_1 : P \qquad \mathtt{v} : B \vdash \mathtt{t}_2 : P}{\vdash \mathtt{decide(inr(b);u.t_1;v.t_2)} : P}$$

This first reduces to

$$\dfrac{\vdash \mathtt{b} : B \qquad \mathtt{v} : B \vdash \mathtt{t}_2 : P}{\vdash P}$$

Then, this cut is eliminated according to the rules for simple cuts.

Reduction of `spread`

The redex `spread(pair(a,b);u,v.t)` corresponds to the proof fragment

$$\dfrac{\dfrac{\vdash \mathtt{a} : A \qquad \vdash \mathtt{b} : B}{\vdash \mathtt{pair(a,b)} : A \wedge B} \qquad \mathtt{u} : A, \mathtt{v} : B \vdash \mathtt{t} : P}{\vdash \mathtt{spread(pair(a,b);u,v.t)} : P}$$

After reduction, the proof becomes

$$\dfrac{\vdash \mathtt{b} : B \qquad \dfrac{\vdash \mathtt{a} : A \qquad \mathtt{u} : A, \mathtt{v} : B \vdash \mathtt{t} : P}{\mathtt{v} : B \vdash P}}{\vdash P}$$

Then, these simple cuts are eliminated. Note that the sequent $B \vdash P$ has been created, but that it disappears after the cut of B is eliminated, so that after the reduction we have a proof of

$$\vdash \mathtt{t}[\mathtt{u} := \mathtt{a}, \mathtt{v} := \mathtt{b}] : \mathtt{P}.$$

Reduction of applications

An application has the following form:

$$\dfrac{\vdash \mathtt{a} : A \qquad \vdash \lambda\mathtt{x.f} : A \to B}{\vdash (\lambda\mathtt{x.f})\mathtt{a} : B}$$

Since we have λ`x.f` as the inhabitant of type $A \to B$, the previous rule must have been an arrow introduction on the right. Hence we must have

$$\dfrac{\vdash \mathtt{a} : A \qquad \dfrac{\mathtt{x} : A \vdash \mathtt{f} : B}{\vdash \lambda\mathtt{x}.\mathtt{f} : A{\rightarrow}B}}{\vdash (\lambda\mathtt{x}.\mathtt{f})\mathtt{a} : B}$$

This reduces to

$$\dfrac{\vdash \mathtt{a} : A \qquad \mathtt{x} : A \vdash \mathtt{f} : B}{\vdash (\lambda\mathtt{x}.\mathtt{f})\mathtt{a} : B}$$

without any change in the term itself, and then this cut is eliminated according to the rules for simple cuts above.

Continuation proofs

To model the capture and application of continuations in the language, we shall define the notion of a continuation proof. Informally, this will be a partial proof which requires another proof to be complete; in essence, it will be a function from proofs to proofs. For any given proof of a formula φ, we will have that any continuation proof arising in the reduction of that proof will, when the continuation proof is applied according to the rules described here, result in a different proof of φ.

Under the ordinary intuitionistic interpretation of the logical connectives, a term representing a proof of an implication $A{\rightarrow}B$ can be considered a function from proofs of A to proofs of B. A continuation proof term is also a function from proofs to proofs; however, it behaves differently when applied within the context of another proof. Suppose we have a proof of a formula φ. A continuation (sub)proof within that proof will appear to be a proof of some formula $A{\rightarrow}\bot$, and the corresponding continuation proof term will appear to have the type $A{\rightarrow}\bot$.

Should that term actually be applied, however (in order to produce an inhabitant of $\perp$), what results is not a term of type $\perp$ within a term of type φ, but rather a new term of type φ, corresponding to a new proof of φ. (Under A-translation, the $\perp$ in the proof would have been translated to a φ already.)

The notation we shall use for the continuation proof terms is intended to hint at their role as functions from proofs to proofs. However, we do not wish to consider the internal structure of the continuation proof corresponding to the term. Accordingly, we denote a continuation proof term of type $A \rightarrow \perp$ as

$$\Lambda \mathtt{x}.[]$$

(This notation replaces the previous "continuation point" notation.) This term represents a proof of φ with a "hole" of type A in it, so that if $\Lambda \mathtt{x}.[]$ is applied to an argument of type A, the result is a new proof of φ, i.e. a new program of type φ. Describing a continuation proof itself is trickier. Associated with every continuation proof term $\Lambda \mathtt{x}.[]$ is a proof which determines the result when the continuation term is applied. To describe this incomplete proof, we use an ordinary sequent proof, except that at one leaf, instead of an identity axiom, we have the sequent

$$\vdash \mathtt{x} : A$$

as an axiom. When the continuation proof is applied, this sequent is replaced with a real proof of $\vdash A$, and the term associated with this proof of A is substituted for $\mathtt{x}$. (In a language using $\mathcal{C}$ and $\mathcal{A}$ this would be an ordinary application followed by an abort; since we are using `call/cc`, however, the application and the abort are performed in one step.)

Continuation proofs arise from the reduction of a term `call/cc(t)`. In this system, the only way such a term arises is as the result of an $\rightarrow$ introduction rule on the right, when there is more than one formula on the right. Thus we know that the form of the proof leading to the term `call/cc(t)` is as described in the section 5.5.2. The redex itself appears as the sequent

$$\cfrac{\cfrac{\vdash \mathtt{h}:((A\rightarrow B)\vee\Delta)\rightarrow\bot \qquad \mathtt{h}:((A\rightarrow B)\vee\Delta)\rightarrow\bot\vdash t: A\rightarrow B,\Delta}{\vdash \mathtt{call/cc}(\lambda\mathtt{h}.\mathtt{inl}(\lambda\mathtt{x}.\mathtt{decide}(\mathtt{f}(\mathtt{x});\mathtt{u}.\mathtt{u};\mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v})))))) : A\rightarrow B,\Delta} \qquad \alpha}{\beta}$$

where

$$t = \mathtt{inl}(\lambda\mathtt{x}.\mathtt{decide}(\mathtt{f}(\mathtt{x});\mathtt{u}.\mathtt{u};\mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v}))))),$$

and where α represents the proof of the other hypotheses (if any) of the rule at this point, β represents the rest of the proof below this rule, and the proof above the call/cc is as previously described in section 5.5.2.

Reduction of this term creates the continuation proof

$$\cfrac{\vdash \mathtt{d} : A\rightarrow B\vee\Delta \qquad \alpha}{\beta}$$

where `d` is a new variable. The continuation proof term associated with this proof is just $\Lambda\mathtt{d}.[]$, and it has type $((A\rightarrow B)\vee\Delta)\rightarrow\bot$. This continuation proof term is then passed as an argument to the term

$$(\lambda\mathtt{h}.\mathtt{inl}(\lambda\mathtt{x}.\mathtt{decide}(\mathtt{f}(\mathtt{x});\mathtt{u}.\mathtt{u};\mathtt{v}.\mathtt{any}(\mathtt{h}(\mathtt{inr}(\mathtt{v})))))),$$

where it is substituted for `h`. When the dust clears and the reduction of the original `call/cc` is complete, the proof structure is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{x}:A\vdash \mathtt{x}:A \qquad \mathtt{f}:A\vdash B,\Delta}{\mathtt{x}:A\vdash \mathtt{f(x)}:B,\Delta} \quad \mathtt{u}:B\vdash \mathtt{u}:B \quad \dfrac{\dfrac{\vdash(\Lambda\mathtt{d}.[]):((A\to B)\vee\Delta)\to\bot \qquad \mathtt{v}:\Delta\vdash \mathtt{inr(v)}:(A\to B)\vee\Delta}{\mathtt{v}:\Delta\vdash(\Lambda\mathtt{d}.[])(\mathtt{inr(v)}):\bot}}{\mathtt{v}:\Delta\vdash \mathtt{any}((\Lambda\mathtt{d}.[])(\mathtt{inr(v)})):B}}{\mathtt{x}:A\vdash \mathtt{decide(f(x);u.u;v.any}((\Lambda\mathtt{d}.[])(\mathtt{inr(v)}))):B}}{\vdash\lambda\mathtt{x.decide(f(x);u.u;v.any}((\Lambda\mathtt{d}.[])(\mathtt{inr(v)}))):A\to B}}{\vdash\mathtt{inl}(\lambda\mathtt{x.decide(f(x);u.u;v.any}((\Lambda\mathtt{d}.[])(\mathtt{inr(v)})))):A\to B,\Delta} \qquad \alpha}{\beta}$$

We now describe how reduction behaves with continuation proof terms. Since continuation terms arise only in the context shown above, if the term is in a redex, it must be that we have taken the right branch of the decide with `f(a)` of type Δ substituted for `v`. So, the proof structure must be

$$\dfrac{\vdash(\Lambda\mathtt{d}.[]) : ((A\to B)\vee\Delta)\to\bot \qquad \vdash\mathtt{inr(f(a))} : (A\to B)\vee\Delta}{\vdash(\Lambda\mathtt{d}.[])(\mathtt{inr(f(a))}) : \bot}$$

Since this is a continuation term, the reduction of $(\Lambda$`d.[])(inr(f(a)))` transforms the whole proof to

$$\dfrac{\vdash\mathtt{inr(f(a))} : A\to B\vee\Delta \qquad \alpha}{\beta}$$

where the context of this sequent is the context described above, so that the proof reverts to the stage at which the `call/cc` term was reduced, with the additional information from the proof of $\vdash\mathtt{f(a)} : \Delta$.

Chapter 6

Properties of the Extraction

In this chapter we present some normalization results for the terms extracted from sequent proofs by the method of the previous chapter. We use these results to prove a variation on Murthy's computational interpretation of Friedman's conservative extension result, which states that if a theorem of the form $\forall x \exists y. R(x, y)$ is provable in Peano (classical) arithmetic (PA), then it is provable in Heyting (constructive) arithmetic (HA). We prove Friedman's theorem by showing that for any classical proof in Peano arithmetic, we can extract a term using the system of the previous chapter, and then normalize it to produce a constructive proof.

6.1 Normalization

We prove two normalization results for the language of the previous chapter. Both proofs are by reduction to a language with control operator $\mathcal{C}$, using the translation

$$\mathit{call/cc}(\lambda h.M) = \mathcal{C}(\lambda k.k((\lambda h.M)k))$$

The results for the language with $\mathcal{C}$ are from Murthy's thesis [Mur90], where the proof is by the soundness of the $\mathcal{C}$-representation of the continuation passing style computation. The argument originated in [Plo75].

To prove the result in our context for a given reduction strategy, we will show that, under that strategy, each reduction step of the *call/cc* reduction is modelled by a sequence of reductions (under the same reduction strategy) of the corresponding $\mathcal{C}$ term. The evaluation rule for $\mathcal{C}$ is

$$E[\mathcal{C}(M)] \rightarrow_1 M(\lambda x.\mathcal{A}(E[x]))$$

and the rule for $\mathcal{A}$ (introduced by evaluation of $\mathcal{C}$) is

$$E[\mathcal{A}(M)] \rightarrow_1 M$$

In both normalization proofs, we use implicitly the following property of the reduction strategy:

Definition 6.1.1 *A reduction strategy is* persistent *if, when M is a redex, $E[M]$ reduces to $E[M']$ and M' is a redex, then M' is the next redex chosen for reduction.*

Note that call-by-name and call-by-value reduction strategies are persistent.

6.1.1 call-by-value normalization

In this section, we prove

Theorem 6.1.1 *Call-by-value evaluation of a program term terminates.*

Proof: Using the definition

$$call/cc(\lambda h.M) = \mathcal{C}(\lambda k.k((\lambda h.M)k)),$$

we need to show that if $E[\mathcal{C}(\lambda k.k((\lambda h.M)k))]$ normalizes under call-by-name reduction strategy, so does $E[call/cc(\lambda h.M)]$.

We demonstrate this by direct computation. For the *call/cc* case, we have

$$E[call/cc(\lambda h.M)] \rightarrow_1 E[M[h := (\lambda d.\mathcal{A}[E[d]])]].$$

Compare this to the evaluation of the term with $\mathcal{C}$:

$$\begin{aligned}
E[\mathcal{C}(\lambda k.k((\lambda h.M)k))] &\rightarrow_1 (\lambda k.k((\lambda h.M)k))(\lambda d.A[E[d]]) \\
&\rightarrow_1 (\lambda d.\mathcal{A}[E[d]])((\lambda h.M)(\lambda d.\mathcal{A}[E[d]])) \\
&\rightarrow_1 (\lambda d.\mathcal{A}[E[d]])(M[h := (\lambda d.\mathcal{A}[E[d]])])
\end{aligned}$$

Now if $(M[h := (\lambda d.\mathcal{A}[E[d]])])$ is a value, we have the following reductions:

$$\begin{aligned}
(\lambda d.\mathcal{A}[E[d]])(M[h := (\lambda d.\mathcal{A}[E[d]])]) &\rightarrow_1 \mathcal{A}[E[(M[h := (\lambda d.\mathcal{A}[E[d]])])]] \\
&\rightarrow_1 E[(M[h := (\lambda d.\mathcal{A}[E[d]])])]
\end{aligned}$$

which is the result of the *call/cc* term evaluation.

If $(M[h := (\lambda d.\mathcal{A}[E[d]])])$ is not a value, then by persistence the rules applied to the *call/cc* term result $E[M[h := (\lambda d.\mathcal{A}[E[d]])]]$ are those which reduce $M[h :=$

$(\lambda d.\mathcal{A}[E[d]])]$. Similarly, because the reduction strategy is call-by-value, the same rules are applied to that subterm in the result of the evaluation of the $\mathcal{C}$ term, namely $(\lambda d.\mathcal{A}[E[d]])(M[h := (\lambda d.\mathcal{A}[E[d]])])$.

There are two possibilities. If any reduction in $(M[h := (\lambda d.\mathcal{A}[E[d]])])$. is an evaluation of a subterm of the form $\mathcal{A}[M']$, then in both cases the result is M', and therefore the *call/cc* term normalizes because the $\mathcal{C}$ term does.

If no $\mathcal{A}$ operators are evaluated in the reduction of $(M[h := (\lambda d.\mathcal{A}[E[d]])])$, then this term reduces to a value M'. We then have the *call/cc* term evaluating to $E[M']$, and the $\mathcal{C}$ term evaluating to

$$\begin{aligned}
(\lambda d.\mathcal{A}[E[d]])(M[h := (\lambda d.\mathcal{A}[E[d]])]) &\rightarrow (\lambda d.\mathcal{A}[E[d]])M' \\
&\rightarrow_1 \mathcal{A}[E[M']] \\
&\rightarrow_1 E[M']
\end{aligned}$$

so again the *call/cc* term must normalize. □

6.1.2 call-by-name normalization

In this section, we prove

Theorem 6.1.2 *Call-by-name evaluation of a program term terminates.*

Proof: Again, to prove the theorem we want to show that if $E[\mathcal{C}(\lambda k.k((\lambda h.M)k))]$ normalizes under call-by-name reduction strategy, so does $E[\mathit{call/cc}(\lambda h.M)]$. In this case, the result is easier since the outermost redex is always evaluated.

The one step reduction of the *call/cc* term is:

$$E[\mathit{call/cc}(\lambda h.M)] \rightarrow_1 E[M[h := (\lambda d.\mathcal{A}[E[d]])]]$$

and the $\mathcal{C}$ term is reduced in four steps to the same result:

$$\begin{aligned} E[C(\lambda k.k((\lambda h.M)k))] &\rightarrow_1 (\lambda k.k((\lambda h.M)k))(\lambda d.\mathcal{A}[E[d]]) \\ &\rightarrow_1 (\lambda d.\mathcal{A}[E[d]])((\lambda h.M)(\lambda d.\mathcal{A}[E[d]])) \\ &\rightarrow_1 \mathcal{A}[E[M[h := (\lambda d.\mathcal{A}[E[d]])]]] \\ &\rightarrow_1 E[M[h := (\lambda d.\mathcal{A}[E[d]])]] \end{aligned}$$

6.2 A direct proof of Friedman's theorem

Using the results of the last section, we can prove directly a computational version of Friedman's theorem:

Theorem 6.2.1 (Friedman) *If PA proves $\exists x.P(x,y)$, where $P(x,y)$ is quantifier free, then HA proves $\exists x.P(x,y)$.*

Since y is free in $\exists x.P(x,y)$, in fact we have the theorem for all Π^0_2 sentences $\forall y \exists x.P(x,y)$.

Before we can prove this theorem here, we must describe how arithmetic is added to the proof system. This is a straightforward representation of the terms and axioms of arithmetic.

6.2.1 Definition of arithmetic

Heyting arithmetic and Peano arithmetic differ only in the logic used to reason about arithmetic terms. Friedman's theorem shows that for Π^0_2 sentences, this difference is not significant.

The language of arithmetic has two components: the term language and the formula language. The term language is defined as follows.

$$
\begin{array}{rcl}
\mathit{Term} & = & \mathit{Var} \\
& \mid & \mathit{Term}_1 + \mathit{Term}_2 \\
& \mid & \mathit{Term}_1 \times \mathit{Term}_2 \\
& \mid & f(\mathit{Term}_1, \ldots, \mathit{Term}_n) \\
& & (\textit{where } f \textit{ is an } n\textit{-ary primitive recursive function symbol}) \\
& \mid & s(\mathit{Term}) \\
& & (\textit{where } s \textit{ is the successor function}) \\
& \mid & 0
\end{array}
$$

In this definition, variables are intended to range over integers.

The formula language is defined as follows.

$$
\begin{array}{rcl}
\mathit{Formula} & = & \mathit{Term} = \mathit{Term} \\
& \mid & \mathit{Formula} \wedge \mathit{Formula} \\
& \mid & \mathit{Formula} \vee \mathit{Formula} \\
& \mid & \mathit{Formula} \rightarrow \mathit{Formula}
\end{array}
$$

$$
\begin{array}{ll}
| & \exists \mathit{Var}.\mathit{Formula} \\
| & \forall \mathit{Var}.\mathit{Formula} \\
| & \bot
\end{array}
$$

We add the following axioms to the logic:

$$
\begin{aligned}
x &= x \\
(s(0) = 0) &\rightarrow \bot \\
x + 0 &= x \\
x + s(y) &= s(x + y) \\
x \times 0 &= 0 \\
x \times s(y) &= x + x \times y
\end{aligned}
$$

Axioms similar to those for $\times$ and $+$ may be added for any primitive recursive function. We also add the following axiom schemas:

$$A(x) \wedge x = y \rightarrow A(y)$$

$$A(0) \wedge \forall x.(A(x) \rightarrow A(S(x))) \rightarrow \forall x.A(x)$$

where $A(x)$ denotes an element of *Formula* with free variable x, and $A(0)$ denotes a formula $A(x)$ with 0 substituted for x.

The first set of axioms have no computational content; this is expressed in Nuprl by defining *axiom* as the only element of their associated types. The axiom schemas have as their computational content a substitution operator and an induction operator, respectively.

Finally, we add rules for existential and universal quantification, along with their term extractions:

$$\frac{\mathtt{f} : \Gamma \vdash \varphi(a)}{\lambda\mathtt{g}.\mathtt{pair}(\mathtt{a}, \mathtt{f}(\mathtt{g})) : \Gamma \vdash \exists x.\varphi(x)}$$

$$\frac{\mathtt{f} : \Gamma \vdash \varphi}{\lambda\mathtt{g}.\lambda\mathtt{y}.\mathtt{f}(\mathtt{g}) : \Gamma \vdash \forall x.\varphi}\,(*)$$

$$\frac{\mathtt{f} : \Gamma \vdash \varphi(a), \Delta}{\lambda\mathtt{g}.\mathtt{decide}(\mathtt{f}(\mathtt{g}); \mathtt{u}.\mathtt{pair}(\mathtt{a}, \mathtt{u}); \mathtt{v}.\mathtt{inr}(\mathtt{v})) : \Gamma \vdash \exists x.\varphi(x), \Delta}$$

$$\frac{\mathtt{f} : \Gamma \vdash \varphi, \Delta}{\lambda\mathtt{g}.\mathtt{decide}(\mathtt{f}(\mathtt{g}); \mathtt{u}.\lambda\mathtt{y}.\mathtt{u}; \mathtt{v}.\mathtt{inr}(\mathtt{v})) : \Gamma \vdash \forall x.\varphi, \Delta}\,(*)$$

$$\frac{\mathtt{f} : \Gamma, \varphi(a) \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{y}.\mathtt{f}(\mathtt{g}, \mathtt{y}(\mathtt{a})) : \Gamma, \forall x.\varphi(x) \vdash \Delta}$$

$$\frac{\mathtt{f} : \Gamma, \varphi \vdash \Delta}{\lambda\mathtt{g}.\lambda\mathtt{y}.\mathtt{spread}(\mathtt{y}; \mathtt{u}, \mathtt{v}.\mathtt{f}(\mathtt{g}, \mathtt{v})) : \Gamma, \exists x.\varphi \vdash \Delta}\,(*)$$

Rules marked with $(*)$ have the extra condition that x is not free in Γ or Δ. Note that none of these rules requires a *call/cc* in its extraction. Also note that the term language remains the same as that used in the propositional case. This means that the computation rules need not be modified, even though we have added quantifiers to the types.

6.2.2 Proof of the theorem

We now prove a slightly more general statement of Friedman's theorem, which is based on the theorem of Murthy in [Mur90]. To state this theorem we need one more definition.

The class $\mathcal{P}$ of *printable types* is defined inductively by:

$$
\begin{array}{rcl}
N & \in & \mathcal{P} \\
a = b & \in & \mathcal{P} \\
P_1, P_2 \in \mathcal{P} & \Rightarrow & P_1 \wedge P_2 \in \mathcal{P} \\
P_1, P_2 \in \mathcal{P} & \Rightarrow & P_1 \vee P_2 \in \mathcal{P} \\
P_1 \in \mathcal{P} & \Rightarrow & \exists x.P_1 \in \mathcal{P}
\end{array}
$$

Note that an inhabitant t of a printable type in normal form can have no binding operators; hence, even under a reduction strategy which does not reduce under a binding operator, there can be no redexes anywhere in the normal form of t.

We may now prove:

Theorem 6.2.2 *If PA proves $\exists x.P(x)$, where $P(x)$ is a printable type, then HA proves $\exists x.P(x)$.*

Proof: Adding the axioms and rules above along with their computational content to the system of the previous chapter results in a proof system for PA. Given a proof of $\exists x.P(x)$ in this system, we may extract a classical term t (i.e. a term with *call/cc*'s) of type $\exists x.P(x)$. Using either a call-by-name or call-by-value reduction strategy, t normalizes to t'. Furthermore, since $P(x)$ is a printable type, t contains no binding operators; hence t' contains no *call/cc*'s or continuations. Thus, t normalizes to a constructive proof of $\exists x.P(x)$. $\square$

Chapter 7

Conclusions

In this thesis, we have explored three aspects of the computational content of proofs. First we studied a computational interpretation of the metatheory of intuitionistic propositional logic, then an extension of this approach to intuitionistic predicate logic, and finally a computational interpretation of classical sequent proofs.

In general, our aim has been to study programs extracted from proofs, with an emphasis on programs which are as useful and as practical as possible. To this end, in chapter 2, we examined what we might call the logical content of the tableau algorithm. We showed that the algorithm can be interpreted as a constructive proof of completeness of Kripke models for intuitionistic propositional logic. Moreover, the algorithm can also be interpreted as a formal comparison between Kripke semantics and realizability semantics. This result also provides an embedding of Kripke semantics in the more computational type theoretic semantics. The proof which results from consideration of the tableau algorithm actually contributes to our understanding of the algorithm, since certain implementation problems can be understood more clearly in the setting of the proof.

In exploring extensions of this result to intuitionistic predicate calculus, we discovered that although we cannot have a decision procedure, we can nevertheless prove a constructive theorem which has useful computational content. We also showed that classical logic can be used in certain ways without destroying the computational content of the proof.

This led to an examination of the computational aspects of classical proofs. Continuing with the tableau example, we showed how we could use a nonlocal control operator as the computational content of a classical axiom to obtain a tableau procedure with a particular tree search strategy. Using the nonlocal control operator *call/cc* as the computational interpretation of the classical axiom $((\varphi \rightarrow \perp) \rightarrow \varphi) \rightarrow \varphi$, we defined a method for extracting programs from classical sequent proofs in propositional logic. We gave a proof-theoretic account of continuations and their behaviour in this context, and proved a number of properties of the extraction.

One point to note is that although we have been studying the logical content of algorithms and control operators, we have not been studying "logics of programs" in the sense of Hoare logic. Our approach has produced proofs which are of interest in themselves, and not solely in the context of proving a property of a program. The programs extracted from proofs using the methods described here are, in general, functional programs, but the work on nonlocal control operators shows that they need not be pure functional programs.

We have shown that the consideration of proofs as programs and programs as proofs need not lead to inefficient programs or incomprehensible proofs. In section 2.4, we also saw that this technique can have practical applications to the automatic generation of programs. Work towards an implementation of the proof in chapter 2 is being done by Caldwell [Cal93], who is beginning with the simpler classical case described in [CH90]. A full implementation of the proof outlined in the last section of chapter 2 and described in detail in [ACU] would be an important test of this method for adding verified tactics to Nuprl. In future work, we hope to explore the applications of this work to program development systems.

Bibliography

[ACU] William Aitken, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. unpublished manuscript.

[Ait94] W. Aitken. *Metaprogramming in Nuprl Using Reflection.* Ph.D. dissertation, Computer Science Department, Cornell University, Ithaca, NY, 1994.

[Bac89] R.C. Backhouse. Constructive type theory–an introduction. In M. Broy, editor, *Constructive Methods in Computer Science,* NATO ASI Series, Vol. F55*: Computer & System Sciences*, pages 6–92. Springer-Verlag, New York, 1989.

[BB92] Franco Barbanera and Stefano Berardi. Continuations and simple types: a strong normalization result. manuscript, 1992.

[BC93] D. Basin and R. Constable. *Metalogical Frameworks*, chapter 1, pages 1–29. Cambridge University Press, Great Britain, 1993.

[Bee85] M.J. Beeson. *Foundations of Constructive Mathematics.* Springer-Verlag, Berlin, 1985.

[Bet56] E. W. Beth. Semantic construction of intuitionistic logic. *Koninklijke Nederlandse Akademie van Wetenschappen, Niewe Serie*, pages 357–388, 1956.

[BJ88] G.S. Boolos and R.C. Jeffrey. *Computability and Logic, Third Edition.* Cambridge University Press, 1988.

[C^+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[Cal93] J. Caldwell. A constructive proof of propositional completeness in Nuprl. implementation notes, June 1993.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CH90] R. L. Constable and D. J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers (North-Holland), 1990.

[Con71] R. L. Constable. Constructive mathematics and automatic program writers. In *Proc. IFIP Congr.*, pages 229–233. North-Holland, 1971.

[deB70] N.G. deBruijn. The mathematical language Automath, its usage and some of its extensions. In *Symp. on Automatic Demonstration,* Lecture Notes in Mathematics, Vol. 125, pages 29–61. Springer-Verlag, 1970.

[DK61] V. H. Dyson and G. Kreisel. Analysis of beth's semantic construction of intuitionistic logic. Technical Report 3, Applied Mathematics and Statistics Laboratories, Stanford University, January 1961.

[Dra88] A. G. Dragalin. *Mathematical Intuitionism*, volume 67 of *Translations of Mathematical Monographs.* American Mathematical Society, Providence, Rhode Island, 1988.

[dS76] H. de Swart. Another intuitionistic completeness proof. *Journal of Symbolic Logic*, 41(3):644–662, September 1976.

[Dum77] M. Dummett. *Elements of Intuitionism.* Oxford Logic Series. Clarendon Press, 1977.

[Dyc92] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, pages Vol.57, Number 3, September 1992.

[FFKD87] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[Fit69] Melvin Fitting. *Intuitionistic Logic, Model Theory, and Forcing.* North-Holland, 1969.

[Fit83] Melvin Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169 of *Synthese library.* D. Reidel Publishing Company, 1983.

[Fri75] H. M. Friedman. Intuitionistic completeness of Heyting's predicate calculus. *Notices of the American Mathematical Society*, 22, 1975.

[Fri77a] H. M. Friedman. The intuitionistic completeness of intuitionistic logic under tarskian semantics. unpublished abstract, State University of New York at Buffalo, 1977.

[Fri77b] H. M. Friedman. New and old results on completeness of hpc. unpublished abstract, State University of New York at Buffalo, 1977.

[Fri78] H. Friedman. Classically and intuitionistically provably recursive functions. In D.S. Scott and G.H. Muller, editors, *Higher Set Theory,* Lecture Notes in Mathematics, Vol. 669, pages 21–28. Springer-Verlag, 1978.

[Gen69] Gerhard Gentzen. Investigations into logical deduction (1934). In M. Szalo, editor, *The Collected Papers of Gerhard Gentzen.* North-Holland, Amsterdam, 1969.

[Gir91a] J-Y. Girard. A new constructive logic: classical logic. *Math. Struct. in Comp. Science*, 1:255–296, 1991.

[Gir91b] Jean-Yves Girard. A new constructive logic : classical logic. *Mathematical Structures in Computer Science*, 1:255–296, 1991.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation,* Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, NY, 1979.

[Gor85] M. Gordon. HOL: A machine oriented formalization of higher order logic. Technical Report 68, Cambridge University, 1985.

[Gri90] T. Griffin. A formulas-as-types notion of control. In *Proc. of the Seventeeth Annual Symp. on Principles of Programming Languages*, pages 47–58, 1990.

[GTL89] J. Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[Hud92] J. Hudelmaier. Bounds for cut-elimination in intuitionistic propositional logic. *Archive for Mathematical Logic*, 31:331–353, 1992.

[Hyl82] J. M. E. Hyland. The effective topos. In A. S. Troelstra and D. S. van Dalen, editors, *L. E. J. Brouwer Centenary Symposium.* North-Holland, Amsterdam, 1982.

[Kle50] S. C. Kleene. Recursive functions and intuitionistic mathematics. In *Proceedings of the International Congress of Mathematicians*, pages 679–685, Providence, RI, 1950. American Mathematical Society.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* D. Van Nostrand, Princeton, 1952.

[KMO91] Stuart A. Kurtz, John C. Mitchell, and Michael J. O'Donnell. Connecting formal semantics to constructive intuitions. In J. P. Meyers and M. J. O'Donnell, editors, *Constructivity in Computer Science*, number 613 in Lecture Notes in Computer Science, pages 1–21. Springer-Verlag, 1991.

[Kri65] S. Kripke. Semantical analysis of intuitionistic logic i. In J. N. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, 1965.

[Lau70] H. Lauchli. An abstract notion of realizability for which intuitionistic predicate calculus is complete. In *Intuitionism and Proof Theory*, pages 227–34. North-Holland, Amsterdam, 1970.

[Lip90a] J. Lipton. Realizabilty and kripke forcing. Technical Report 90-1163, Cornell University, Ithaca, NY, October 1990.

[Lip90b] James Lipton. Constructive kripke semantics and realizability. Technical Report 90-71, Mathematical Sciences Institute, Cornell University, 1990.

[Lip90c] James Lipton. *Relating Kripke Models and Realizability.* Ph.D. dissertation, Cornell University, 1990.

[Men88] P.F. Mendler. *Inductive Definition in Type Theory.* Ph.D. dissertation, Cornell University, Ithaca, NY, 1988.

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.

[Mur90] C. Murthy. *Extracting Constructive Content from Classical Proofs.* Ph.D. dissertation, Cornell University, Dept. of Computer Science, 1990. (TR 89-1151).

[Mur91a] C. Murthy. Classical proofs as programs: How, when, and why. Technical Report 91-1215, Cornell University, 1991.

[Mur91b] C. Murthy. An evaluation semantics for classical proofs. In *Proc. of Sixth Symp. on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.

[Ner89] Anil Nerode. Some lectures on intuitionistic logic. Technical Report 89-6, Mathematical Sciences Institute, Cornell University, 1989.

[NS94] A. Nerode and R. Shore. *Logic for Applications*. Springer-Verlag, New York, 1994.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: An integrated approach to specification and verification. Preprint, January, 1992.

[Par92] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. International Conference on Logic Programming and Automated Deduction,* St. Petersburg, Russia, Lecture Notes in Computer Science, Vol. 624, pages 190–201. Springer-Verlag, 1992.

[Pau88] L.C. Paulson. A formulation of the simple theory of types (for Isabelle). In P. Martin Löf and G. Mints, editors, *Proc. of the Int. Conference on Computer Logic,* Lecture Notes in Compuer Science, Vol. 417, pages 246–274. Springer-Verlag, New York, December, 1988.

[PD] Luis Pinto and Roy Dyckhoff. A loop-free construction of counter-models for intuitionistic propositional logic. draft of July 2, 1993.

[Pit] A. M. Pitts. A co-induction principle for recursively defined domains. to appear in Theoretical Computer Science.

[Plo75] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–59, 1975.

[Pol90] R. Pollack. Lego user's guide. Technical report, University of Edinburgh, 1990.

[Smu68] R. M. Smullyan. *First–Order Logic*. Springer–Verlag, New York, 1968.

[Sta79] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

[TvD88a] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: an Introduction*, volume 2. North-Holland, 1988.

[TvD88b] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction*, volume 1. North-Holland, 1988.

[Und90] J. Underwood. A constructive completeness proof for the intuitionistic propositional calculus. Technical Report 90-1179, Cornell University, 1990.

[Und93a] J. Underwood. The tableau algorithm for intuitionistic propositional calculus as a constructive completeness proof. In *Proceedings of the Workshop on Theorem Proving with Analytic Tableaux, Marseille, France*, pages 245–248, 1993. Available as Technical Report MPI-I-93-213 Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[Und93b] Judith Underwood. On the computational content of classical sequent proofs. In *Transactions of the Eleventh Army Conference on Applied Mathematics and Computing*, ARO Report 94-1, pages 41–54. Army Research Office, 1993.

[vD80] D.T. van Daalen. *The Language Theory of AUTOMATH.* Ph.D. dissertation, Technical University of Eindhoven, Eindhoven, Netherlands, 1980.

[Vel76] Wim Veldman. An intuitionistic completeness theorem for intuitionistic predicate logic. *Journal of Symbolic Logic*, 41(1):159–166, March 1976.