DECIDABILITY EXTRACTED: SYNTHESIZING "CORRECT-BY-CONSTRUCTION" DECISION PROCEDURES FROM CONSTRUCTIVE 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

James L. Caldwell II

August 1998

DECIDABILITY EXTRACTED: SYNTHESIZING "CORRECT-BY-CONSTRUCTION" DECISION PROCEDURES FROM CONSTRUCTIVE PROOFS.

James L. Caldwell II, Ph.D.

Cornell University 1998

The topic of this thesis is the extraction of efficient and readable programs from formal constructive proofs of decidability. The proof methods employed to generate the efficient code are new and result in clean and readable Nuprl extracts for two non-trivial programs. They are based on the use of Nuprl's set type and techniques for extracting efficient programs from induction principles.

The constructive formal theories required to express the decidability theorems are of independent interest. They formally circumscribe the mathematical knowledge needed to understand the derived algorithms. The formal theories express concepts that are taught at the senior college level. The decidability proofs themselves, depending on this material, are of interest and are presented in some detail.

The proof of decidability of classical propositional logic is relative to a semantics based on Kleene's strong three-valued logic. The constructive proof of intuitionistic decidability presented here is the first machine formalization of this proof. The exposition reveals aspects of the Nuprl tactic collection relevant to the creation of readable proofs; clear extracts and efficient code are illustrated in the discussion of the proofs.

BIOGRAPHICAL SKETCH

James Caldwell was born September 25, 1956.

The earliest manifestation of the impulse to create something permanent occurred about 1961 when he discovered a bag of concrete mix in the woods and, dragging it to a nearby stream, tore it open, and added water, building his first site sculpture. To his own amazement, the sculpture was still extant in 1962.

He graduated from Clarkstown High School in New City, New York in 1974 and attended the State University of New York Cortland, studying painting with Jim Thorpe. In 1976, on the recommendation of his mentors, Caldwell moved to New York City to study in the Empire State College studio art program run by Irving Kriesberg at Westbeth[1].

He met Penelope Potter in the Westbeth studio; building her a wall and then crossing the Brooklyn bridge with her on foot. Together they moved to Boston, to continue studies at the Boston Museum School of Fine Arts, living in Back Bay at first, and later behind a painting in a Congress Street loft. In 1978 they were married and moved to Second Street in NYC. Their daughter Clea Caldwell was born in NYC. Taxi driving, cabinetry work in SoHo lofts, and picture framing supported the Caldwell family.

In 1979 he was hired and trained as a draftsman on a an early computer aided drafting system, beginning his computer career. Soon after writing his first computer program, a routine to generate random drawings, Caldwell realized that the concerns of theoretical computer science and logic appealed to his aesthetic sensibilities.

[1]Ironically, during his tenure at Cornell, one of Kriesberg's paintings, on loan from the permanent collection of the Johnson Museum, hung in Upson 4160.

In 1984 he received an undergraduate degree in Computer Science from State University of New York at Albany. By 1985 he was employed at the General Electric Research and Development Center in Schenectady, New York and spent his time pondering design synthesis and formal correctness. Penelope gave birth to their second child, James Garrett, at home in Albany in 1986. Caldwell completed his masters degree in Computer Science at SUNY Albany in 1988 under Dan Rosenkrantz's tutelage.

Also in 1988, Caldwell accepted a position at NASA Langley Research Center in Hampton, Virginia, joining a newly formed theorem proving group. Supported by NASA, he attended Cornell in 1990 to pursue his Ph.D. in Computer Science. In 1993 he returned to Langley, but returned to Ithaca again in 1997 to complete his degree.

In the summer of 1998 he accepted a position as Assistant Professor in Computer Science at the University of Wyoming in Laramie.

To my parents.

ACKNOWLEDGEMENTS

Bob Constable has encouraged and supported me throughout my Cornell career and I owe him truly great thanks. He is also responsible for creating and nurturing the intellectual environment in which the Nuprl group works, a great achievement that others who have gone before me, and those who follow can best appreciate. Thank you Bob.

Fred Schneider has been a fair-minded and deeply critical mentor who has always insisted on the highest standards. I am proud to have had him serve on my committee.

Anil Nerode has always seemed to me to be the happy Buddha (although he denies any Buddhist affiliation) and I have always enjoyed his conversation and insights. I never got enough of his company. Time spent with Richard Platek, both at Cornell and at his company Odyssey Research Associates, has always been similarly enjoyable.

At Cornell, Stuart Allen has been a great friend and has taught me many of the intricacies of constructive type theory to boot. Thanks to Stuart and Tamiko for all their help and support in getting this thesis out the door. Aswin van den Berg volunteered to do the leg-work to get this thesis submitted and I thank him. Paul Jackson and Doug Howe often helped me understand details of the system which were beyond my ken. Judith Underwood and her husband Ian Gent proved to be wonderful collaborators and friends. Jon Beck offered me his friendship and, always ready to challenge accepted mathematical practice, proved to be a good sounding-board for ideas. Aside from relieving me from the dubious honor of being the most senior grad-student in the department, David Pearson was both a good friend and great climbing partner.

At NASA, my manager Chuck Meissner supported my application for graduate study leave. Ricky Butler, although a staunch believer of excluded-middle, gave me leave to pursue my work unfettered by NASA cares. Paul Miner and Victor Carreño were colleagues and friends who shared the same path for many years. Bernice Weinstein provided friendship. Special thanks go to Mike Lowry at NASA Ames who showed genuine enthusiasm for my work and spurred me on to the finish.

Finally, greatest thanks go to my wife Penelope who has supported me unconditionally. Also, thanks to my children, Garrett and Clea, who did without me more than they should have had to.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Goals

This thesis addresses two topics: the first is the problem of formally synthesizing decision procedures as a way to ensure the correctness of their claims; this involves the formalization of mathematical knowledge and the application of those formalisms in potentially large proofs. The second topic, largely motivated by the search for methods to accomplish the first, is the development of methods to make programming by synthesizing code from constructive proofs a practical endeavor.

This thesis presents type theoretic formalizations of, and constructive proofs of decidability for, classical and intuitionistic propositional logics. Doing so addresses the goals noted above quite concretely. The mathematics has been formalized in the Nuprl System, an implementation of an extremely rich constructive type theory that is supported by a mature interactive theorem proving environment.

The formal development of the decision procedures is interesting in its own right.

When we say our goal is to make program synthesis a practical endeavor we mean two things. The first is that the effort required is comparable to the effort required for other methods offering similar guarantees of correctness. If the development formally justifies the mathematical claims by formal proofs, it should represent no more effort than it might have required to use another proof system guaranteeing a similar level of assurance. The evidence for the claim that Nuprl is satisfactory on this account can be found by examining the complexity of the mathematical claims that have been proved in Nuprl and comparing their formalizations to those done in other systems. The best resource for this information is the Nuprl web-page[CT98]. In developments where the mathematical claims are not completely justified, the effort required to use of the system should compare favorably with less formal program development methods. Early evidence that Nuprl can function in this mode can be found in [BC85], more recent evidence can be found in [CGU99].

The second aspect of practicality we intend is a property of the extracted programs themselves: the synthesized programs, as artifacts, must be comprehendible in their own right. The extracted programs must be readable, they must reveal the structure of the computation they are meant to perform; it must be possible, although it is not always necessary, to synthesize programs that are as efficient as hand crafted code written by expert programmers. The programs extracted from the proofs presented in this thesis will be seen to satisfy this requirement.

1.2 Background and Related Work

This work is situated in the larger context of constructive mathematics, formal methods, and program synthesis.

1.2.1 History and Related Systems

As early as 1971, in an IFIP paper entitled *Constructive Mathematics and Automatic Program Writers* [Con71], Constable presented the idea of programming via constructive proofs. He reckoned that Kleene's realizability method for the arithmetization of μ-recursive functions could be modified in such a way that programs in a high level programming language like Algol, or instances of machines in an abstract computational model like the RAM, could serve as realizers. In that paper, Constable showed that there is an effective procedure for extracting programs from formal proofs in Intuitionistic Number theory, thereby showing that the theory is a Turing complete programming language.

In 1980 Bates implemented a system called λ-PRL based on these ideas, extracting the realizers as Lisp programs. This started the line of system development that has evolved into the current Nuprl system [Con86]. The motivation of the system is programming via proofs. Bates and Constable considered this problem again in a 1981 paper [BC85] entitled *Proofs as programs* that was about λ-PRL and its applications.

The endeavor of programming via constructive proofs touches on many topics, both theoretical and practical. These ideas have been widely pursued in many independent efforts which we mention next.

Martin Löf's intuitionistic type theories [ML73, ML82] inspired the group at Göteborg, Sweden to develop methodology for program development in type theory not unlike Nuprl [NPS90], and to implement a prover for it [MN94].

Nuprl's type theory [Con86] is an extension of Martin Löf's 1982 system. Its development was influenced by the work of de Bruijn's Automath system [deB70] and by Scott's Constructive Validity paper [Sco70]. The extensions include the quotient type [CZ84], the set type [Con83, Con85], and the inductive type [CM85, Men88]. Recently, rules for the intersection type have been added, as have rules for a newly developed partial type [CC98] which allows for reasoning about partial functions in type theory.

In France, Coquand and Huet designed the Calculus of Constructions [CH85, CH88a], a higher-order impredicative constructive type system based on Girard's system F [Gir86]. The Coq system [DFH$^+$93a, CCF$^+$95] provides a computer implementation of the Calculus of Constructions, extending it by the addition of an inductive type. Paulin-Mohring [PM89] developed methodology for programming in Coq.

At Edinburgh, Luo [Luo89, Luo94] designed an extended theory of constructions, the Extended the Calculus of Constructions. Pollack implemented a prover for that theory named LEGO [Pol90, LP92].

In Japan, Hayashi and Nakano implemented a program synthesis system called

PX [HN88] based on Feferman's theories [Fef79]. More recently, Hayashi [Hay94] has designed a new type theory for program extraction based on intersection and union types.

In Munich, Schwictenberg's group has recently implemented a first order natural deduction theorem prover for program extraction called MINLOG.

Considering the scale of the research program undertaken by this international group, it is perhaps surprising that the application of the methods has not had more impact on practice. Much of the effort has been toward building the theoretical basis for the methods, yet only now does it seem that the implementations are reaching the potential promised by the theory.

1.2.2 Related Work

The research reported on here owes much to Underwood's work on constructive completeness proofs for intuitionistic propositional logic as a means of extracting tableau decision procedures: she reported on that work in [Und93, Und94, Und95], and also, in work done jointly with Aitken and Constable [ACU]. Her formalization of intuitionistic decidability motivated the development of the classical case presented here in chapter 4. Underwood worked out the type theoretic presentation of the problem and detailed informal proofs. Chapter 5 of this thesis is a formal implementation and extension of the proof given in [ACU]. The implementation of the proofs and the mathematics supporting them provided the means to examine the extracted programs and to manipulate the statements of the theorems and their formal proofs, resulting in clean and efficient extracts.

The idea of verifying decision procedures is not new: proposals to extend theorem provers by adding formally verified decision procedures were made as early as 1977 [DS77]. Harrison provides a detailed survey of two approaches to the disciplined extension of prover capabilities in [Har95]. Actual formal verifications of decision procedures are less common. One example that has been repeated a number of times is Boyer and Moore's propositional tautology checker in the form of an `IF-THEN-ELSE` normalization procedure [BM79, Les81, Pau86b, Hed91, PMW93]. Both Shankar [Sha85] and Hayashi [HN88] verify deciders for implicational fragments of propositional logic presented in sequent forms.

With respect to the goal of extracting readable programs from constructive proofs, Paulin-Mohring and Werner's work in the Coq system is the closest to ours. In [PMW93] they report on the extraction of the Boyer and Moore tautology checker. Their development address issues related to the efficiency of the extracted program.

Recently, Weich [Wei98a] has formalized a proof of decidability for the implicational fragment of propositional intuitionistic logic in MINLOG. His work is also closely related to the proof presented in Chapter 5 here; indeed, his effort was also inspired by Underwood's formulation of constructive decidability. Weich's proof is based on the contraction-free calculus of Dyckhoff [Dyc92]. He reports [Wei98b] that the extracted program is large (about 60KB) and he is working on minimizing its size.

Another recent application of program extraction technology include Théry's [Thé98] extraction of Buchberger's algorithm for computing Gröbner bases. This work was done using the Coq system and formalizes a sophisticated piece of mathematics. The extracted program has been applied computationally.

Finally, in Nuprl, and jointly work with Gent and Underwood [CGU99], Caldwell extracted an implementation of Conflict-directed backjump search [Pro93]. This was done in a classical extension to Nuprl which thereby allowing the extracted program to contain non-local control operators, in this case the `call/cc` operator of Scheme. The extracted program was translated into Scheme and applied to the Hamiltonian circuit problem, resulting in a new solution. This search scheme has been applied to decision procedures for propositional logic.

1.3 Results

The technical accomplishments of the thesis are as follows:

Clean extracts. The programs extracted from constructive proofs of decidability are the most tangible results of the work presented here. The program extracted from the proof of intuitionistic decidability is the first formally justified implementation for the full set of propositional operators. The extracted programs are eminently readable, more so in the classical case, but in the intuitionistic case as well. The programs are efficient in that they do not perform extraneous computations related to the logical parts of their specifications, nor do they contain unreadable artifacts of the proof in their texts. These qualities will be most evident to those familiar with the state of the art in program extraction. Relative to these properties of readability and efficiency, these programs compare favorably with extracted programs appearing anywhere in the literature.

Proof Methodology. In the course of the research described here, a new methodology for using the existing Nuprl system has been developed. The objective of these new methods is the generation of clean and readable extracts while ensuring, as far as possible, that the proofs are no more difficult than they would have been had they been developed using the standard Nuprl methodology. The method is based on use of the set type combined with the application of induction tactics that result in efficient general recursion schemes. The author extracted these efficient recursion schemes from proofs of general induction principles. A style of proof based on delaying the introduction of explicit computational content until as late as possible has been used here. This approach insures that the substitution of set types for existential quantifiers will have minimal effect on the actual proofs. This approach is presented in the course of the thesis but is the particular emphasis of Chapter 3.

Formalized Mathematics and Pedagogy. This thesis represents a significant effort in the development of formal mathematical theories. The formalizations presented here have been polished by use; the false starts and bad choices inevitable in such an effort are not revealed here. An aspect of formal mathematics that is perhaps not appreciated by those who do not practice it is that the precise form of a definition, while not always mathematically significant, can have great impact on utility in applied formal mathematics. As such, the formalizations and proofs presented here represent an investment of time and energy worthy of study.

The theories supporting the classical decidability have been used to teach Logic to undergraduates at Cornell in the past and will be used again in the future. Students use the system to navigate proofs and to view definitions. It turns out that well-motivated interactive presentations of formalized mathematics are readily accepted by students. The presentation of the intuitionistic case, previously unavailable, will become source material for future courses. Not only do the formal theories contain the Nuprl definitions, but the formal proofs, as objects, are available as well and record the structure of the arguments.

Chapter 2

Nuprl

2.1 The Nuprl Type Theory

The Nuprl type theory is a sequent presentation of a constructive type theory via type assignment rules. It supports an untyped lambda-calculus as its programming language. Following Barendregt [Bar92] we can distinguish the Nuprl type theory from the perhaps more familiar Church style typed lambda-calculus by calling it a lambda calculus in the style of Curry or a lambda calculus with type assignment. In Nuprl the underlying programming language is untyped and the objective of a proof is to prove a type is inhabited, *i.e.* to show some program (term) is a member of the type. A complete presentation of the type theory can be found in the Nuprl book [Con86] (which will be referred to subsequently as "the book").

The Nuprl system, as distinguished from the type theory, implements a rich environment to support reasoning about and computing with the Nuprl type theory. The system implementing the type theory has evolved since publication of the book but (with a few extensions) the type theory presented there is faithfully implemented by the Nuprl system. Complete documentation is included in the Nuprl V4.2 distribution available on the World Wide Web [CT98].

The following sections give a brief introduction to the Nuprl computation system, the type theory and some aspects of the system.

2.1.1 The computation system

Nuprl's *terms* include the constructs of its untyped functional programming language with additional constructs for denoting types and propositions. Terms will be printed here in typewriter font: `this is a term`. Variables denoting terms will sometimes be printed in italics (t). Terms (other than variables) are either in *canonical* form, meaning no further evaluation is possible, or in *noncanonical* form. Whether a term has canonical form depends only on its outermost form (they may contain subterms of non-canonical form). The Nuprl computation system provides reduction rules for evaluation of noncanonical forms. The Nuprl evaluator is the computer implementation of these rules. A more complete description of canonical and non-canonical forms and the semantics for the computation system is provided in the book.

For terms `t` and `t'` we will write `t` $\triangleright$ `t'` to indicate that `t` (the *redex*) evaluates to `t'` (the *contractum*) under the reduction rules. In later sections we will apply an extended version of the basic computation system via the rewrite facility. For terms `t` and `t'` we will write `t`$\triangleright_R$`t'` to indicate that `t` reduces to `t'` in this system.

As usual, the notation t[t'/x] denotes the term resulting from the capture-avoiding substitution of t' for free occurrences of x in t. Similarly, the notation $t[t_1,\cdots,t_n/\mathtt{x}_1,\cdots,\mathtt{x}_n]$ denotes the simultaneous capture-avoiding substitution of each t_i for each $\mathtt{x}_i$ in t.

The reduction rules used in the decidability proofs are presented here for quick reference.

$(\lambda x.\,b)(t)$	$\triangleright$	$b[t/x]$
$\mathtt{list\_ind}([\,];b;x,y,z.\,u)$	$\triangleright$	b
$\mathtt{list\_ind}(h::t;b;x,y,z.\,u)$	$\triangleright$	$u[h,t,\mathtt{list\_ind}(t;b;x,y,z.\,u)/x,y,z]$
$\mathtt{decide}(\mathtt{inl}(t);x.\,l;y.\,r)$	$\triangleright$	$l[t/x]$
$\mathtt{decide}(\mathtt{inr}(t);x.\,l;y.\,r)$	$\triangleright$	$r[t/y]$
$\mathtt{spread}(<t_1,t_2>;x,y.\,t)$	$\triangleright$	$t[t_1,t_2/x,y]$
$\mathtt{atom\_eq}(a;b;t_1;t_2)$	$\triangleright$	t_1 if $a=b$
$\mathtt{atom\_eq}(a;b;t_1;t_2)$	$\triangleright$	t_2 if $a\neq b$
$\mathtt{rec\_ind}(a;h,z.\,d)$	$\triangleright$	$d[a,\lambda z.\mathtt{rec\_ind}(z;h,z.\,d)/h,z]$

The first rule is the ordinary beta-reduction rule. Since it is included in the computation system, and since Nuprl terms are not tagged with type-information (*a la* Church), the evaluator is an interpreter for the untyped lambda calculus (extended with the computation rules just defined). The Nuprl term evaluator implements a left-most outermost (lazy) evaluation strategy.

2.1.2 The types

A Nuprl type is a term T of the computation system with an associated transitive and symmetric relation denoted by the term x=y∈T. This relation is known as *equality on* T and respects evaluation in terms x and y (it is an equivalence relation when restricted to members of `T`). Membership in `T` is expressed by `x∈T` which is defined as `x=x∈T`.

A formula `x∈T` is well formed (is a meaningful proposition) only when `T` is a type and `x` and `y` are both elements of type `T`; if `T` is not a type, or either `x` or `y` is not an element of `T` (or neither is), then the term `x=y∈T` denotes nothing, it is nonsense. Thus, `x∈T` is assigned a meaning when `x` is a member of `T`. Type membership is different from set membership in that `x∈T` cannot be false in Nuprl; it is either true or meaningless.

In addition to the type membership equality provided with each type, there is an equality on types. Equality of types is intensional, *i.e.* type equality in Nuprl is a structural equality modulo the direct computation rules. This means that, unlike sets which enjoy extensional equality, there can be types `T` and `T'` such that `x∈T, y∈T, x∈T',y∈T'` and `x=y∈T` but not `x=y∈T`.

Interpreting the type membership equality relation and type membership as types is made sensible via the propositions-as-types interpretation [Con86, pg.29–31].

`x=y∈T` is an *equality* term. It denotes a type when `T` is a type and `x∈T` and `y∈T`; otherwise it denotes nothing, it is nonsense. If `x` and `y` are not equal elements in `T` then the type is empty. If `x` and `y` denote equal elements in `T`, the type in inhabited by the single element denoted by the constant term `Axiom` (even when the equality is not axiomatic).

`x∈T` is a *membership* term. It is an encoding for the equality term `x=x∈T`. It denotes nothing if `T` is not a type or if `x` is not in `T`, and is inhabited by the single term `Axiom` if `T` is a type and `x` is in `T`.

Like the related type theory of Marin Löf [ML73] or the type theory of Whitehead and Russell's *Principia Mathematica*, the Nuprl's type theory is a predicative type theory supporting an unbounded cumulative hierarchy of type universes. Every universe is itself a type and every type is an element of some universe.

$\mathbb{U}\{i\}$ denotes the type *universe* where i is a universe level expression.[1] The members of the universe $\mathbb{U}\{\mathtt{i}\}$ are types and other universes $\mathbb{U}\{\mathtt{j}\}$ for `j<i`. The property of `T` being a type is approximated by $\mathtt{T}\in\mathbb{U}i$. When the level expression parameter is "i" it is elided by the standard display forms and $\mathbb{U}i$ appears simply as $\mathbb{U}$.

$\mathbb{P}\{\mathtt{i}\}$ is a synonym for $\mathbb{U}\{\mathtt{i}\}$ and is sometimes used to emphasize the propositional side of the propositions-as-types interpretation. In Nuprl no formal distinction is made between propositions and types.

The other Nuprl types and their members include the following:

`Void` is the *empty* type of which there are no members. Given a declaration `x:Void` (absurdly declaring the existence of an element of the empty type) anything follows. The constant term `any` denotes uses of elements of `Void` in extracts,

[1]Level expressions are not documented in the book. Based on Allen's design [All87a], universe level expressions provide a means of polymorphically referring to universe levels without specifying explicitly which level is intended. Jackson describes the implementation in Nuprl V4 [Jac95b, pg.23].

this is an exception condition. The term `any` is such that for all types `T` (including `Void`), `any(x)`$\in$`T`.

$\mathbb{Z}$ is the type *integer* whose members are denoted by the numerals

$\cdots$`,`$-$`1,0,1,2,`$\cdots$.

`Atom` is the type whose elements are *strings* of the form `‘‘`$\cdots$`’’` where $\cdots$ is any character string. Atoms are equal when they are the same character string. Atoms really are atomic, as there is no way in the logic to analyze an atom into characters.

`T list` is the type of *lists* of elements of type `T`. The elements of `T list` include the empty list, denoted `[]`, and conses of the form `a::t` where `a`$\in$`T` and `t`$\in$`T list`. Lists are equal either when they are both the empty list or when they have equal heads and their tails are equal.

`y:A`$\longrightarrow$`B[y]` is the *dependent function* type, not uncommonly called the *pi type* and denoted by $\mathbf{\Pi}$`y:A.B[y]`. The members of this type are functions `f` with domain of type `A` such that `f(y)`$\in$`B[y]` where `y` is a variable possibly occurring free in `B`. A lambda abstraction of the form $\boldsymbol{\lambda}$`x.M` is an element of the type `y:A`$\longrightarrow$`B[y]` if `M[a/x]`$\in$`B[a/y]` for `a`$\in$`A`. These are the functions whose range may depend on the element of the domain applied to. Function equality is extensional.

`A`$\longrightarrow$`B` is the *function* type which is an encoding of terms of the form `y:A`$\longrightarrow$`B` when `y` does not occur free in `B`.

`x:A×B[x]` is the *dependent product* type consisting of pairs `<a,b>` where `a∈A` and `b∈B[a/x]`. This type is also sometimes called the *sigma type* and is denoted `Σx:A.B[x]`. Two pairs `<a,b>` and `<a',b'>` are equal in `x:A×B[x]` when `a=a'∈A` and `b=b'∈B[a/x]`.

`A×B` is the *product* type and is an encoding of terms `x:A×B` where `x` does not occur free in `B`.

`A | B` denotes the *disjoint union* of types `A` and `B`, *i.e.*elements of this type are tagged elements of the form `inl(a)` for `a∈A` and `inr(b)` for `b∈B`. Two elements of the disjoint union are equal when their tagged elements are equal in the underlying type `A` (if the tag is `inl`) or `B` (if the tag is `inr`).

`rec(x.T)` is the Nuprl *inductive type* constructor where `x` is a variable bound in term `T`, and free occurrences of `x` in `T` denote subtypes of the type; thus, its members are the members of `T[rec(x.T)/x]`. There are some technical constraints on the form of `T` but we do not include them here. Whenever `rec(x.T)` is a type, members `a` and `b` are equal if `a=b∈T[rec(x.T)/x]`.

`{y∈T|P[y]}` denotes a *set type* when `T` is a type and `P[y]` is a proposition possibly containing free occurrences of the variable `y`. Elements `x` of this type are those elements of `T` such that `P[x/y]` is true. Equality for set types is just the equality of `T` restricted to `{y∈T|P[y]}`. This type is related to the dependent product type.

`x,y:A//E[x,y]` denotes a *quotient* which is a type whenever `A` is a type, and

E[x,y] is an equivalence on A. Its members are elements of A and it identifies elements a and b whenever the equivalence E[a,b/x,y] holds.

∩x:T.P[x] denotes the *intersection* type. It is a type whenever T is a type and P[z/x] can be shown to be a type under the condition that z is a variable of type T. Two members a and b are equal in type ∩x:T.P[x] if T is a type and a=b∈P[z/x] for every z∈T.

2.1.3 Judgements, Sequents, Rules, and Proofs

Nuprl judgements are the assertions one proves in the system. Nuprl judgements take the following form:

$$\mathtt{x_1{:}T_1,\cdots,x_n{:}T_n >> S}$$

where $\mathtt{x_1,\cdots,x_n}$ are distinct variables and $\mathtt{T_1,\cdots,T_n}$, S, and s are terms (n may be 0), every free variable of $\mathtt{T}_i$ is one of $\mathtt{x_1,\cdots,x}_{i-1}$ and every free variable of S or of s is one of $\mathtt{x_1,\cdots,x_n}$. The list $\mathtt{x_1{:}T_1,\cdots,x_n{:}T_n}$ is called the *hypothesis list*, each $\mathtt{x}_i\mathtt{:T}_i$ a declaration (of $\mathtt{x}_i$), each $\mathtt{T}_i$ is a *hypothesis*, and S is the *consequent* or *conclusion*. The syntactic form is called a *sequent*.

A judgement where S is of the form t∈T is called a *well-formedness goal*.

The conditions under which a Nuprl sequent is deemed true are rather technical because of the so-called functionality constraints insuring equal elements of hypotheses can be freely substituted into the consequent and extract terms; the reader is referred to the Nuprl book [Con86, pg.141]; for a full account of Nuprl semantics the reader is directed to [All87b]. Informally, a judgement asserts that, assuming the hypotheses are well-formed types, and the conclusion and extract terms are functional in those types, then the term `S` is an inhabited type. If `S` is inhabited there may be more than one inhabitant and different proofs may yield different inhabitants.

Nuprl proofs are constructed by refinement, *i.e.* in a top-down manner. A sequent is proved by applying a *refinement rule* which induces a set of subgoals. These subgoals are, in turn, proved by refinement. If a refinement induces no subgoals, the truth of the goal is axiomatic and is justified by the rule.

Nuprl's rules are schemes for inference consisting of three parts: a Nuprl sequent called the *goal*, which is paired with an term of the computation system called the *extract*; a rule name and parameter list pair; and a collection of sequents called the *subgoals* (or `premises`), each of which is paired with a variable denoting the extract of the subgoal. The form of an introduction rule is:

$$
\begin{array}{l}
\mathtt{H} \vdash \mathtt{T\ ext\ t} \\
\quad \mathtt{BY}\ \mathit{rule\_name}\ \mathit{rule\_parms} \\
\qquad \mathtt{H}_1 \vdash \mathtt{T}_1\ \mathtt{ext}\ \mathtt{t}_1 \\
\qquad \vdots \\
\qquad \mathtt{H}_k \vdash \mathtt{T}_k\ \mathtt{ext}\ \mathtt{t}_k
\end{array}
$$

Here, $\mathtt{H}_1,\cdots,\mathtt{H}_k$, and `H` are meta-variables denoting all or part of a hypothesis list of a Nuprl sequent. For the elimination rules, hypotheses of the goal and subgoal sequents are of the form `H',z:Z, H''`. Parameters required to completely instantiate a rule (new variables, universe levels, etc.) are specified in the list *rule_parms*.

Rules serve both to specify proof refinement steps, to be applied in a top-down fashion, as well as a scheme for constructing extract terms from the subgoal extracts $t_1,\cdots,t_k$. The fact that the extract `t` inhabits `T` is an artifact of the proof of `T`. The computational content of well-formedness subgoals or equality subgoals is the trivial term `Axiom` and so is not displayed.

For example, consider the introduction and elimination rules for dependent function type that appear as follows in the Nuprl theory `rules_1` in the standard library.

```
*R lambdaFormation
  H ⊢ x:A ⟶ B ext λz.b
    BY lambdaFormation level{i} z
      H, z:A ⊢ B[z/x] ext b
      H ⊢ A = A ∈ 𝕌
*R dependent_functionElimination
  H, f:x:A ⟶ B, J ⊢ T ext t[(f a),Ax/y,v]
    BY dependent_functionElimination #$i a y v
      H, f:x:A ⟶ B, J ⊢ a = a ∈ A
      H, f:x:A ⟶ B, J, y:B[a/x], v:y = f a ∈ B[a/x] ⊢ T ext t
```

A Nuprl proof is a tree structure in which the root is a Nuprl judgement having no hypotheses. The children of each node are instances of sequents justified by some refinement rule applied to the node. A proof of a sequent shows that the goal, viewed as a type, is both well-formed and inhabited. Given the extract terms inhabiting the subgoals of a rule, a proof specifies how to construct an extract term inhabiting the type in the conclusion of the rule; thus, proofs contain instructions for the construction of witnesses. *Extraction* is the process of constructing a witness term as specified by a proof. Although extracts are not displayed by the system, the extract term can be retrieved by applying the function `extract_of_thm_object` to a token containing the name of the theorem to be extracted. The extract of a completed proof of a sequent is a closed term; the extract of an incomplete proof is a term possibly containing free variables.

2.2 The Nuprl system

The Nuprl system supports construction of top-down proofs by refinement. The prover is implemented as a tactic based prover in the style of LCF [GMW79] and built on a base of ML. In Nuprl and related constructive systems [ML82, Nor81, CH88b], the so-called proposition-as-types interpretation allows for presentations to be cloaked in either logical or more purely type-theoretic terms. Paul Jackson's rational reconstruction of the Nuprl V3 tactics and display forms serves as the basis of the Nuprl V4 system; it gives the system a distinctly logical appearance (in contrast to the more type-theoretic appearance of other systems).

2.2.1 The Library

The system supports a library mechanism which provides for grouping of Nuprl objects. The status and class of an object are indicated in the library by a two character sequence preceding the name of the entry in the library, the first character indicating the status and the second indicating the type of object. The six main kinds of objects (and their single character labels) are the following: an *abstraction* object (`A`) which defines an operator; a *comment* object (`C`) which may contain any untyped data, a *display form* object (`D`), an *ml* object (`M`), a *rule* object (`R`), and *theorem* objects, containing possibly incomplete proofs, and which are labeled by the lower case character (`t`) if the theorem is unexpanded and are labeled by the upper case character (`T`) if expanded. Every object has associated with it a status, also labeled by a single character, which is either *raw* (`?`), *bad* (`-`), *incomplete* (`#`), or *complete* (`*`). A *raw* status means an object has been changed but not yet checked. A *bad* status means an object has been checked and found to contain errors. An *incomplete* status is meaningful only for theorem objects and signifies that its proof contains no errors but has not been finished. A *complete* status indicates that the object is correct and complete.

Typically, each new operator is defined by three (or four) library objects: a display form object, an abstraction or definition object, a well-formedness theorem object describing the type of the operator, and possibly an ML object containing tactics and or conversions characterizing the behavior of the operator.

2.2.2 Display Forms

Nuprl supports a unique display mechanism. The structure of Nuprl terms is specified independently of their display characteristics. Terms are edited in the Nuprl structure editor, and since they are never parsed, there are no externally generated constraints based on considerations of grammar. The display form mechanism includes a rich language for display specification. Since actual term structure is independent of display it is possible to over-load displays without introducing the attending complexity of disambiguation. Indeed, mathematics can be displayed in Nuprl with the same conventions used in traditional presentations of mathematics. The Nuprl terms presented here appear as they do in the system and have been generated by the system. One significant advantage of the separation of display from definition is that individual users can customize displays to their own tastes.

For example, the following shows two different Nuprl displays of the identical term.

```
(x:ℤ)(y:ℤ)(∃z:ℤ)(x+z = y)
∀x,y:ℤ. ∃z:ℤ. x+z = y
```

2.2.3 Definitions

Definitions are added to Nuprl by creating so-called abstraction objects in the Nuprl library.

One important set of definitions is the encoding of constructive logic in the type theory. The Heyting interpretation is encoded in the Nuprl type theory by the following abstractions which are defined in the Nuprl V4 `core_1` system library.

`*A true`	`True`	$\overset{\text{def}}{=}$	`0 ∈ ℤ`
`*A false`	`False`	$\overset{\text{def}}{=}$	`Void`
`*A and`	`P ∧ Q`	$\overset{\text{def}}{=}$	`P × Q`
`*A or`	`P ∨ Q`	$\overset{\text{def}}{=}$	`P | Q`
`*A implies`	`P ⇒ Q`	$\overset{\text{def}}{=}$	`P ⟶ Q`
`*A not`	`¬A`	$\overset{\text{def}}{=}$	`A ⇒ False`
`*A exists`	`∃x:A. B[x]`	$\overset{\text{def}}{=}$	`x:A × B[x]`
`*A all`	`∀x:A. B[x]`	$\overset{\text{def}}{=}$	`x:A ⟶ B[x]`

Thus, `True`, which could be modeled by any inhabited type, is modeled here as the type asserting `0`$\in$ $\mathbb{Z}$ (which happens to be true and has the single inhabitant `Axiom`). `False` is encoded as the empty type; conjunction is just a simple product; disjunction is disjoint union; implication is the function type; negation of a proposition `P` is defined to be the function that, when applied to an element of `P`, returns an element of the empty type; existential quantification is encoded as the dependent product type; and universal quantification is the dependent function type. The correspondence between the propositions encoded as above and the type theory is elaborated on in [Con86, Section 3.6,]. The Nuprl tactics have been built to deal uniformly with either the propositional or type formulations.

Chapter 3

Programming in Nuprl

In this chapter we describe attributes of Nuprl that allow it to be used both as a program synthesis tool and as a program verification tool.

The essence of constructive methods for program development is that the proofs of theorems having the shape $\forall\exists$ contain all the information necessary to build the object claimed to exist. This idea is rooted in Kleene's realizability semantics for intuitionistic number theory [Kle52]. Other implementations of Martin-Löf type theory [Nor93, Mag95] provide similar extraction methods as do Coq [DFH$^+$93b] and Hayashi's system PX [HN88] and its descendent [Hay94].

If the exact form of the intended program is known, then the corresponding proof obligation is to prove that it inhabits the appropriate type – this is program verification. In other cases, the system is used to synthesize a program from the proof. Of course, proofs are often hybrid in the sense that some parts of the extract are purely synthesized while others parts are explicitly defined and verified.

3.1 Proofs and Extracts

In this section we give definitions to distinguish proofs based on how the computational content of the proof arises. We define three classes of proofs, *strictly pure proofs*, *pure proofs*, and *computationally explicit proofs*. We introduce these distinctions here to capture the informal idea that some proofs have some or all of the computational content explicitly provided while other proofs use the constructive proof rules in a pure way.

Nuprl proofs are stored compactly as tactic trees; however, their expanded proofs form trees of rule instances and their subgoals. We will use variables p and p' to refer to expanded proof trees. Two Nuprl proofs p and p' are *equal* ($p = p'$) if and only if their expanded rule trees are identical. The extract term of a proof p is denoted $ext(p)$. The raw extract of a proof (even one designed to result in a clean extract) usually contains many applications that disappear under one bottom-up traversal of the term performing all β-reductions possible. As mentioned above in Chapter 2, the reduction system can be extended to arbitrarily include other steps of direct computations. Two terms t_1 and t_2 are *equivalent modulo a reduction system* R ($t_1 =_R t_2$) if and only if for some term t, $t_1 \rhd_R t$ and $t_2 \rhd_R t$. From now on, when we say two terms are equivalent we will mean equivalent modulo (some unspecified) reduction system R unless otherwise noted.

Each valid proposition has many distinct proofs; distinct proofs sometimes share identical extracts while in other cases proofs are distinguished not only by their rule trees but also by their extracts. Thus, extract terms can be seen to define equivalence classes of proofs.

Proofs p and p' are *strongly computationally equivalent* ($p =_{ext} p'$) if and only if the term $ext(p)$ is alpha congruent (see [Bar81]) to the term $ext(p')$. Thus, strong computational equivalence is an equivalence defined on the syntactic structure of the extract term, modulo variable renaming. This equivalence is the strongest extract equivalence on proofs.

We can further identify proofs by the extensional behavior of their extract terms. Proofs p and p' are *computationally equivalent* if in every context $\mathcal{C}[\ ]$, $\mathcal{C}[ext(p)]$ and $\mathcal{C}[ext(p')]$ normalize to alpha congruent terms.

If every proof of a given valid proposition makes use of some occurrence of a subterm of the proposition, then we will say that occurrence is *essential*.

The strictly positive parts (s.p.p) of a formula P are defined inductively as follows:

i. P is a s.p.p. of P;
ii. If $A \wedge B$ or $A \vee B$ are s.p.p of P then so are A and B;
iii. If $A \rightarrow B$ is a s.p.p. of P then so is B;
iv. If $\forall x\!:\!A.B$ or $\cap x\!:\!A.B$ are a s.p.p. of P then so is $B[t/x]$ for all terms t;
v. If $\exists x\!:\!A.B$ or $\{x\!:\, A|B\}$ are s.p.p. of P then so are A and $B[t/x]$ for all t;

In sequent calculus derivations having P at the root, the elimination rules apply to negative parts of P and introduction rules apply to the positive occurrences in P. Thus, strictly positive occurrences of subformulas can only be manipulated in a proof by an intro rule (*i.e.* a rule that operates on the right side of the sequent).

A *Harrop formula* [Har60] is a formula that does not contain a strictly positive part with $\vee$ or $\exists$ as its principal operator. We extend this standard definition to further exclude formulas having a strictly positive part with the set type constructor as its principal operator.

A *strictly pure proof* is a proof of a proposition (say `P`) whose computational content is never explicitly provided to the prover, but instead, it is implicit in its structure of rule applications. This class is restricted to those proofs that never apply the `explicit intro` rule or any existential introduction rule.

The class of strictly pure proofs is too narrow: it excludes any proof of a proposition having essential s.p.p. occurrences of an existential operator. We relax the condition by distinguishing where in the proof the existential introduction rules occur. What we wish to exclude is those proofs where an existential introduction rule occurs too early and thereby explicitly introduces computational content that might instead have been generated by applications of the ordinary proof rules.

Now consider the proofs containing no instances of the `explicit intro`rule, but possibly containing existential introduction rules. Among them there are proofs in which the occurrences of the existential intro rules occur as low (near the leaves) in the rule tree as possible. If $t_1, t_2, \cdots, t_n$ are the witness terms to these rules (as they occur in some fixed traversal order of the tree), then all proofs in the strong computational equivalence class that share the same witness set $t_1, t_2, \cdots, t_n$ will be called a *pure proof.*

A *computationally explicit proof* is one that is neither strictly pure nor pure. It is one in which computational content is provided, before it is necessary. Typically this class arises by application of an `explicit intro` rule or by certain applications of existential introduction rules.

Consider the trivial case first, *i.e.* the case where the computational content is provided by the `explicit intro` rule. Then, a Nuprl term (say `t`) is provided as the computational content of a judgement of the form $\Gamma \vdash$ `P`. This results in a subgoal of the form $\Gamma \vdash$ `t∈P`. Viewing the proposition `P` as a specification, a proof of this subgoal is a verification that `t` satisfies the specification `P`. Because it is a membership goal (its outermost operator is the `member` operator) this is a well-formedness goal. Computationally explicit proofs do not include all proofs of theorems having essential positive occurrences of existential operators.

If instead, the witness to the existential introduction rule introduces computational content that could have been otherwise constructed, we say the proof is *computationally explicit.*

The distinction we are intending is a qualitative difference not easily captured in a formal definition. The definition we have given here for a pure proof is fixed by the form of the proposition. This leaves open the possibility that a reformulated proposition having the same logical content might subvert the intended meaning of the definition.

Indeed Thompson [Tho91] recommends reformulating $\forall\exists$ theorems into theorems of the form $\exists\forall$ by Skolemizing, via the constructive axiom of choice, apparently believing it is better to verify a program than it is to synthesize it. This was an attempt to avoid the use of the set type, which makes no sense in the intensional version of Martin-Löf type theory that Thompson adopted for his book. Thus a theorem $\forall x : T.\exists y T'.P[x, y]$ becomes $\exists f : T \rightarrow T'.\forall x : T.P[x, f(x)]$. Under this scheme, the first element of the pair inhabiting the proof of the second theorem is the explicit computational content of the proof of the first. It should be clear that a proof of the Skolemized form (pure or otherwise) is not pure in the sense we intend here. The second specification is less natural and this form is never applied in this thesis. Further refinement of our characterization to rule out this type of reformulated specification is beyond the scope of this thesis.

This thesis is largely based on the idea that program synthesis is generally preferred over program verification, but that the integration of the two is necessary for practical program development. When possible, we believe pure proofs are preferred. By avoiding explicit introduction of terms of the programming language, pure proofs are at a higher level of abstraction than computationally explicit proofs. Pure proofs are more independent of the underlying programming language.

A Nuprl proof shows that a term is both inhabited (*i.e.* **true**) **and** that it is well-formed (*i.e.* that it denotes some type). Because of this, virtually all Nuprl proof rules generate well-formedness subgoals. The vast majority of the well-formedness subgoals that arise in ordinary Nuprl proofs are discharged automatically by the auto-tactic `Auto`.

The well-formedness subgoal that arises in computationally explicit proof often takes on a decidedly different character (from the point of view of the Nuprl user) from pure proofs. Some part of this difference can be attributed to the fact that verification is not the standard mode of Nuprl use; better tactic support would undoubtedly help to alleviate the problem. But there is an intrinsic difference between proofs of well-formedness goals and proofs of ordinary propositions, resulting from the structure of the Nuprl type theory itself.

3.2 Existential Types

Nuprl's existential types include the dependent product type (also widely referred to as the sigma type in the literature) and the set type; they are classified as existential types because applications of the introduction rules for these types require witness terms. Recall that the existential quantifier of Nuprl's logic is merely an abbreviation for dependent product so we will not distinguish existential propositions from product types in the discussion that follows, simply referring to the two of them as the existential type.

Methods of generating efficient and readable extracts by the use of the set type (as opposed to the existential) were presented by the author in [Cal97]. We reiterate the main points here.

Inhabitants of the existential `∃x:T.P[x]` are pairs `<a,b>` where `a∈T` and where `b∈P[a/x]`. `b` is a term inhabiting `P[a/x]` and specifies, as far as the proofs-as-programs interpretation goes, how to prove `P[a/x]`. When an existential type of the form above occurs as a hypothesis it can be decomposed into two hypotheses, one of the form `a:T` and another asserting `b:P[a/x]`. If `v` is the name of the variable denoting the existential hypothesis, occurrences of `a` in the final extract will appear as π_1`(v)`, and occurrences of `b` appear as π_2`(v)`.

Alternatively, consider the Nuprl set type `{y∈T|P[y]}`. Its inhabitants are elements of `T`, say `a`, such that `P[a/y]` holds. Thus, a set type does not carry the computational content associated with the logical part `P[a/y]`. Since the computational content is not available, the fact that the `a` has the property `P[a/x]` is not freely available in parts of a proof where it might find its way into an extract. When a set type of this form, occurring as a hypothesis, is decomposed, it results in two new hypotheses: one of the form `a:T`; and the other, a "hidden" hypothesis, of the form `b:P[a/x]`. Hidden hypotheses are discussed in more detail below; however, we briefly remark here that they are hypotheses having no computational content, and so they may not appear in an extract. If `v` is the name of the variable denoting the set type hypothesis, occurrences of `a` in the final extract will appear as `v` and, in general, occurrences of `b` may not appear unless computational content is explicitly provided for it.

The Nuprl system manages hidden hypotheses by "unhiding" them when appropriate and by preventing their inadvertent use. Hidden hypotheses are freely available in the parts of a proof that do not contribute to computational content; these parts include proofs of well-formedness (membership) subgoals, equality subgoals (to see why this membership and equality reasoning do not contribute to the computational content, recall that the only inhabitant for these judgements is the term `Axiom`) when the computational content on a branch of the proof has already been fully determined, or when the conclusion is decidable, stable, or squash stable. Hidden hypotheses may be "unhidden" when their computational content can be effectively decided, typically when they themselves can be shown to be decidable, stable, or squash stable.

3.3 Decidability, Stability, the Squash Type, and Squash Stability

Being constructive, not all propositions are assumed to be decidable in Nuprl, *i.e.* for arbitrary propositions `P`, `P ∨¬P` is not a theorem of Nuprl [Smi89]. Even though decidability for an arbitrary proposition P is not assumed, for many `P` it is uniformly decidable (*i.e.* there is an algorithm to decide) which of `P` or `¬P` holds. A constructive proof of `P ∨¬P` yields a decision procedure. Thus we define decidability by the following definition.

```
*A decidable     Dec{P}  =def  P ∨¬P
*T decidable_wf  ∀P:ℙ{i}. (Dec{P} ∈ ℙ{i})
```

Note that the well-formedness theorem asserts that for all propositions `P`, the term `Dec{P}` is itself a proposition, but it does not prove it is inhabited for arbitrary `P`. For definitions given below we will not show unexceptional well-formedness theorems but the reader should assume they have been stated and formally proved.

A related notion is that of *stability.* Stability is constructively weaker than decidability and like decidability is not constructively valid.

```
*A stable      Stable{P}  =def  (¬¬P) ⇒ P
```

Thus, if evidence for the fact that a proof of `¬P` is absurd can be shown to yield evidence for `P`, then we say `P` stable. Since $\neg(\neg\neg P \wedge \neg P)$ is intuitionistically valid, from $\neg\neg P$ we can conclude that $\neg P$ is not valid. Thus, inhabitants of doubly negated propositions can be thought of as evidence that the proposition is not falsifiable. We will say that proofs of doubly negated propositions P provide evidence for the *weak truth* of P since the proofs do not contain computational content for inhabitants of P itself but instead show it cannot be falsified.

For a proposition to be stable, the weak truth of the proposition must contain enough evidence to construct a proper inhabitant of the proposition. Stability is closely related to Gödel's double-negation translation [Göd65] which embeds classical logic into intuitionistic logic via double negations propagated throughout the structure of a formula.

It is an interesting exercise to try to prove `Stable{P} ⟶Dec{P}` to discover how the proof fails.

A squashed type (or proposition) is one whose computational content has been discarded. It is defined in Nuprl using the set type as follows:

`*A squash` $\quad$ `↓T` $\overset{\text{def}}{=}$ `{True| T}`

Thus for a type (proposition) `T`, `↓T` is inhabited if and only if `T` is, and furthermore, has as its only inhabitant the term `Axiom` (the sole inhabitant of the proposition `True`). This "squashes" the representatives of members of the type down to the single term `Axiom`.

Squash stability is weaker even than stability, though classically the two notions are equivalent.

`*A sq_stable` $\quad$ `SqStable{P}` $\overset{\text{def}}{=}$ `↓{P} ⟶ P`

`False` is squash stable, (it follows from the decidability of `False`). All provably inhabited propositions are squash stable as well: to see this, note that if `p∈P` then `(λx.p) ∈↓P`.

Nuprl theorems ordering these notions by their implicational strength are given below.

```
*T stable__from_decidable   ∀P:ℙ. Dec(P) ⇒ Stable{P}
*T sq_stable_from_decidable  ∀P:ℙ. Dec(P) ⇒ SqStable(P)
```

None of these implications hold in the opposite direction. Since `↓P ⇒ ¬¬P` we also have the following theorem.

```
*T sq_stable__from_stable    ∀P:ℙ. Stable{P} ⇒ SqStable(P)
```

Following Troelstra [Tro73], Salvesen and Smith [SS87, SS88] show that the natural type-theoretic analog of the Harrop formulas [Har60] preserve stability in extensional Martin-Löf type theory.

The analog of this observation is extended and implemented in Nuprl for *squash stability*. In Nuprl, squash stability is the weakest condition on a proposition `P` which allows occurrences of it as a hidden hypothesis in a sequent to be unhidden. That is, an application of the Unhide tactic to a sequent containing a hidden hypothesis `P` yields a subgoal of the same form except where `P` is no longer hidden. This is also true if `P` is stable or decidable.

The mechanism for automatically proving squash stability goals in Nuprl (*i.e.* sequents having conclusions of the form `SqStable{G}`) was implemented by Jackson [Jac95b] in the `ProveSqStable` tactic. This tactic backchains through hypotheses in the sequent declaring an operator to be squash stable, and through lemmas in the library having names with the prefix `sq_stable__`*op_id_name*. Thus, the tactic uses the ordering lemmas shown above, an extensible set of characterization lemmas, and also relevant hypotheses occurring in the sequent being proved.

The following lemmas characterize the squash stable analogue of the Harrop formulas.

```
*T decidable__false    Dec(False)
*T decidable__true     Dec(True)
*T sq_stable__equal   ∀A:𝕌. ∀x,y:A.  SqStable(x = y)
*T sq_stable__and     ∀P,Q:ℙ.  SqStable(P) ⇒ SqStable(Q) ⇒
                                 SqStable(P ∧ Q)
*T sq_stable__implies ∀P,Q:ℙ.  SqStable(Q) ⇒ SqStable(P ⇒ Q)
*T sq_stable__all     ∀A:𝕌. ∀P:A⟶ℙ. (∀x:A. SqStable(P[x])) ⇒
                                 SqStable(∀x:A. P[x])
```

Squashed terms are trivially squash stable. Also, since intuitionistic negation of a proposition `P` is simply defined to be `P⇒False`, and since bi-implication is defined to be the conjunction of implication in both directions, the following lemmas are added as well.

```
*T sq_stable__squash  ∀P:ℙ. SqStable(↓P)
*T sq_stable__not     ∀P:ℙ. SqStable(¬P)
*T sq_stable__iff     ∀P,Q:ℙ.  SqStable(P) ⇒ SqStable(Q) ⇒
                                 SqStable(P ⟺ Q)
```

To prune the search space, the `ProveSqStable` tactic does not unfold definitions. Thus, the standard methodology is to add squash stability lemmas for new operators if possible.

3.3.1 A discussion of the set type

The set type was proposed by Constable [CZ84] as a means to define sub-types in analog with a constrained set comprehension principle (As we shall see, this analogy with set theory, reflected in the standard notation for the type, may have been unfortunate). The set type was seen to eliminate the second element of the pair inhabiting a dependent product, and indeed Constable proposed it be used to generate "clean extracts". As discussed above, this second component of the dependent product is often computationally irrelevant.

The set type was subsequently adopted by the Göteburg group [NPS90] in their implementation of Martin-Löf type theory as a programming logic.

Early on, Salvesen [SS87, Sal89] explored the use of the set type in specifications and its effect on programs extracted from proofs. Indeed, she was perhaps the first to apply the set-type in the early implementation of Nuprl. In section 2 of [Sal89] she reports on a Nuprl proof of the following theorem.

```
∀x:ℕ List. ∀n:{ z: ℕ | z ∈ x}. n ∈ x
```

Thompson [Tho92] makes much of the fact that Salvesen's proof was non-trivial, which undoubtedly it was in 1989. In the current version of Nuprl, which includes tactic support for proving squash stability, the proof of this proposition is quite straightforward. Aside from proving the squash stability of the list membership operator, the proof of this theorem is absolutely trivial. In the current system the proof that list membership is squash stable is quite easy. Thus, it appears that one of the main criticisms, that proofs of theorems containing set types are non-trivial, no longer holds. Improved technology has apparently solved it.

An apparently deeper criticism appears in Thompson [Tho92] where he cites Salvesen and Smith's work [SS87, SS88] showing that in Martin-Löf's intensional type theory, the set type is absurd; this has no bearing on Nuprl, which is based on the extensional Martin-Löf theory, but it does seem to indicate an essential weakness in the intensional theory since they show unequivocally that it cannot be extended to reasonably accommodate a subset type.

3.4 General Recursive Definitions in Nuprl

Nuprl is unique among existing implementations of constructive systems in its admissibility of general recursive definitions. This possibility for the Nuprl type system was first noted by Allen, who realized that applications of Y could be assigned a type. In a Nuprl seminar in 1984 he presented a typing of Kleene's minimization operator (μ) using his methods. Based on Allen's proof, Howe [How88, How93] developed the standard Nuprl methodology for using general recursive functions and Jackson [Jac95a, Jac95b] implemented the methodology in his tactics for Nuprl 4.

Curry's fixedpoint combinator is defined in the Nuprl system library `core_2` as follows:

$$\texttt{*A ycomb} \qquad \texttt{Y} \overset{\text{def}}{=} \boldsymbol{\lambda}\texttt{f.(}\boldsymbol{\lambda}\texttt{x.f (x x))(}\boldsymbol{\lambda}\texttt{x.f (x x))}$$

Its usefulness in defining general recursive functions is based on its fixed point property.

$$\texttt{Y F} \;\rhd_R\; \texttt{F (Y F)}$$

Here, `F` is any term.

Allen's typing of applications of Y depended on the direct computation rules of the system; indeed, he presented his proof as part of his argument for their adoption. By definition, the direct computation rules observe subject reduction (*i.e.* they preserve typing). The fixed-point property is easily justified purely in terms of direct computation rules and, since they preserve typing, well-formedness goals are never generated by unfoldings of Y.

Jackson's [Jac95a, Jac95b] implementation of these ideas in Nuprl V4 used his rewrite package which includes direct computation rules as one form of justification for rewrites. It is a particularly important one because it does not require justification via functionality lemmas and so is the most efficient form of rewrite. The rewrite package is used to describe equivalences under direct computation by defining conversions encapsulating the desired behavior. Typically they appear in the library as ML objects named `<opid>_unroll`. The conversion for the Y combinator named `YUnrollC` encapsulates the fixed-point property of the Y combinator and unrolls one step of computation.

In this thesis, an additional level of abstraction hides the Y combinator. The `letrec` form is implemented through three display forms and abstractions that hide the underlying use of the fixed point combinator.

`*A letrec`	`(letrec f b[f])` $\overset{\text{def}}{=}$ `Y (`$\boldsymbol{\lambda}$`f.b[f])`
`*A letrec_body`	`= b` $\overset{\text{def}}{=}$ `b`
`*A letrec_arg`	`x b[x]` $\overset{\text{def}}{=}$ $\boldsymbol{\lambda}$`x.b[x]`

Thus

`(letrec f` $\vec{x}$ `=` b`[f,`$\vec{x}$`])` is defined to be `(Y (`$\boldsymbol{\lambda}$`f.`$\boldsymbol{\lambda}\vec{x}$`.`$b$`[f,`$\vec{x}$`]))`

The conversion `letrec_unrollC` captures the following computational behavior of applications of `letrec` terms.

`(letrec f t =` b`[f;t])(T)` $\rhd_R$ b`[(letrec f t =` b`[f;t]),T/f,t]` *i.e.* the recursive call is substituted for `f` in the term `b[f;t]` and the argument `T` is substituted for `t`.

3.4.1 An example definition

In this section an operator for list quantification is developed. The `list_3` library supporting the decidability theorems has four library objects associated with the definition of the operator: a display form object exhibiting a non-trivial use of the Nuprl display mechanism; an abstraction object defined by general recursion; an associated ML object that reflects the computational behavior of the operator via the rewrite system; and a theorem object which shows the definition is well-formed.

```
*D list_all_df
   ∀<x:element>∈<L:List>.<P:T⟶ℙi:L> ≝  list_all{}(<L>; <x>.<P>)
*A list_all
   ∀x∈L.P[x]  ≝
      (letrec list_all L =
                 list_ind L of [] => True | h::t => P[h] ∧ list_all t
      ) L
*M list_all_unroll
   let list_all_conv T =
       FwdMacroC  `list_all_unrollC`
         (AllC [UnfoldC `list_all`;
                letrec_unrollC;
                ReduceC;
                TryC (FoldC `list_all`)])
      T
   ;;
   let list_all_unrollC =
    SomeC [list_all_conv ⌜∀x∈[].P[x]⌝;
           list_all_conv ⌜∀x∈(h::t).P[x]⌝]
   ;;
   add_AbReduce_conv `list_all` list_all_unrollC;;
*T list_all_wf
   ∀T:𝕌. ∀P:T → ℙ. ∀L:T List.  ∀x∈L.P[x] ∈ 𝕌
   Extraction:
      λT,P,L.list-case(L) of [] => Axiom | u::v => %.Ax
```

The first entry is a display form object named `list_all_df`. This object provides a template for display of instances of the `list_all` operator. The naming convention in Nuprl is to give display-form objects names of the form *op_id_name*`_df` where *op_id_name* is the operator identifier of the abstraction being displayed. The display-form itself has two components appearing as "*[lhs]* $\overset{\text{def}}{=}$ *[rhs]*". The *lhs* is the display template with slots. In the display-form for `list_all` there are three: one slot for the variable `x` which is bound by the quantifier; one slot for the list `L` which is the domain of the quantification; and one slot for the predicate `P[x]` possibly containing free occurrences of the variable `x`. In a complete display-form, *rhs* is an instance of the operator being displayed. In this case the operator is `list_all{}(<L>; <x>.<P>)`, having the operator identifier `list_all`, and having no parameters (which would be enclosed within the set brackets), and having three meta-variables `L`, `x`, and `P` with the notation `<x>.<P>` indicating that `x` is bound in `P`. The Nuprl display form mechanism supports additional features not used here.

The second library entry is an abstraction with the name `list_all`. The naming convention for abstractions is to give them the same name as the operator identifier of the operator being defined. In the remainder of this paper, unless there is some interesting characteristic of a display-form itself, display definitions will not be shown, and abstractions will be displayed as above; *e.g.* with the instantiated display-form to the left of the " $\overset{\text{def}}{=}$ " and with the definition to the right. In this case the definition consists of a recursive function, defined via a `letrec` form, applied to the argument `L`.

Based on these definitions, the behavior of the abstraction for the `list_all` operator should be transparent. The computational behavior can be explained by considering the rewrite conversions defined in the ML object associated with its definition.

The ML object named `list_all_unroll` contains the code used to selectively unroll occurrences of the `list_all` operator. The details of the ML code are unimportant, but it is worth pointing out how selective rewrite support is provided for recursively defined functions. Upon evaluation of the ML object `list_all_unroll` by the system, the two objects `list_all_conv` and `list_all_unrollC` result in ML objects of type conversion. When applied by the rewrite system, the conversions exhibit the following behavior.

$$\forall \mathtt{x}\in\mathtt{[].P[x]} \ \rhd_R \ \ \mathtt{True}$$
$$\forall \mathtt{x}\in\mathtt{h::t.P[x]} \ \rhd_R \ \mathtt{P[h]} \wedge \forall \mathtt{x}\in\mathtt{h::t.P[x]}$$

The last line of the ML object adds the `list_all_unrollC` conversion to the list of conversions applied by the tactic `AbReduce` when an operator with operator id `‘list_all‘` is encountered having either the empty list or a cons as its primary argument. For a complete account of the rewrite system and the Nuprl ML system, the documentation provided with the system should be consulted.

The third library entry above is a well-formedness theorem for the `list_all` abstraction. The theorem is named `list_all_wf` following the convention used by the well-formedness tactics which will search for it by name and automatically apply it when well-formedness goals are induced during the proving process. The theorem says that for appropriately typed arguments, the `list_all` operator denotes a type. More precisely, for every type `T`, and for every `T List`, and for every proposition (function from `T` onto ℙ) the `list_all` operator is a member of all levels of the type universe hierarchy. Typically we will omit the presentation of well-formedness goals that simply state an abstraction is an element of some type universe.

The proof of `list_all_wf` provides the extract

```
λT,P,L.list-ind(L) of [] => Axiom | u::v => Axiom
```

This term is one function inhabiting the proposition

```
∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List.  ∀x∈L.P[x] ∈ 𝕌
```

The theorem was proved by induction on `L` and so extraction includes the the primitive recursive `list-ind` form which is the computational content of list induction. In the case the list is empty the extract evaluates to `Axiom`; this is because the element inhabiting the proposition `True∈𝕌` is `Axiom`. The inductive hypothesis of the list induction establishes that the recursive call is well-formed, thus in the case of a cons `u::v`, the extract evaluates to `Axiom`.

The following useful lemma characterizes the `list_all` operator in terms of the `x` in `T` such that `x` is a member of `L` under the discrete equality `eq`.

```
* THM list_all_all_lemma
∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List.  ∀eq:{T=₂}.
   ∀x∈L.P[x]⟸⟹(∀x:{x:T| x(∈eq) L} . P[x])}
```

The lemma says for every type `T`, every proposition `P`, every list `L`, and every discrete equality over `T`, the proposition `∀x∈L.P[x]` holds, if and only if for every `x` of type `T` that is in the list `L`, `P[x]` holds. The set type is being used here as a sub-typing mechanism to restrict attention to elements of `T` that happen to occur in the list `L`. We shall see a distinctly different application for the set type in the next section.

3.5 Extraction of general recursive content

Most of the material appearing in this section has appeared elsewhere [Cal97]. The definition just given for `list_all` does not apply the proofs-as-types interpretation. It is essentially classical program verification performed in the Nuprl type theory. The goal of this section is to show how to specify and prove a theorem having as its extract the same function as the one used to directly define the `list_all` operator above.

3.5.1 A first specification

The first task is to state a theorem whose inhabitants are of the correct type. The type we are interested in is given by the well-formedness theorem for the `list_all` operator, *i.e.*

```
 T:𝕌 ⟶ P:(T ⟶ ℙ) ⟶ L:T List ⟶ ℙ
```

We use the characterization given by the lemma `list_all_all` as a basis for the specification of the behavior of the function.

```
* THM list_all_exists_lemma
∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List.
   ∃p:ℙ. ∀eq:{T=₂}. p ⟸⟹ (∀x:{x:T| x(∈eq) L} . P[x])
```

Under the propositions-as-types interpretation we can understand the theorem as a specification for functions of type

```
 T:𝕌 ⟶ P:(T ⟶ ℙ) ⟶ L:T List ⟶ p:ℙ × 𝒯[p]
```

where $\mathcal{T}$`[p]` is the proposition

```
  ∀eq:{T=₂}. p ⟸⟹ (∀x:{x:T| x(∈eq) L} . P[x])
```

The elements of the type are the terms inhabiting (proving) the proposition. In this specification, `p` is a proposition (an element of type ℙ) that is true whenever $\mathcal{T}$`[p]` is inhabited.

Using the extract of this theorem, we can easily define a function that is extensionally equivalent to the one we are after by taking the first projection of the result of applying it to the appropriate arguments. Thus if `f` is the extract of the theorem, we can easily prove

```
∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List. (f(T)(P)(L)).1 ∈ ℙ
```

Where for any pair ⟨`x,y`⟩, ⟨`x,y`⟩`.1 = x`.

This is precisely the approach described in the Nuprl book [Con86, section 4.4] and elsewhere [NPS90, section 21.1]. But this approach fails if we are interested in using extracted programs as operators in proofs where we need efficient selective

```
λT,P,L.
 (letrec f (L) =
    if null(L)
    then <True, λeq.<λ%,x.any Ax, λ%.Ax>>
    else h::t = L in let <p,%1> = (f(t)) in
       <P[h] ∧ p,
        λeq.<λ%1@0.let <%2,%3> = %1@0  in
              λx.let <%4,%5> = (%1(eq))  in
                  let <%9,%10> =
                     (ext{discrete_equality_properties}{i:l}(T)(eq)) in
                   case ext{decidable__assert}((eq(x)(h)))
                     of inl(%14) => let <%18,%19> = (%9(x)(h)) in %2
                      | inr(%15) => %4(%3)(x),
              λz.<z(h), let <%3,%4> = (%1(eq)) in %4((λx.z(x)))>
             >
       >
    fi)(L)
```

Figure 3.1: Extract of `list_all_exists_lemma`

unfolding of terms and partial evaluations. Figure[1] shows the term extracted from a natural proof of the `list_all_exists_lemma` after one step of reduction under the `Reduce` tactic. Although the term in the figure is correct and computes reasonably when applied to ground terms, there is no obvious way to partially evaluate the term on a non-ground argument without blowing up the term size with every unfolding. The parts of the program we are not interested in are contained in the right element of the pair computed by the program.

3.5.2 Minimization of the logical content

One idea to maintain the existential specification but to minimize the logical content is to hide most of the logical content in a set type.

```
* THM list_all_exists_lemma_1
∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List.
   ∃{p:ℙ | ∀eq:{T=₂}. p ⟸⟹ (∀x:{x:T| x(∈eq) L} . P[x])}. True
```

Proofs of this theorem still compute pairs but the right element of the pair is simply the term `Ax`.

The following theorem justifies the elimination of the existential quantifier completely.

```
* THM exists_iff_set
∀T:𝕌. ∀P:T ⟶ ℙ. ∃{x:T | P[x]}. True ⟸⟹ {x:T | P[x]}
```

This lemma leads to the elimination of the existential quantifier in favor of the set type.

3.5.3 A Refined Specification

The Nuprl set type was used above to define a subtype, but now we use it to discard the unwanted computational content carried by proofs of specifications based on the existential quantifier.

Replacing the existential quantifier by the set type as follows gives the type we're interested in.

```
* THM list_all_ext
∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List.
   {p:ℙ| ∀eq:{T=₂}. p ⟸⟹ (∀x:{x:T| x(∈eq) L} . P[x])}
```

At first sight this specification seems curious. Reading it literally it says

> *For every type* `T`*, every proposition on type* `T`*, and every* `T list`*, the set of propositions* `p` *such that* $\mathcal{T}$`[p]` *holds, is true.*

Recall, the Nuprl logic is an encoding in the type theory via the Curry Howard isomorphism; every statement in the logic is an encoding for type. Under the propositions-as-types interpretation, truth corresponds to inhabitation, and so the goal of every proof is to show that the encoded type is inhabited. The proof rules for the set type are noticeably similar to those for dependent product, but the extract of the set type does not include the proof that $\mathcal{T}$`[p]` holds. The type of this specification is the one we want.

```
T:𝕌 ⟶ P:(T ⟶ ℙ) ⟶ L:T List ⟶ ℙ
```

Having a statement of the theorem with the correct type and with the intended meaning we must prove it in a way that generated the extract we are most interested in. Specifically, we want to prove the theorem so that the extract is a recursive function defined by `letrec`. We will return to the proof of the theorem `list_all_ext` after developing the necessary mechanism.

3.5.4 List Induction Extracting `letrec`

Induction on lists is defined by the Nuprl inference rule `listElimination`. The application of the rule generates the extract `list_ind` defined above. The `ListInd` tactic is used to apply the rule. Our goal is to develop a new list induction tactic whose behavior mimics the `ListInd` tactic but having a recursion combinator defined using `letrec` as its extract.

The following theorem captures the familiar list induction principle.

```
* THM list_ind_with_y
∀T:𝕌. ∀P:T List → ℙ'.
  P[[]] ⇒ (∀u:T. ∀v:T List.  P[v] ⇒ P[u::v]) ⇒
   (∀M:T List. P[M])
```

Since our goal is a specific extract, we explicitly provide the witness term we are interested in.

```
 λT,P,b,g.
    letrec f (L) = if null(L) then b
                   else  h::t = L in g(h)(t)(f(t))
                   fi
```

Given the witness, the remainder of the proof is a verification that the witness term does indeed inhabit the type specified by the theorem. The proof is surprisingly intricate although it is modeled on a similar induction principle developed by Howe [How93] for natural numbers and having a recursion combinator defined using `Y` as its extract.

A new tactic, `ListIndY`, facilitates the application of the induction principle. `ListIndY` duplicates the behavior of the ordinary `ListInd` tactic in most contexts. Taking as argument the hypothesis number of the induction variable, the tactic constructs the induction proposition (the function `P` of type `T` $\longrightarrow$ $\mathbb{P}$) and then instantiates the `list_ind_with_y` lemma. The instantiation of the lemma generates a number of well-formedness goals which are, in most contexts, easily discharged by the `Auto` tactic. Of the three remaining goals, one corresponds to the base case, the other to the induction step, and the third to the original sequent with the induction principle fully instantiated as a hypothesis. This third subgoal is discharged by an application of `HypBackchain THEN Auto`, leaving only two subgoals which match those produced by the `ListInd` tactic.

Extracts of theorems proved with the `ListIndY` tactic refer indirectly to the computational content of this theorem by mentioning `ext{list_ind_with_y}{i:l}`. An ML object extends the reduction system to automatically unfold the extract when it is encountered by `Reduce`.

The context in which `ListIndY` does not behave as its counterpart `ListInd` is when proving well-formedness goals. The `ListIndY` tactic cannot be used to show well-formedness. This is because the instantiation of the `list_ind_with_y` lemma generates well-formedness subgoals for the induction proposition, and these will essentially be identical to the original well-formedness lemma. However, this is not a limitation to the methodology since well-formedness goals for abstractions defined directly via extracted terms are trivially proved by appeal to the theorem the abstractions are extracted from.

Application of the tactic will be shown in the next section when we return to the proof of the `list_all_ext` theorem.

3.5.5 A proof and extract

In this section we step through the proof of `list_all_ext` until we've completed as much as is required to generate the desired extract.

Recall the statement of the theorem (displayed here as a sequent with no hypotheses).

```
⊢ ∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List.
    {p:ℙ| ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) L} . P[x])}
```

Stripping off the quantified variables results in the following sequent.

```
1. T: 𝕌
2. P: T ⟶ ℙ
3. L: T List
⊢ {p:ℙ| ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) L} . P[x])}
```

The proof is by induction on the list `L` so we apply the tactic `ListIndY (-1)`, which results in two subgoals.

The first is the base case where `L` has been replaced by the empty list `[]` in the conclusion.

```
⊢ {p:ℙ| ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) []} . P[x])}
```

To complete the proof we must choose a witness for `p`. Noticing that `x(∈eq) []` is false, we see that the right side of the if and only if is vacuously true and so we supply `True` as the witness for `p`. At this point the computational content on this branch of the proof is complete. The resulting subgoal is to verify the logical property that the proposition defining the set is indeed true when `True` substituted for `p`.

The subgoal for the inductive case is the following.

```
4. u: T
5. v: T List
6. {p:ℙ| ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) v} . P[x])}
⊢ {p:ℙ| ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) (u::v)} . P[x])}
```

Decomposing the induction hypothesis results in the following.

```
6. p: ℙ
[7]. ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) v} . P[x])
⊢ {p:ℙ| ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) (u::v)} . P[x])}
```

Hypothesis `7` is a hidden hypothesis (denoted by the brackets) that cannot be used to build computational content. But we construct the witness for the set type in the conclusion without relying on `7`. The variable `p` of hypothesis `6` corresponds to the proposition that is true iff the specification holds for the list `v`. Thus, the proposition `P[u] ∧ p` is the witness for the set type in the conclusion.

Once the witness is provided, the computational content is completed and the hidden hypotheses can be used in the verification of the resulting logical property. This unhiding is automatically done by the system in the proof step that provides the witness. A key to making the proof go through is to decompose any set types required for the verification so that the hidden hypotheses are exposed at the point in the proof that the witness is provided.

The raw extract of this proof is the following term[1].

```
λT,P,L.
 (λ%1.(λ%2.%2(L))
  ((%1(T)(λ₂L.{p:ℙ| ∀eq:{T=₂}. p ⟺ (∀x:{x:T| x(∈eq) L} . P[x])})
     (True)
  ((λu,v,%.(λ%1.P[u] ∧ %1)(%))))))
 (ext{list_ind_with_y}{i:l})
```

[1]Interested readers will note the second-order λ-term on the third line which was generated by the `ListIndY` tactic.

One step of reduction under `Reduce` (extended with `list_ind_with_y_unrollC` defined above) results in the following term.

```
λT,P,L.
   (letrec f (L) = if null(L) then True
                   else h::t = L in P[h] ∧ f(t)
                   fi )(L)
```

This is exactly the program we are interested in, modulo renaming of the recursion variable `f`.

3.5.6 Display forms, abstractions, and well-formedness theorems

To use the term just extracted in the same way the original `list_all` operator is used, we must define a display form, an abstraction (having the extracted term as the definition), and a well-formedness theorem. The ML function `add_extract_abs`, accepts a display-form template and the name of the theorem, and constructs the display form and the abstraction; it also automatically constructs the well-formedness goal and proves it by backchaining through the theorem generating the extract. All arguments outside the scope of the application of `letrec` that occur within the scope of the body of the `letrec` are made parameters of the abstraction. In practice this approach seems to work. Note that the first argument `T` in the extract above does not occur in the body of the `letrec` and so is not included as an argument of the generated abstraction.

The extract term can be directly used in contexts where a display form is not required. It is referred to by entering the name of the theorem (in this case `list_all_ext`) in a term slot. When used in this mode, the rewrite system can be extended to include conversions for selectively unfolding and computing with the extract whenever it is applied to arguments of the appropriate type.

3.6 Efficient Induction Schemes

Nuprl supports primitive recursive induction over the built-in types such as numbers and lists. It also supports induction on recursive types. Nuprl users also extend the available forms of induction by proving new induction schemes in the form of lemmas.

A type of well-founded binary relations over a type is defined in the Nuprl standard library as follows.

```
*A wellfounded
WellFnd{i}(A;x,y.R[x; y]) ==
   ∀P:A → ℙ.
     (∀j:A. (∀k:A. R[k; j] ⇒ P[k]) ⇒ P[j]) ⇒
   {∀n:A. P[n]}
```

Using a lemma stating, for instance, that `WellFnd{i}(ℕ;x,y.x < y)`, we can apply the lemma to do well-founded induction on the natural numbers over the ordinary less-than ordering. Nuprl4.2 provides tactic support doing these inductions.

The following recursion scheme is an inhabitant of this type.

```
   λP,g. (letrec f(n) = g(n)(λk,p. f(k)))
```

Here g corresponds to the computational content of the induction hypothesis. In this scheme, g takes two arguments, the first being the principal argument on which the recursion is formed, while the second argument to g is a function inhabiting the proposition ∀k:A. R[k; j]⇒ P[k], *i.e.* a function which accepts some element k of type A along with evidence for R[k;j] and which produces evidence for P[j]. In the scheme, the evidence that R[k;j] holds takes the form of the argument p to the innermost **λ**-binding. The variable p occurs nowhere else in the term and does not contribute to the actual computation of P[j]; instead it is a vestige of the typing. In the context of any complete proof, this argument will be a term justifying R[k;j]. This term is not part of what one would ordinarily consider part of the computation.

As an alternative, we have defined another notion of well-foundedness that hides the ordering under R in a set type.

```
* ABS sq_stable_wellfounded
WF{i}(A;x,y.R[x; y]) ==
    ∀P:A ⟶ ℙ.
      (∀j:A. (∀k:{k:A| R[k; j]} . P[k]) ⇒ P[j]) ⇒
   {∀n:A. P[n]}
```

This type can only usefully be applied in proofs when R is squash stable, hence the name. However, it should also be noted that type equality is trivially squash stable, as are the order relations numbers. Indeed, for program development, this constraint on R seems not to matter since it is hard to imagine how a termination ordering that is not squash stable could be applied.

Since the ordering relation is hidden in the right side of a set type, it cannot contribute to the computational content. The recursion scheme extracted from a proof of this type is nearly identical to the previous one, but the extra level of lambda-abstraction is gone.

```
    λP,g. (letrec f(n) = g(n)(λk. f(k)))
```

Using the `sq_stable_wellfounded` relation on natural numbers we are able to define the following measure induction principle.

```
* THM measure_ind
∀T:𝕌. ∀ρ:T ⟶ ℕ.  WF{i}(T;x,y.ρ x < ρ y)
Extraction:
  λT,ρ,P,g.(letrec f(n) = (g(n)(λk.f (k))))
```

Note that the measure function ρ does not occur in the body of the extract.

The proof of intuitionistic decidability presented below in chapter 5 is by induction on the lexicographic ordering of inverse images onto natural numbers. Using a squash stable well-founded relation, we are able to define this induction scheme as follows.

```
* THM lexicographic_measure_ind
∀T:𝕌. ∀ρ,ρ':T ⟶ ℕ.
   WF{i}(T;k,j.ρ k < ρ j ∨ (ρ k = ρ j ∧ ρ' k < ρ' j))
Extraction:
  λT,ρ,ρ',P,g.(letrec f(n) = (g n (λj.f j)))
```

It is interesting to note that the simplest proof for this induction principle (as for instance given in [MW90]) is based on the least element principle. This principle is the contrapositive of the standard definition of well-founded and therefore is not, in general, constructively valid.

3.7 Related Work

The methodology presented in this section owes much to [How93]. In that paper, Howe described verification and extraction methodologies applied to Boyer-Moore's fast majority algorithm in Nuprl 3. He developed a natural number induction theorem having as its extract the recursion combinator defined by `Y`. Although Howe mentions the possibility of using the set type to clean-up the extracts, he did not do so there.

The Nuprl 4.2 `int_1` library contains an induction lemma for complete induction, having as its extract a recursion combinator. The proof of that lemma was based on Allen's typing of Y and has served as a basis for the proofs of all the well-founded induction lemmas presented above.

Paulson [Pau86a] presented a theory of well-founded relations for constructing recursion schemes other than primitive recursion. In that paper he gives full details of the proofs with the idea that they might be reused in provers for type theory such as Nuprl. Interestingly, Paulson remarks that the only induction principle of the Boyer and Moore prover, a system known among automated systems for its inductive strength, is well-founded inductions over lexicographic products of inverse images of $<$. This is our lexicographic measure induction.

Certainly it is well known in the Nuprl community how the set type can be used to hide unwanted computational content. However, the approach has rarely been applied in practice. Indeed, even in examples where the goal of the exercise is to extract computational content [Con86, pg.86–93], they prefer the existential quantifier to the set type and choose to project the first element of the pair.

The Calculus of Constructions is similar to Nuprl and the problem of extracting clear programs from proofs in Coq has also been addressed there. The approach developed by Paulin-Mohring [Moh86, PM89] to allow the possibility of clean extracts in Coq is based on separating "computationally informative" and "non-informative" propositions by declaring them to either be of type `Set` or `Prop`. If `A` is of type `Prop`, the elements of `A` are proofs which are ignored from the computational view, and `A` is strictly logical. If `A` has type `Set`, then elements of `A` are "program developments" from which Coq extracts programs that are correct with respect to `A`. This duality of types provides a method of eliminating the parts of the program corresponding to the logical specification.

`PX` [HN88] uses a similar approach based on separating non-computational content from computationally interesting content. This system of partitioning did not apparently work in practice; in a later paper [Hay94] Hayashi motivates a new system based on set types and intersection types by claiming the earlier was not suitable for practical code development.

3.8 Remarks

The work reported on in this Chapter was motivated by need. In large proofs, it often happens that the form of some subprogram is well known. In that case, the existing Nuprl methodology works well in allowing the definition and verification of the appropriate property. However, when development is driven from the proof side, often, the logical specification is known but the implementation is not. In the methodology developed here, extracted programs obtain the same status as verified programs with respect to later use in other contexts of definition and proof.

We shown by example how to massage the statement of the theorem `list_all_exists_lemma` into a form of the correct type. The methodology has been applied to define a number of abstractions being applied in a proof of propositional intuitionistic decidability.

We have have developed tactics which generate proofs having `letrec` forms as their extracts. The extracts of the theorems proved with the tactics coexist with established Nuprl methodology for recursive functions.

Perhaps most surprisingly, the proof of the theorem using the set-type specification, `list_all_ext`, is *identical* to the proof for the existential version of the lemma `list_all_exists_lemma`. This seems to be generally true of pure proofs – a natural pure proof of the existential form is identical to the natural proof of the reformulated theorem with the set type replacing the existential quantifier. Note that this does not hold for computationally explicit proofs that provide explicit witnesses (a pair) to discharge an existential subgoal. The pair is not the of the correct type to eliminate a set, and so the proof will fail if it is rerun on a restated theorem where existential quantifiers have been changed to set type quantifiers.

The fact that pure proofs often rerun with no change suggests that in many contexts, the existential quantifier, although it is the natural form, may be the wrong one.

Chapter 4

Decidability of Classical Propositional Logic

A formalization of a sequent presentation of classical propositional logic is described in this chapter. The formalized mathematics is then used to produce a fully formal proof of decidability. The program extracted from this proof is exceedingly clear; indeed, it is the natural recursive program an experienced functional programmer would naturally write. The extracted program is efficient in that no artifacts of the proof remain in the extract. The extracted code has been run on a number of examples.

The formal theories developed in support of the decidability proof were designed to give as clear an account of the material as can be found in the best published presentations, as for example in Smullyan [Smu68], Gallier [Gal86a] or Nerode and Shore [NS94]. The proof of the main theorem presented in this chapter follows the one outlined by Constable and Howe in [CH90]. However, the semantic base used here is Kleene's three-valued logic.

The Nuprl formalization of the syntax and semantics of the Nuprl logic is given in the next section and the proof of decidability is given in the following section. A analysis and discussion of the program extracted from the proof follows.

4.1 Type Theoretic Formalization

In this section the Nuprl definitions supporting the statement and proof of the decidability theorem are presented. The syntax of propositional formulas is formalized using Nuprl's recursive types. The semantic notion of a Kleene (partial) valuation on formulas is also defined. This requires the development of a small theory of three-element types as well as the definition of the logical operators of Kleene's strong three-valued logic. Kleene valuations are defined over three-valued assignments using the operators. The semantics for formulas is then defined in terms of Kleene valuations by giving definitions for formula satisfiability, formula falsifiability and a unique definition of formula validity. Subsequently, a sequent type is defined as the Cartesian product of lists of formulas. The semantics of sequents is given by lifting the formula semantics in the natural way to define sequent satisfiability, sequent falsifiability, and sequent validity.

The decidability theorem is stated and proved in the next section using these definitions. The proof is by induction on the complexity of sequents where the measure is the number of propositional operators occurring in the sequent. Thus, the definition of sequent rank (defined in terms of formula rank) is also given here.

4.1.1 Variables and Formulas

The Nuprl formalization of the logic formulas is by a recursive type.

```
*A Formula
Formula  ≝  rec(F.Var | F | (F × F) | (F × F) | (F × F))
```

The `Formula` type abstraction is defined to be the recursive type whose members are a disjoint union of five elements. The first element of the disjoint union is the uninterpreted type `Var` or propositional variables. Since the variable `F`, bound by the rec-type operator, doesn't occur in the first component, terms of the form `inl(x)`, where `x` is an element of the type `Var`, form the basis elements of the recursive type. The second component of the disjoint union is an instance of the bound variable `F` denoting a recursively smaller element of the formula type. The third, fourth, and fifth elements of the disjoint union are the Cartesian products (pairing) of two recursively smaller formulas. When the semantics of the propositional formulas is defined below it will become clear that the second disjunct of the formula type denotes the negation of the formula F, and the pairs of formulas in the third, fourth, and fifth disjuncts denote the operators for conjunction, disjunction, and implication.

It should be remarked that the recursive type applied to this purpose is exceedingly efficient. Smullyan devotes three pages to this. The recursive type definition of syntax gives the uniqueness of decomposition for free since the `rec` type automatically supports structural induction on elements of the type.

For concreteness, the variables are defined to be Nuprl atoms, however, any discrete type will do.

4.1.1.1 Constructors

To facilitate manipulation of the type `Formula`, a collection of constructors and a destructor are defined. The constructors and their well-formedness theorems are as follows.

`*A fvar`	⌜x⌝ $\overset{\text{def}}{=}$ `inl x`
`*T fvar_wf`	`∀x:Var. (⌜x⌝ ∈ Formula)`
`*A fnot`	(⌜∼⌝p) $\overset{\text{def}}{=}$ `inr inl p`
`*T fnot_wf`	`∀p:Formula. (⌜∼⌝p ∈ Formula)`
`*A fand`	(p⌜∧⌝q) $\overset{\text{def}}{=}$ `inr inr inl <p,q>`
`*T fand_wf`	`∀p,q:Formula. (p⌜∧⌝q) ∈ Formula`
`*A for`	(p⌜∨⌝q) $\overset{\text{def}}{=}$ `inr inr inr inl <p,q>`
`*T for_wf`	`∀p,q:Formula. (p⌜∨⌝q) ∈ Formula`
`*A fimp`	(p⌜⇒⌝q) $\overset{\text{def}}{=}$ `inr inr inr inr <p,q>`
`*T fimp_wf`	`∀p,q:Formula. (p⌜⇒⌝q) ∈ Formula`

Thus, the term `(((⌜x⌝⌜⇒⌝⌜y⌝)⌜⇒⌝⌜x⌝)⌜⇒⌝⌜x⌝)` is a formula.

A formula of the form ⌜x⌝, where `x` denotes an element of type `Var`, will be called an *atomic formula* (or simply *atomic*) and all others are called *non-atomic formulas*.

4.1.1.2 Case analysis

The `formula_case` operator defined below is the destructor for the `Formula` type. It is defined using nested case analysis on the disjoint union type. A nested series of `decide` operators gives the case analysis for the type `Formula`. The definition is as follows:

```
*A formula_case
case F:
      ⌜x⌝ ⟶ varC[x];
      ⌜∼⌝p1 ⟶ notC[p1];
      p2⌜∧⌝p3 ⟶ andC[p2; p3];
      p4⌜∨⌝p5 ⟶ orC[p4; p5];
      p6⌜⇒⌝p7⟶impC[p6; p7];
    ≝
   decide F of
     inl(x) => varC[x]
     | inr(F) => decide F of
     inl(p1) => notC[p1]
     | inr(F) => decide F of
     inl(x) => let p2,p3 = x in andC[p2; p3]
     | inr(F) => decide F of
     inl(x) => let p4,p5 = x in orC[p4; p5]
     | inr(x) => let p6,p7 = x in impC[p6; p7]
```

In the abstraction, the display slots contain occurrences of the second-order variables `varC[x]`, `notC[p1]`, `andC[p2;p3]`, `orC[p4;p5]` and `impC[p6,p7]`. In an instantiation of the `formula_case` operator, terms, possibly containing free occurrences of the bracketed variables, are substituted for the variables. The bracketed variables are are bound by the names to the left of the ⟶.

As an example use of the operator we define a function which collects the principal subformulas of a formula into a list.

```
*A principal_subformula
    principal_subformula(F)  def=  case F:
                                   ⌜y⌝ ⟶ [];
                                   ⌜∼⌝p ⟶ (p::[]);
                                   p⌜∧⌝q ⟶ (p::q::[]);
                                   p⌜∨⌝q ⟶ (p::q::[]);
                                   p⌜⇒⌝q ⟶ (p::q::[]);
```

In this instantiation of the `formula_case` operator, the display-form variable `x` is bound to `y` and the term denoting the empty list (`[]`) is bound to the second-order variable `varC[y]`. In the second case, the variable `p1` is bound to `p` and the term `(p::[])` is bound to the second-order variable `notC[p]`. The other cases are similarly explained. Thus, for a formula `F`,

`principal_subformula(⌜∼⌝F)` $\rhd_R$ `F::[]`

e.g. under the computation system extended to include the behavior of the `principal_subformula` operator, `principal_subformula(⌜∼⌝F)` evaluates to the list containing the single formula `F`.

4.1.1.3 Formula rank

Having defined the type `Formula` and the supporting constructors and destructor, we define a well-founded measure on formulas. The following measure function, a count of the number of operators in a formula, is the simplest and most natural ordering to consider.

```
*A formula_rank  $\boldsymbol{\rho}$  $\overset{\text{def}}{=}$
    letrec measure(f) =
      case f:
        $\ulcorner$x$\urcorner$ $\longrightarrow$ 0;
        $\ulcorner\sim\urcorner$p $\longrightarrow$ (measure(p) + 1);
        p$\ulcorner\wedge\urcorner$q $\longrightarrow$ (measure(p) + measure(q) + 1);
        p$\ulcorner\vee\urcorner$q $\longrightarrow$ (measure(p) + measure(q) + 1);
        p$\ulcorner\Rightarrow\urcorner$q $\longrightarrow$ (measure(p) + measure(q) + 1);
```

The well-formedness theorem for the `formula_rank` function certifies it is a function from formulas to natural numbers.

```
*T formula_rank_wf  $\boldsymbol{\rho}$ $\in$ Formula $\longrightarrow$ $\mathbb{N}$
```

4.1.2 Semantics

In standard treatments of decidability (and completeness) for classical propositional logic, for example as found in Mendelson [Men79], *truth assignments* are total functions mapping the (countably infinite) propositional variables onto the Booleans. Smullyan [Smu68] considers finite functions mapping the set of variables occurring in a formula (or set of formulas) onto the Booleans. In both presentations it as shown how assignments uniquely determine *Boolean valuations.* The decision procedure developed here not only determines whether a propositional sequent (to be defined below) is valid, but in the case the sequent is not valid, it returns a falsifying assignment. In the natural development, falsifying assignments generated by the decision procedure are finite functions and, furthermore, their domains may not even include all variables occurring in the sequent. Thus, the formal development must either account for an arbitrary extension of the domain of a partial falsifying assignment to include all variables occurring in the sequent, or the notion of partial assignments must be accounted for in the semantics. The second approach is developed here.

We define the semantics of propositional logic in terms of Kleene's strong three-valued logic [Kle52]. A Kleene valuation reflects the classical interpretations of the standard propositional connectives under fully determined assignments (those assigning true or false to every variable in the formula); but also, when a partial assignment has "enough information" to determine the truth or falsity of a formula, the Kleene valuation induced by it does as well. This strategy is sometimes called "short circuit evaluation". For example, if either conjunct of the formula $\mathtt{p}\wedge_K\mathtt{q}$ is *false* under the Kleene valuation induced by a partial assignment a, then $\mathtt{p}\wedge_K\mathtt{q}$ is *false* under the valuation too. It does not matter what value the other conjunct has, or even if it is defined. Under all extensions of a, the valuation of $\mathtt{p}\wedge_K\mathtt{q}$ is false. Thus, once determined, an assignment remains fixed. Similar rules apply for the other operators which are formally defined below.

To proceed with the formalization in Nuprl we first define a three-valued type.

4.1.2.1 A three-element type

There are a number of ways to formalize a three element type *e.g.* $\mathbb{N}_{\mathbf{2}}$. Here we choose to use a three-way disjoint union.

`*A Three` $\quad \mathbf{3} \overset{\text{def}}{=}$ `Unit | Unit | Unit`

The single element of `Unit`, called "dot", is displayed as '·'. Thus, **3** is the type containing the injections of · into the first, second, or third components of the three part disjoint union. We provide names and display forms for each of the three elements below.

```
*A Three_false    F3  =def  inl ·
*A Three_undef    ?3  =def  inr inl ·
*A Three_true     T3  =def  inr inr ·
```

A case discriminator for the three-valued type is also defined.

```
*A Three_case
case x: F3 ⟶ case0; ?3 ⟶ case1; T3 ⟶ case2;  =def
    decide x of
      inl(zero) => case0
    | inr(one_or_two) =>
          decide one_or_two of
            inl(one) => case1
          | inr(two) => case2
```

Continuing with the development of the theory, we define two tactics: `ThreeInd` and `ThreeNEQ`. Following the convention used in the Nuprl V4 tactics, we name the tactic that does the case analysis on the type **3**, `ThreeInd`.[1]

```
H, (i) x:3, H' >> C[x] by ThreeInd i
    H, H'[F3/x] >> C[F3/x]
    H, H'[?3/x] >> C[?3/x]
    H, H'[T3/x] >> C[T3/x]
```

[1]To read the rule-like characterization of the tactic: the top line is the goal sequent and the indented lines below it are the three subgoals generated by application of `ThreeInd` to hypothesis number `i` in the goal.

If a variable `x` is declared to be of type **3** in hypothesis `i` of a Nuprl sequent, the application of the tactic `ThreeInd i` generates three subgoals, one for each of the three elements of the type. The subgoals are formed by substituting one of `F3`, `?3`, or `T3` for occurrences of the variable `x` in the hypotheses to the right of `x` in the hypothesis list and in the goal of the sequent. The raw extract generated by application of the ThreeInd tactic is of the following form.

```
decide x of
  inl(x1) => Ext0
| inr(y)  => decide y of
               inl(x) => Ext1
             | inr(y1) => Ext2
```

Here, Ext_0, Ext_1, and Ext_2 denote the extracts of the proofs of the three subgoals generated by the tactic. Observing that this term schema is an instance of the `Three_case` abstraction defined above, we will use the following tidied version, formed by folding the abstraction `Three_case`, when displaying the extract:

```
case x: F3 ⟶ Ext0; ?3 ⟶ Ext1; T3 ⟶ Ext2;
```

The tactic `ThreeNEQ` solves goals (generating no subgoals) of the following form,

H`,` `(i)` a `=` $b \in \mathbf{3}$`,` H' `>> C` by `ThreeNEQ i`

where a and b are different constants of type **3** and hypothesis `(i)` falsely asserts their equality. The extract generated by the application of the tactic is a term which, applied to any argument, returns the constant `Axiom`. The raw extract and its reduction are shown here:

`λ%.(λ%1.Axiom) Axiom` $\triangleright_R$ `λ%.Axiom`

The following theorem asserts **3** is a discrete type, that is, that the equality on **3** is decidable. This theorem is the first having a proof with interesting computational content. Since it is the first such proof, we examine it and its extracted term in some detail.

```
*T decidable__equal_Three
∀x,y:3. Dec{x = y ∈ 3}
```

The first step of the proof is the elimination of the outermost universal quantifiers by the tactic `UnivCD THENA Auto`. This yields the following Nuprl sequent.

```
 1. x: 3
 2. y: 3
 ⊢ Dec{x = y ∈ 3}
```

The extract resulting from this proof step has the form **λ**`x,y.`***Ext***, where *Ext* is the extract of the proof of resulting subgoal.

The next step in the proof is case analysis on `x` and then on `y`. Two applications of the `ThreeInd` tactic accomplish this. This results in nine subgoals.

```
 1* ⊢ Dec{F3 = F3 ∈ 3}
 2* ⊢ Dec{F3 = ?3 ∈ 3}
 3* ⊢ Dec{F3 = T3 ∈ 3}
 4* ⊢ Dec{?3 = F3 ∈ 3}
 5* ⊢ Dec{?3 = ?3 ∈ 3}
 6* ⊢ Dec{?3 = T3 ∈ 3}
 7* ⊢ Dec{T3 = F3 ∈ 3}
 8* ⊢ Dec{T3 = ?3 ∈ 3}
 9* ⊢ Dec{T3 = T3 ∈ 3}
```

The extract resulting from this step will be nested occurrences of the `Three_case` operator with the first splitting on `x` and the second on `y`.

Each of the nine cases is easily proved. Recall that `Dec{P}` is the constructive disjunction `P∨¬P`. To prove cases `1`, `5`, and `9`, the first disjunct is selected, which in turn is discharged by the `Auto` tactic. Equality terms viewed as types, when true, have as their inhabitants the single element denoted by the constant `Axiom`. The proofs of each of these three cases contribute the extract `inl(Axiom)` to their respective case splits.

In the six other cases, the equality is false; to prove the theorem, the second disjunct is chosen, resulting in a subgoal having the negated form of the equality as its consequent. Eliminating the negation results in a false hypothesis which is then discharged by the `ThreeNEQ` tactic. The proofs of these six cases contribute the extract `inr(λ%.Axiom)` to the corresponding case.

The extract of the entire proof is a term deciding if two elements of the type **3** are in fact equal.

```
λx,y. case x: F3 ⟶ case y: F3 ⟶ inl(Axiom) ;
                           ?3 ⟶ inr(λ%.Axiom) ;
                           T3 ⟶ inr(λ%.Axiom) ;;
              ?3 ⟶ case y: F3 ⟶ inr(λ%.Axiom) ;
                           ?3 ⟶ inl(Axiom) ;
                           T3 ⟶ inr(λ%.Axiom) ;;
              T3 ⟶ case y: F3 ⟶ inr(λ%.Axiom) ;
                           ?3 ⟶ inr(λ%.Axiom) ;
                           T3 ⟶ inl(Axiom) ;;
```

Thus, to decide `x = y` $\in$ **3** it is enough to apply the function to `x` and `y` and then to observe whether it returns a left or right injection; the content under the `inl` or `inr` is not used. This function is evidence for the proposition that the type is discrete.

4.1.2.2 Kleene's Strong Three-Valued Logic

In this section the operators of Kleene's three-valued logic are defined over the type **3**. Inspection of the definitions below reveals that on inputs restricted to `F3` and `T3` (which are to be interpreted as false and true respectively and will often be referred to as such below), the operators behave exactly as the familiar boolean operators of the same names. More technically, following Kleene [Kle52], we may say these operators are uniquely determined as the strongest possible regular extensions of the classical 2-valued operators.

```
*A K_not     ∼_K p ==    case p:  F3⟶ T3;
                                  ?3⟶ ?3;
                                  T3⟶ F3;
```

Thus, for negation the undefined value `?3` is a fixedpoint and, as is the case for the other Kleene operators, on the values `F3` and `T3` the Kleene operator reflects the behavior of its two-valued counterpart.

For a conjunction, in the case that one of `p` or `q` is false then the conjunct $\mathtt{p}\wedge_K\mathtt{q}$ is false too. A conjunction is undefined either if one of the conjuncts is true and the other is undefined or if they're both undefined. It is true otherwise.

```
*A K_and     p ∧_K q  =def case p:  F3⟶ F3;
                                    ?3⟶ case q: F3⟶ F3;
                                                 ?3⟶ ?3;
                                                 T3⟶ ?3;;
                                    T3⟶q;
```

For disjunctions, if one of p or q is true then the disjunction p$\vee_K$q is too. It is undefined either if one disjunct is false and the other is undefined or if both disjuncts are undefined. It is false otherwise.

```
*A K_or      p ∨_K q  =def case p:  F3⟶ q;
                                    ?3⟶ case q:   F3⟶ ?3;
                                                   ?3⟶ ?3;
                                                   T3⟶ T3;;
                                    T3⟶ T3;
```

For an implication p$\Rightarrow_K$q, if either p is false or q is true then the implication is true as well. An implication is undefined if p is true and q is undefined or if both p and q are undefined. A Kleene implication is false otherwise.

```
*A K_imp     p ⇒_K q  =def case p:  F3⟶ T3;
                                    ?3⟶ case q:   F3⟶ ?3;
                                                   ?3⟶ ?3;
                                                   T3⟶ T3;;
                                    T3⟶ q;
```

The well-formedness theorems for the Kleene operators exhibit their closure over the type **3**.

```
*T K_not_wf  ∀p:3.(∼_K p ∈ 3)
*T K_and_wf  ∀p,q:3.(p ∧_K q ∈ 3)
*T K_or_wf   ∀p,q:3.(p ∨_K q ∈ 3)
*T K_imp_wf  ∀p,q:3.(p ⇒_K q ∈ 3)
```

4.1.2.3 Assignments and Kleene Valuation

The type of three-valued assignments is defined as follows.

```
*A Assignment:          Assignment  $\overset{\text{def}}{=}$  Var $\longrightarrow$ $\mathbf{3}$
```

The `valuation` function recursively computes the Kleene valuation of the formula `F` under a partial assignment `a`. The valuation of `F` under assignment `a` (displayed as `(F under a)`) is defined as follows.

```
*A valuation
(F under a)   $\overset{\text{def}}{=}$  (letrec val f =
                    case f:
                      $\ulcorner$x$\urcorner$ $\longrightarrow$ (a x);
                      $\ulcorner\sim\urcorner$p $\longrightarrow$ $\sim_K$ val p;
                      p$\ulcorner\wedge\urcorner$q $\longrightarrow$ val p $\wedge_K$ val q;
                      p$\ulcorner\vee\urcorner$q $\longrightarrow$ val p $\vee_K$ val q;
                      p$\ulcorner\Rightarrow\urcorner$q $\longrightarrow$ val p $\Rightarrow_K$ val q;
                 )  F
```

The abstraction is defined by the application of the recursive procedure `val` to the formula `F`. The body of the recursive procedure is defined via case analysis on the parameter `f`. In the base case, the formula `f` is a formula of the form $\ulcorner$`x`$\urcorner$, then the result is the value returned by the application of assignment `a` to the variable `x`. If `f` is a non-atomic formula, the valuation is computed by applying the corresponding Kleene operator to the recursively computed values of the subformulas of `f`.

As expected, the well-formedness theorem for the valuation operator says it's an element of the type **3**.

```
*T valuation_wf  ∀a:Assignment.∀F:Formula.((F under a)  ∈ 3)
```

4.1.2.4 Satisfaction and Falsification of Formulas

Using the Kleene valuation we define the semantic notion of a formula being satisfied (falsified) by an assignment `a`.

```
*A formula_sat              a |= F   ≝  (F under a)  = T3 ∈ 3
*A formula_falsifiable      a |≠ F   ≝  (F under a)  = F3 ∈ 3
```

Thus, a formula `F` is satisfied by assignment `a` (written `a |= F`) when `(F under a)` evaluates to `T3`. Similarly, a formula `F` is falsified by assignment `a` (written `a |≠ F` when `(F under a)` evaluates to `F3`.

The satisfiability (or falsifiability) of a formula under an assignment is clearly a decidable property; to decide if a formula is satisfied (falsified) by `a`, evaluate `(F under a)` and check whether the result is equal to `T3` (`F3`). This property is captured by the following theorems.

```
*T decidable__formula_sat:
 ∀a:Assignment. ∀F:Formula. Dec{a |= F}
 Extraction:
  λa,F.((λ%1.%1 (F under a)  T3) ext{decidable__equal_Three})
*T decidable__formula_falsifiable:
 ∀a:Assignment. ∀F:Formula. Dec{a |≠ F}
 Extraction:
  λa,F.((λ%1.%1 (F under a)  F3) ext{decidable__equal_Three})
```

The functions extracted from the formal proofs reflect the informal argument just given, *i.e.* they accept as arguments an assignment `a` and a formula `F` and then apply the decision procedure for equality over **3** to the terms `(F under a)` and `T3` (`F3`). The function deciding equality over **3** is referred to by the term `ext{decidable__equal_Three}`, which denotes the extract of the theorem of the same name. Reference to the extract of a previously proved lemma arises by reference to the previously proved lemma in the proof.

Some useful lemmas follow immediately from the definitions of satisfiability and falsifiability. These lemmas relate semantic notions with syntactic structure by characterizing the satisfiability (or falsifiability) of a formula in terms of its subformulas.

```
*T formula_not_sat_lemma
   ∀F:Formula. ∀a:Assignment
       a |= ⌈∼⌉F ⟺ a |≠ F
*T formula_not_falsifiable_lemma
   ∀F:Formula. ∀a:Assignment.
       a |≠ ⌈∼⌉F⟺ a |= F
*T formula_and_sat_lemma
   ∀a:Assignment. ∀q:Formula. ∀r:Formula.
       a |= q⌈∧⌉r ⟺ a |= q  ∧ a |= r
*T formula_and_falsifiable_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |≠ q⌈∧⌉r⟺ a |≠ q ∨ a |≠ r
*T formula_or_sat_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |= q⌈∨⌉r ⟺ a |= q  ∨ a |= r
*T formula_or_falsifiable_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |≠ q⌈∨⌉r⟺ a |≠ q ∧ a |≠ r
```

```
*T formula_imp_sat_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |= q⌈⇒⌉r ⟺ a |≠ q ∨ a |= r
*T formula_imp_falsifiable_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |≠ q⌈⇒⌉r⟺ a |= q  ∧ a |≠ r
```

These lemmas are proved by unfolding the definitions of `formula_sat` and `formula_falsifiable` and then case analysis. These characterizations have been shown to hold in both directions (⟺) but are typically applied as rewrites in a left to right form (⇒).

4.1.2.5 Reconciling classical semantics with Kleene semantics

In this section we show that the use of the semantics based on the three-valued Kleene valuation coincides with the standard two-valued semantics.

Given assignments `a'` and `a`, we define the *restriction* of `a'` to `a`, denoted (`a'↓a`), to be the assignment that is undefined (*i.e.* equal to `?3`) whenever `a` is undefined. The formal definition is as follows.

```
*A restriction
a'↓a  ≝  λx.case (a x): F3 ⟶ (a' x); ?3 ⟶ ?3; T3 ⟶ (a' x);
```

The well-formedness theorem establishes that `a'↓a` is in fact an assignment.

```
*T restriction_wf
∀a,a':Assignment. (a↓a' ∈ Assignment)
```

If the restriction of an assignment `a'` to an assignment `a` is identical with `a`, we say `a'` is an *extension* of `a` or `a'` *extends* `a`. We formalize this notion in the following abstraction.

```
*A extension   a' extends a  $\overset{\text{def}}{=}$  a'↓a = a ∈ Assignment
```

The well-formedness goal for the the definition asserts that it is indeed a proposition.

The following lemma characterizes the notion of extension and verifies the main property of interest; specifically, if `a'` extends `a` then `a'` and `a` agree on every variable for which `a` is defined.

```
*T extension_lemma
∀a,a':Assignment
   a' extends a ⇒
     (∀x:Var. ¬((a x) = ?3 ∈ 3) ⇒ ((a x) = (a' x) ∈ 3))
```

Having formally defined the notion of one assignment being an extension of another, we can state a theorem justifying the Kleene valuation semantics with respect to the standard two-valued semantics.

```
*T assignment_monotone
∀a,a':Assignment
   ∀F:Formula
      a' extends a ⇒
        (a |= F  ⇒ a' |= F ) ∧ (a |≠ F ⇒ a' |≠ F)
```

The lemma is proved by induction on the structure of the formula `F`. It tells us that any assignment extending a satisfying (falsifying) assignment for a formula `F` is also a satisfying (falsifying) assignment of `F`. Extensions of partial assignments not having the value `?3` in their range comprise the standard two-valued assignments, thereby justifying our use of three-valued semantics based on the Kleene operators.

4.1.2.6 Fullness and Validity

Although we are ultimately interested in determining the validity of sequents, introducing the concepts of fullness and validity in relation to formulas first is worthwhile. The definitions presented in this section are not used in the decidability proof itself but do serve to illustrate fullness and validity in the simpler context of formulas. These definitions *are* used to prove a theorem characterizing the relationship between validity of formulas and validity of sequents.

In two-valued semantic presentations, a formula F is said to be *valid* when

$$\forall a : \texttt{Assignment}.\, a \models F.$$

However, under this definition there are no valid sentences of Kleene's three-valued logic. To see why, consider the constant assignment ($\boldsymbol{\lambda}$`x.?3`); examination of the matrices for the Kleene operators shows that *no* formula is *true* under this assignment. Thus, if validity requires a formula to be true under every Kleene valuation, there are no valid formulas. Validity was not at issue for Kleene, who used the logic for reasoning about partial recursive predicates; for us, an acceptable notion of validity is crucial.

Toward this end we will say an assignment is *full* for a formula `F` if the assignment either satisfies `F` or falsifies `F`. For example, let `a` be the assignment that maps variable `x` to the value `T3` and maps all other variables to the undefined value, `?3`. Then (`⌜x⌝ under a`) evaluates to `T3` and so satisfies the formula `⌜x⌝`; `a` is full for the formula `⌜x⌝`. On the other hand, (`⌜x⌝⌜⇒⌝⌜y⌝ under a`) evaluates to (⌜y⌝ `under a`) which in turn valuates to `?3` , and thus `a` is not full for the formula ⌜x⌝⌜⇒⌝⌜y⌝. This notion of fullness allows for consideration of only those assignments that contain "enough information" to completely determine the value of a formula. In the formalization, full assignments are defined as a subtype of `Assignment`.

```
*A full_formula_assignment
    Full(F)  ≝  {a:Assignment| (a |= F  ∨ a |≠ F)}
```

Thus, `Full(F)` is the type of assignments satisfying the fullness predicate for formula `F`.

It was shown above that `formula_sat` and `formula_falsifiable` are decidable properties; hence, for any formula `F` and any assignment `a`, it can be uniformly decided whether `a` is in the type `Full(F)` or not. In general, if the defining predicate of a set type is decidable, we may disregard the restrictions on the use of the properties specified by the defining predicate. The following property lemma is used by the Nuprl decomposition tactics to decompose full assignments.

```
*T full_formula_assignment_properties
  ∀F:Formula. ∀a:Full(F). a |= F  ∨ a |≠ F
```

This lemma is proved by eliminating the outermost quantifiers and then decomposing the type `Full(F)`. This results in the following sequent.

```
 1.  F:Formula
 2.  a: Assignment
 [3]. a |= F  ∨ a |≠ F
 ⊢ a |= F  ∨ a |≠ F
```

Hidden hypotheses are labeled hidden by the square brackets surrounding their hypothesis numbers.

The proof is trivial if hypothesis 3 can be unhidden. To do this we assert its decidability, which results in two subgoals: one to show that the disjunction is in fact decidable; and the second to show the original goal under the additional hypothesis of decidability of the disjunct. The first subgoal is reduced to trivial subgoals by the `ProveDecidable` tactic, which establishes the decidability of the disjunct using a lemma in the library characterizing when disjunctive formulas are decidable (*i.e.* whenever the principal sub-terms are too). The second subgoal generated by asserting `dec{a |= F ∨ a |≠ F}` is discharged by applying the `UnhideHyp` tactic to the hidden hypothesis. This results in a subgoal requiring us to show that the decidable predicate is squash stable which in turn is discharged by applying the `ProveSqStable` tactic, completing the proof of the properties lemma.

Using the definition of fullness just given, we formalize the notion of validity as follows.

`*A formula_valid` $\quad$ `|= F` $\overset{\text{def}}{=}$ `∀a:Full(F). a |= F`

Thus, a formula is valid when it is satisfied by every full assignment. If an assignment contains enough information to determine the truth or falsity of a formula and every such assignment corroborates the truth of the formula then the formula is valid.

4.1.3 Sequents

Sequents are formalized as pairs of lists of formulas; of course, other options are possible, pairs of bags (multi-sets) of formulas chief among them.

```
*A Sequent:  Sequent  $\stackrel{\text{def}}{=}$  Formula list × Formula list
```

Another trivial inclusion lemma is provided for the type checking tactics.

```
*T Sequent_inc:  Sequent ⊆ (Formula list × Formula list)
```

A functional interface is provided for decomposing sequents into the hypothesis and conclusion lists by the `H` and `C` abstractions.

```
*A H:   s.H  $\stackrel{\text{def}}{=}$  let h,c = s in h
*A C:   s.C  $\stackrel{\text{def}}{=}$  let h,c = s in c
```

4.1.3.1 Sequent Rank

Before defining a measure on sequents we define the rank of a list of formulas; it is simply the sum of the ranks of the formulas occurring in the list.

```
*A list_rank    ρ  $\stackrel{\text{def}}{=}$  λL.reduce((λx,y.(ρ(x) + y));0;L)
*T list_rank_wf ρ ∈ (Formula list ⟶ ℕ)
```

This definition uses the `reduce` operator on lists, which accepts three arguments: a two argument function which is associative; an identity for the operator; and a list. Note that we have not distinguished the display forms for the `formula_rank` function (which is used within the definition of the right associative operator) and the `list_rank` function being defined here; they share the same display in the system as well. The same display is used for the `sequent_rank` function defined below. It is clear from the context which operator is being used and, in the system, should it be confusing at any point which operator a display denotes, a click of a mouse button distinguishes them.

A useful property of `list_rank` is that it, in some sense, "distributes" over list append (denoted in infix notation here by @). More formally we say `list_rank` is homomorphic with respect to append and addition. Of course this property can be formulated more abstractly in that any right associative operation (addition in this case) iteratively applied to list is homomorphic with the append operator; the following theorem is a special case of that fact and is useful in the decomposition of ranked lists.

```
*T list_rank_append_homomorphism
   ∀M,N:Formula list. ρ(M @ N) = (ρ(M) + ρ(N)) ∈ ℕ
```

Using the `list_rank` just defined, the rank of a sequent is simply defined to be the sum of the ranks of the hypothesis and conclusion lists.

```
*A sequent_rank     ρ  =def  λS.(ρ(S.H) + ρ(S.C))
*T sequent_rank_wf  ρ ∈ (Sequent → ℕ)
```

We call sequents having rank 0 *atomic* sequents.

4.1.3.2 Sequent satisfiability and falsifiability

In this section the semantics of sequents is given. First, the meaning of a sequent is given in informal mathematical terms and then this definition is translated into the three-valued model being developed here.

A sequent S is of the form $\langle [H_1, H_2, \ldots, H_n], [C_1, C_2, \ldots, C_m] \rangle$, where $[H_1, \ldots, H_n]$ and $[C_1, \ldots, C_m]$ are lists of formulas corresponding to the hypothesis and conclusion respectively. S is interpreted to be true precisely when the conjunction of the hypotheses implies the disjunction of the conclusions.

$$(H_1 \wedge \cdots \wedge H_n) \Rightarrow (C_1 \vee \cdots \vee C_m)$$

Adopting the convention that an empty conjunction denotes truth and the empty disjunction denotes falsity, the sequent $\langle [H_1, \ldots, H_n], [\,] \rangle$ means $\neg H_1 \vee \cdots \vee \neg H_n$, $\langle [\,], [C_1, \ldots, C_m] \rangle$ means $C_1 \vee \cdots \vee C_m$, and the empty sequent, $\langle [\,], [\,] \rangle$, denotes an unsatisfiable sequent.

The discussion above follows the standard presentation for a two-valued semantics of sequents; here however, as for formula above, we are interested in the satisfaction of sequents under Kleene valuations induced by partial assignments. Adapting the discussion above to the analogous definition under Kleene interpretation, which we've already defined for formulas, we find the following.

A partial assignment $\mathtt{a}$ *satisfies* a sequent $\langle [H_1, \ldots, H_n], [C_1, \ldots, C_m] \rangle$ if and only if

$$\mathtt{a} \models (H_1 \lceil \wedge \rceil \cdots \lceil \wedge \rceil H_n) \lceil \Rightarrow \rceil (C_1 \lceil \vee \rceil \cdots \lceil \vee \rceil C_m)$$

where $\models$ is the formula satisfaction relation defined above using the Kleene interpretation induced by $\mathtt{a}$. Similarly, an assignment *falsifies* a sequent of the form

$\langle[H_1,\ldots,H_n],[C_1,\ldots,C_m]\rangle$ if and only if

$$\mathtt{a} \not\models (H_1 \lceil\wedge\rceil \cdots \lceil\wedge\rceil H_n) \lceil\Rightarrow\rceil (C_1 \lceil\vee\rceil \cdots \lceil\vee\rceil C_m)$$

These definitions could be formalized as presented and would serve to define the semantic notions of sequent satisfiability, sequent falsifiability and sequent validity; however, the properties of the operators for conjunction, disjunction and implication under the Kleene valuation suggest a computationally simpler characterization. Under the definition above, a sequent is satisfiable under an assignment `a` either when there is some hypothesis that is falsified by `a` or there is some formula in the conclusion that is satisfied by `a`. This suggests the following definition.

```
*A sequent_satisfiable
a |= S  ≝  ∃F∈S.H.a |≠ F ∨ ∃F∈S.C.a |= F
```

Similarly, a sequent `S` is falsifiable under an assignment `a` if every hypothesis of `S` is satisfied by `a` and every conclusion of `S` is falsified by `a`. Again, this is the formal definition adopted here.

```
*A sequent_falsifiable
a |≠ S  ≝  ∀F∈S.H.a |= F  ∧ ∀F∈S.C.a |≠ F
```

These definitions exhibit the first use of the bounded list quantification operators. The list existence quantifier is non-void (true) if, for any member `x` of the list `L`, the predicate `P[x]` is non-void. Thus, for empty lists it is false. Similarly, the list forall quantifier is true if every `x` in `L` satisfies `P[x]`. For the empty list, the quantifier is vacuously true.

It can effectively be decided whether a sequent is satisfied or falsified by an assignment; this follows from the decidability of the same properties for formulas. These facts are captured in the following two decidability theorems.

```
*T decidable__sequent_satisfiable:
  ∀S:Sequent. ∀a:Assignment. Dec{a |= S}
*T decidable__sequent_falsifiable:
 ∀S:Sequent. ∀a:Assignment. Dec{a |≠ S}
```

4.1.3.3 Full Sequent Assignments and Sequent Validity

Fullness for a formula is now defined for sequents.

```
*A full_sequent_assignment
   Full(S)  ≝  {a:Assignment| (a |= S ∨ a |≠ S)}
```

Again, we provide a trivial sub-typing lemma and a properties lemma for use by the decomposition tactics.

```
*T full_sequent_assignment_inc
   ∀S:Sequent. Full(S) ⊆ Assignment
*T full_sequent_assignment_properties
   ∀S:Sequent. ∀a:Full(S). a |= S ∨ a |≠ S
```

Validity can now be defined with respect to fullness.

```
*A sequent_valid   |= S   ≝  ∀a:Full(S). a |= S
```

If sequent validity has the relationship to formula validity we expect, then for every formula `F`, the sequent ⟨`[],F::[]`⟩ should be valid precisely when `F` itself is. This is captured by the following theorem which is easily proved by unfolding the definitions for sequent validity and formula validity, followed by some steps of computation.

```
*T formula_valid_iff_sequent_valid
   ∀F:Formula. |= F ⟺ |= <[],F::[]>
```

4.2 Decidability

Our goal is to prove decidability of propositional logic, *i.e.* to show that we can decide if a propositional sequent is valid. The most natural formalization of the theorem would simply say

```
     ∀S:Sequent. |= S ∨ ¬(|= S)
```

A proof of this theorem would yield a function accepting a sequent as an argument and then returning an `inl` term or an `inr` term, depending on whether the sequent was valid or not. But we know a proposition is not valid if and only if there is some full assignment which falsifies it.

```
     ∀S:Sequent.¬(|= S) ⟺ (∃a:Assignment. a|≠ S).
```

Using this logically equivalent form of unsatisfiability we state a computationally stronger version of the decidability theorem.

```
 ∀S:Sequent. |= S ∨ (∃a:Assignment. a |≠ S)
```

That is, every sequent is either valid or there exists an assignment which falsifies it. The revised version of the theorem is stronger in the sense that we extract more interesting computational content from its proof. A witness for the theorem is a function of type

```
    S:Sequent ⟶ (|= S  |  a:Assignment × a |≠ S)
```

Thus, a constructive proof of this theorem results in a function accepting a sequent `S` as its argument and returning one of `inl(t)` or `inr(⟨a,e⟩)`. The term `t` under the injection `inl` has little interest for us and so we squash it; however, the first element of the pair ⟨`a,e`⟩ under the `inl` injection is most interesting. It is an assignment falsifying the proposition. This assignment provides diagnostic information telling exactly when the proposition is false. We have formalized the semantics in terms of partial assignments to be able to refine the information content in the falsifying assignment further: depending on the form of the proof, the falsifying assignment returned by the procedure can be minimal in the sense that only those variables contributing to the falsification of the formula are assigned one of `true` or `false`, all others are left `undefined`. Of course, this depends on the form of the proof; it is shown below where the partiality plays a part.

Modifying the statement of the theorem to eliminate the computationally uninteresting parts of the extract results in the following theorem.

```
*THM propositional_decidability
 ∀S:Sequent. ↓(|= S) ∨ {a:Assignment | a |≠ S}
```

In Chapter 5, a proof type is formalized and formal proofs are returned in the case that an intuitionistic sequent is valid; although we have not done it, the same approach could be duplicated here.

4.2.1 A Sequent Proof System for Classical Propositional Logic

Consider the propositional proof system shown in Figure 4.1.

$$\frac{}{M, q, N \;\vdash\; M', q, N'}$$

$$\frac{M,\ N \vdash p,\ \mathit{concl}}{M,\ \lceil\sim\rceil p,\ N \vdash \mathit{concl}} \qquad \frac{p,\ \mathit{hyp} \vdash M,\ N}{\mathit{hyp} \vdash M,\ \lceil\sim\rceil p,\ N}$$

$$\frac{q,\ r,\ M,\ N \vdash \mathit{concl}}{M,\ q\lceil\wedge\rceil r,\ N \vdash \mathit{concl}} \qquad \frac{\mathit{hyp} \vdash q,\ M,\ N \quad \mathit{hyp} \vdash r,\ M,\ N}{\mathit{hyp} \vdash M,\ q\lceil\wedge\rceil r,\ N}$$

$$\frac{q,\ M,\ N \vdash \mathit{concl} \quad r,\ M,\ N \vdash \mathit{concl}}{M,\ q\lceil\vee\rceil r,\ N \vdash \mathit{concl}} \qquad \frac{\mathit{hyp} \vdash q,\ r,\ M,\ N}{\mathit{hyp} \vdash M,\ q\lceil\vee\rceil r,\ N}$$

$$\frac{M,\ N \vdash q,\ \mathit{concl} \quad r,\ M,\ N \vdash \mathit{concl}}{M,\ q\lceil\Rightarrow\rceil r,\ N \vdash \mathit{concl}} \qquad \frac{q,\ \mathit{hyp} \vdash r,\ M,\ N}{\mathit{hyp} \vdash M,\ q\lceil\Rightarrow\rceil r,\ N}$$

Figure 4.1: Proof System for Classical Propositional Logic

Recall that a sound rule preserves validity, *i.e.* the validity of its premises implies the validity of its conclusion. A proof rule is said to be *invertible* when every assignment satisfying the conclusion also satisfies all the premises. For the rules used here, if any premise of an invertible rule is falsified by a given three-valued assignment, then the conclusion is falsified by the same assignment.

Each of the proof rules has been formally shown to be both sound and invertible.

Negation on the left

```
*T formula_not_left_sound
   ∀concl,M,N:Formula list. ∀p:Formula.
     |= <M @ N,p::concl>  ⇒ |= <M @ (⌜∼⌝p::N),concl>
```

```
*T formula_not_left_invertible
   ∀concl,M,N:Formula list. ∀p:Formula. ∀a:Assignment.
      a |≠ <M @ N,p::concl>⟺ a |≠ <M @ (⌜∼⌝p::N),concl>
```

Conjunction on the left

```
*T formula_and_left_sound
    ∀concl,M,N:Formula list. ∀q,r:Formula
        |= <q::r::M @ N,concl>  ⇒ |= <M @ (q⌜∧⌝r::N),concl>
*T formula_and_left_invertible
   ∀concl,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
      a |≠ <q::r::M @ N,concl>⟺ a |≠ <M @ (q⌜∧⌝r::N),concl>
```

Disjunction on the left

```
*T formula_or_left_sound
   ∀concl,M,N:Formula list. ∀q,r:Formula
     |= <q::M @ N,concl>  ⇒ |= <r::M @ N,concl>
     ⇒ |= <M @ (q⌜∨⌝r::N),concl>
*T formula_or_left_invertible
   ∀concl,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment
       a |≠ <q::M @ N,concl> ∨ a |≠ <r::M @ N,concl>
        ⟺ a |≠ <M @ (q⌜∨⌝r::N),concl>
```

Implication on the left

```
*T formula_imp_left_sound
   ∀concl,M,N:Formula list. ∀q,r:Formula
     |= <M @ N,q::concl>  ⇒ |= <r::M @ N,concl>
     ⇒ |= <M @ (q⌜⇒⌝r::N),concl>
*T formula_imp_left_invertible
   ∀concl,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment
      a |≠ <r::M @ N,concl> ∨ a |≠ <M @ N,q::concl>
       ⟺ a |≠ <M @ (q⌜⇒⌝r::N),concl>
```

Negation on the right

```
*T formula_not_right_sound
 ∀hyp,M,N:Formula list. ∀p:Formula.
   |= <p::hyp,M @ N>  ⇒ |= <hyp,M @ (⌈∼⌉p::N)>
*T formula_not_right_invertible
   ∀hyp,M,N:Formula list. ∀p:Formula. ∀a:Assignment.
      a |≠ <p::hyp,M @ N>⟺ a |≠ <hyp,M @ (⌈∼⌉p::N)>
```

Conjunction on the right

```
*T formula_and_right_sound
   ∀hyp,M,N:Formula list. ∀q,r:Formula
     |= <hyp,q::M @ N>  ⇒ |= <hyp,r::M @ N>
    ⇒ |= <hyp,M @ (q⌈∧⌉r::N)>
*T formula_and_right_invertible
   ∀hyp,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment
      a |≠ <hyp,q::M @ N> ∨ a |≠ <hyp,r::M @ N>⇐
      ⇒ a |≠ <hyp,M @ (q⌈∧⌉r::N)>
```

Disjunction on the right

```
*T formula_or_right_sound
   ∀hyp,M,N:Formula list. ∀q,r:Formula
     |= <hyp,q::r::M @ N>  ⇒ |= <hyp,M @ (q⌈∨⌉r::N)>
*T formula_or_right_invertible
   ∀hyp,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
     a |≠ <hyp,q::r::M @ N>⟺ a |≠ <hyp,M @ (q⌈∨⌉r::N)>
```

Implication on the right

```
*T formula_imp_right_sound
   ∀hyp,M,N:Formula list. ∀q,r:Formula
     |= <q::hyp,r::M @ N>  ⇒ |= <hyp,M @ (q⌈⇒⌉r::N)>
```

```
*T formula_imp_right_invertible
   ∀hyp,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
      a |≠ <q::hyp,r::M @ N>⟺ a |≠ <hyp,M @ (q⌈⇒⌉r::N)>
```

It should be remarked here that the propositional proof rules given are the ordinary rules presented by Gentzen [Gen69] or system G' as presented by Gallier in [Gal86b], for example (modulo ordering of formulas in the antecedents of the rule hypotheses). The reader might suspect that the Kleene semantics somehow make the logic special, but the Kleene semantics simply allows for the construction of tighter counter-examples. By defining validity in terms of full assignments, the `assignment_monotone` lemma (presented above) shows that our decidability result applies to ordinary two valued classical logic. Above the layer of abstraction provided by the definitions of satisfaction, falsification, and validity, the effect of the Kleene semantics on the decidability proof and the extracted program is isolated to a single lemma.

The ordering of formulas in the antecedents of hypotheses has no impact on the logic but could affect the efficiency of the search procedure. In our rules we have chosen to move decomposed parts of sequents to the front, hoping to cut down on the complexity of the search for "normal" cases. If the procedure is applied to "almost normal" sequents containing formulas of low complexity, then leaving them in place would be no worse than our ordering. If, on the other hand, sequents are composed of formulas having relatively higher complexity, moving them to the front of the hypothesis and conclusion lists will have the effect of cutting down on the length of the search for formulas of non-zero rank. In either case, worst case search time for the list structure we have is $O(n^2)$.

In the proof presented here, application of the axiom rule (in the form of the lemma `valid_or_falsifiable`) is further restricted to the case when all formulas in the hypothesis and conclusion lists are atomic. This restriction is not required for soundness but is a tradeoff in complexity of the execution time. For the list data-structure we have used here to implement sequents, in the worst case, looking for axioms before getting to a normal sequent could multiply, by a factor of n, the overall complexity of the algorithm.

4.2.2 A strategy for the proof

Soundness and invertibility of the proof rules, coupled with the observation that the backwards application of each rule results in one or two sequents having smaller rank, suggests a recursive procedure for eliminating propositional operators, resulting in a collection of sequents having the following properties:

i.) the induced sequents are all atomic (and atomic sequents are easily decided),

ii.) if all the induced sequents are valid then so is the original sequent (by soundness), and

iii.) if any of the induced sequents is falsified by an assignment then that assignment falsifies the original sequent too (by invertibility).

These observations suggest the existence of a normalization procedure for sequents that produces at each step a collection of sequents having smaller rank; and which preserves joint validity, and preserves the existence of a falsifiable member. By transitivity of implication, the repeated application of the one-step procedure would result in a collection of atomic sequents whose collective validity implies the validity of the goal and, if any are individually falsifiable, then the falsifying assignment falsifies the original sequent too. The existence of such a collection is established by the following lemma.

```
∀G:Sequent
   ∃L:Sequent list
    ∀s∈L.ρ s = 0 ∧
   (∀s∈L.|= s)  ⇒ |= G ∧
   (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ G)
```

The extract from a proof of this lemma returns a list of sequents having the desired properties, but it pairs that list with proofs that the properties are satisfied. These are of no interest to us. The following lemma results in an extract term containing the list of sequents without the accompanying proofs.

```
* THM normalize
 ∀G:Sequent
    {L:Sequent List|
     ↓((∀s∈L. ρ(s) = 0)
      ∧ ((∀s∈L. |= s ) ⇒ |= G )
      ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ G))}
```

The observation that atomic sequents are easily decided is generalized by the following lemma which says that, for every list of atomic (zero rank) sequents, either they are all valid or there is some assignment falsifying some sequent in the list. A natural statement of this fact is as follows.

```
   ∀L:Sequent list.
      (∀s∈L. ρ s = 0)  ⇒
        ∀s∈L.|= s  ∨ (∃a:Assignment. ∃s∈L.a |≠ s)
```

We reformulate the lemma as follows, eliminating non-computational structure from the resulting extract term.

```
* THM valid_or_falsifiable
∀L:{L:Sequent List| ∀s∈L.(ρ(s) = 0)}
 ↓(∀s∈L. |= s) ∨  {a:Assignment| ∃s∈L . a |≠ s}
```

This lemma is the only point in the decidability proof that makes explicit use of the Kleene semantics. In its proof, if there is a sequent `s` in the list `L` having a disjoint antecedent and succedent, a decision must be made as to which values to assign to variables not occurring in an atomic sequent. Rather than arbitrarily choosing true (`T3`) or false (`F3`), as we would do in a two-valued semantics, under the Kleene semantics we assign the "undefined" value (`?3`), resulting in a tighter counter-example.

4.2.3 Decidability proof

We present highlights of the Nuprl proof of decidability.

```
⊢ ∀S:Sequent. ↓(|= S) ∨ {a:Assignment| a |≠ S}
```

Decomposing the universal and instantiating the normalization lemma with `S` as the goal results in the following Nuprl sequent.

```
 1. S: Sequent
 2. L: Sequent list
[3.] ↓((∀s∈L.ρ(s) = 0) ∧
       (∀s∈L.|= s) ⇒ |= S ∧
       ∀a:Assignment. (∃s∈L.a |≠ s) ⇒ a |≠ S)
 ⊢ ↓(|= S) ∨ {a:Assignment| a |≠ S}
```

Instantiating the lemma `valid_or_falsifiable` with `L` leaves a disjunction asserting that either all elements of `L` are valid or some element of `L` is falsifiable. Decomposing this disjunction leaves two subgoals. In the first case we know all sequents in `L` are valid and so choose to prove the first disjunct of the conclusion. In the second case we have an assignment that falsifies some sequent in `L` and so choose to prove the second disjunct of the main goal in that case.

Consider the first case.

```
[3.] ↓((∀s∈L.𝝆(s) = 0) ∧
       (∀s∈L.|= s) ⇒ |= S ∧
       ∀a:Assignment. (∃s∈L.a |≠ s) ⇒ a |≠ S)
 4. ↓(∀s∈L. |= s)
 ⊢ ↓(|= S)
```

Because the conclusion is squashed, the hidden hypothesis (3) can be freely unhidden. Eliminating the squash operators and then decomposing the conjuncts in 3 results in the following:

```
 3. ∀s∈L.𝝆(s) = 0
 4. (∀s∈L.|= s) ⇒ |= S
 5. ∀a:Assignment. (∃s∈L.a |≠ s) ⇒ a |≠ S
 6. ∀s∈L. |= s
 ⊢ |= S
```

Backchaining through hypothesis 4 combined with the fact stated in 6 completes the proof of this branch.

Now consider the second case.

```
 4. {a:Assignment| ∃s∈L. a |≠ s}
  ⊢ {a:Assignment| a |≠ S}
```

After decomposing the conjunction in hypothesis 3 (see above) and then decomposing the set type in hypothesis 4, we provide the resulting assignment as the witness for the set type in the conclusion. This yields the following subgoal.

```
3. ∀s∈L.ρ(s) = 0
4. (∀s∈L.|= s)  ⇒ |= S
5. ∀a:Assignment. (∃s∈L.a |≠ s) ⇒ a |≠ S
6. a:Assignment
7. ∃s∈L. a |≠ s
 ⊢ a |≠ S
```

The hidden hypotheses have automatically been unhidden by the system because the computational content of the proof has been completed at this point. The remaining goal is proved by appeal to facts in hypotheses 5 and 7. This completes the proof.

The program extracted from this proof (after one step of reduction) is the following term.

```
λS.decide ext{valid_or_falsifiable}(ext{normalize}(S))
      of inl(%3) => inl(Axiom)
       | inr(%4) => inr(%4)
```

It accepts a sequent `S` as input and applies the normalization procedure to it. The result is a list of zero rank sequents serving as input to `valid_or_falsifiable`. This returns a term of the form `inl(Ax)` or `inr(a)`, where `a` is a partial assignment falsifying some element of `L` (and which by extension falsifies `S`). A case split is made on the form of this term, which is then packaged up and returned as the final result of the procedure. Thus, we see that this program is nearly the natural one to write given the procedures `ext{valid_or_falsifiable}` and `ext{normalize}`. A simple optimization results in the following simpler program which foregoes the redundant `decide`.

```
λS. ext{valid_or_falsifiable}(ext{normalize}(S))
```

4.2.4 Deciding atomic sequents

Recall the statement of the lemma asserting that collections of atomic sequents are either all valid or there is some assignment falsifying at least one of them.

```
* THM valid_or_falsifiable
∀L:{L:Sequent List| ∀s∈L.(ρ(s) = 0)}
 ↓(∀s∈L. |= s) ∨  {a:Assignment| ∃s∈L . a |≠ s}
```

The proof rests on the observation that a sequent containing only atomic formulas is falsifiable if and only if the hypotheses and the conclusions are disjoint. If they share a hypothesis and conclusion in common, it is an instance of an axiom. If not, the assignment that assigns true to all the variables in the hypothesis and false to all the variables in the conclusion falsifies the sequent.

Thus, the proof idea is to search `L` to see if it is composed completely of sequents that are instances of axioms. If not, then there is a disjoint sequent that serves as the basis for a falsifying assignment. The following lemma specifies this search as property of sequent lists.

```
* THM all_intersect_or_exists_disjoint
∀L:Sequent List
  ↓(∀s∈L.{F:Formula| F∈s.H ∧ F∈s.C})
    ∨ {s:Sequent| s∈L ∧ disjoint(=₂;s.H;s.C)}
```

Note that the lemma holds for all sequent lists, not necessarily only those composed of zero-rank sequents. The property of lists being disjoint is decidable if the type of the list elements is discrete. Thus the proof of this lemma is split into two cases, *i.e.* depending on whether every sequent in L is disjoint or not.

The following term was extracted from the proof of valid_or_falsifiable. It shows how the extract of all_intersect_or_exists_disjoint is used as a sub-program.

```
λL. decide ext{all_intersect_or_exists_disjoint}(L)
     of inl(_) => inl(Ax)
      | inr(s) => inr(λv.if (⌈v⌉∈ s.H) then T3
                          else if (⌈v⌉∈ s.C) then F3
                               else ?3
                               fi
                          fi)
```

Thus, the result of applying ext{all_intersect_or_exists_disjoint} to L is either inl(Ax) or inr(s) where s is a disjoint sequent. The extract shows how the sequent is used in the proof of valid_or_falsifiable to construct a falsifying assignment for s. The falsifying assignment is partial; it only assigns values to variables occurring in the disjoint sequent. Since not every variable occurring in the original goal sequent need be in the leaves of the derivation tree, not every variable of the goal contributes to a falsifying assignment. The end result is a tighter counter-example returned by the procedure.

Before moving to the proof of the normalization lemma some remarks about this proof and its extract are in order. The first thing to notice is that this is the only place in the decidability proof where an equality on sequents is required, but it does not contribute to the computational content. The discrete equality on formulas is used here to search for the atomic formula ⌜x⌝ that is a member of both the hypothesis and conclusion lists.

The assignment chosen to falsify the sequent, having non-disjoint hypotheses and conclusions, is a partial assignment. Indeed, it only assigns values to variables occurring in the non-disjoint sequent. Since not every variable occurring in a sequent contributes content to a falsifying assignment the partial assignment used here provides more information; it not only makes assignment of false and true to variables, but also shows which variables do not affect the falsification of the sequent by the particular assignment returned. In fact, the proof goes through essentially unchanged when the following total assignment is used.

```
λv.if (⌜v⌝(∈eqF) x.H) then T3 else F3 fi
```

The development of a three-valued type and the Kleene operators presented here with the Kleene valuation is more complex, but the end result is tighter information on the falsifying assignment returned by the procedure since we can identify a class of “don’t care” variables in the counter-example.

4.2.5 The Normalization Proof

Decidability rests on the properties of the list L whose existence is established by the normalization lemma. The proof of this lemma provides the core of the computational procedure. The proof is by measure induction and so yields a recursive procedure that maps a single sequent `G` to a list of atomic sequents which collectively validate `G` or at least one of which falsifies `G`.

The proof of this lemma provides the core of the computational procedure. The proof is by induction on the rank of a sequent. Recall the statement of the lemma.

```
⊢ ∀G:Sequent
  {L:Sequent List|
   ↓((∀s∈L. ρ(s) = 0)
    ∧ ((∀s∈L. |= s ) ⇒ |= G )
    ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ G))}
```

The proof is by induction on the rank of a sequent; accordingly, the measure induction tactic is invoked with the measure function `sequent_rank`. Decomposing `G` into its component formula lists, `hyp` and `concl`, results in the following subgoal.

```
  1. hyp: Formula List
  2. concl: Formula List
  3. IH: ∀k:{k:Sequent| ρ(k) < ρ(<hyp, concl>)}
          {L:Sequent List|
           ↓((∀s∈L. ρ(s) = 0)
            ∧ ((∀s∈L. |= s ) ⇒ |= k )
            ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ k))}
⊢ {L:Sequent List|
    ↓((∀s∈L. ρ(s) = 0)
    ∧ ((∀s∈L. |= s ) ⇒ |= <hyp, concl>)
    ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ <hyp, concl>))}
```

The proof proceeds by inductively decomposing non-zero rank elements of the sequent `<hyp,concl>` if there are any; if not, we directly argue the theorem holds.

Thus, to proceed with the proof we case split on whether the list `hyp` contains any non-zero rank formula. In the case where all formulas in `hyp` are atomic we do a case split on whether `concl` is atomic or not. Thus, in all, we have three cases. We consider this last case first.

The sequent is atomic: In this case the list `<hyp,concl>::[]` witnesses the set type. A step of reduction leaves the following squashed conjunction to prove.

```
 4. ¬∃f∈hyp.ρ(f) > 0
 5. ¬∃f∈concl.ρ(f) > 0
⊢ ↓((∀s∈(<hyp,concl>::[]). ρ(s) = 0)
    ∧ ((∀s∈(<hyp,concl>::[]). |= s ) ⇒ |= <hyp, concl>)
    ∧ (∀a:Assignment. (∃s∈(<hyp,concl>::[]). a |≠ s)
                ⇒ a |≠ <hyp, concl>))
```

By hypotheses 4 and 5, the first conjunct holds and the remaining two conjuncts are trivial.

The hypotheses contain a non-atomic formula: Now we consider the case where the formula list `hyp` contains a non-zero rank formula, `∃f∈hyp.(ρ(f)>0)`.

Whenever property (`P`) is asserted to hold for some element of a list `L`, we use the following lemma to decompose the list, explicitly naming an element of the list having the property.

```
* THM list_exists_decomposition
∀T:𝕌. ∀P:T ⟶ ℙ. ∀L:T List
  (∃x∈L.P[x])⇒∃M:T List.∃x:T.{N:T List| L = M @ (x::N) ∧ P[x]}
```

Forward chaining through this lemma with hypothesis `∃f∈hyp.(ρ(f)>0)` yields

```
 4. ∃f∈hyp.(ρ(f) > 0)
 5. M: Formula List
 6. f: Formula
 7. N: Formula List
[8]. hyp = M @ (f::N) ∧ ρ(f) > 0
⊢ {L:Sequent List|
    ↓((∀s∈L. ρ(s) = 0)
    ∧ ((∀s∈L. |= s ) ⇒ |= <hyp, concl>)
    ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ <hyp, concl>))}
```

Now we have a name (`f`) for the non-zero rank formula occurring in `hyp`. Using `f` we provide the following term as a witness for the set type in the conclusion.

```
case f:
 ⌜x⌝ ⟶ [];
 ⌜¬⌝x ⟶ (IH(<M @ N, x::concl>));
 x1⌜∧⌝x2 ⟶ (IH(<x1::x2::(M @ N), concl>));
 x1⌜∨⌝x2 ⟶ (IH(<x1::(M @ N), concl>) @ IH(<x2::(M @ N), concl>));
 x1⌜⇒⌝x2 ⟶ (IH(<x2::(M @ N), concl>) @ IH(<M @ N, x1::concl>));
```

This term encodes the left rules of the sequent proof system presented above. This step results in two subgoals: the first a well-formedness goal to show that the term is in the type `Sequent List`, which is easily shown by case analysis on `f` and then reduction; the second to show that term satisfies the three-part conjunction defining the set. After this step, the computational content for this branch of the proof is complete.

The remaining proof obligations are to verify that the witness term satisfies the logical part of the specification. There are subgoals for negation, conjunction, disjunction and implication occurring on the left (the case in which `f` is a variable is trivially discharged by the auto tactic).

Negation on the left: This is the case where `f` is a negation of the form ⌜∼⌝`x`.

```
8. x: Formula
9. hyp = M @ (⌜∼⌝x::N)
10. ρ x + 1 > 0
⊢ ↓(∀s∈(IH <M @ N, x::concl>).(ρ s = 0)
   ∧ (∀s∈(IH <M @ N, x::concl>).|= s ⇒ |= <hyp, concl>)
   ∧ (∀a:Assignment.
        ∃s∈(IH <M @ N, x::concl>).a |≠ s ⇒ a |≠ <hyp, concl>))
```

We decompose the induction hypothesis with ⌜`<M @ N, x::concl>`⌝, resulting in the following subgoal.

```
⊢ <M @ N, x::concl> ∈ {k:Sequent| ρ k < ρ <M @ (⌜¬⌝x::N), concl>}
```

This well-formedness subgoal is discharged by direct computation and application of the SupInf arithmetic decision procedure. By inspecting the witness term the reader can satisfy himself that the rank of every sequent occurring as an argument to the induction hypothesis `IH` in the witness term is strictly smaller than the rank of the sequent it normalizes and so we will not present these well-formedness goals again.

Two subgoals remain, the first to show that `<M @ (`⌜∼⌝`x::N), concl>` is valid under the assumption that every sequent in the list computed by the application of the induction hypothesis is, *i.e.* to show:

```
12. ∀s∈(IH <M @ N, x::concl>).(ρ s = 0)
13. ∀s∈(IH <M @ N, x::concl>).|= s  ⇒ |= <M @ N, x::concl>
14. ∀a:Assignment.
        ∃s∈(IH <M @ N, x::concl>).a |≠ s ⇒ a |≠ <M @ N, x::concl>
```

```
15. ∀s∈(IH <M @ N, x::concl>).|= s
⊢ |= <M @ (⌈∼⌉x::N), concl>
```

This branch is discharged by appeal to soundness of negation on the left (lemma `formula_not_left_sound`) and then by backchaining through the hypotheses.

The final remaining subgoal for the case of negation on the left is to show that there is an assignment falsifying `<M @ (⌈∼⌉x::N), concl>` if there is an assignment falsifying some element of the list computed by the application of the induction hypothesis.

```
12. ∀s∈(IH <M @ N, x::concl>).(ρ s = 0)
13. ∀s∈(IH <M @ N, x::concl>).|= s  ⇒ |= <M @ N, x::concl>
14. ∀a:Assignment.
      ∃s∈(IH <M @ N, x::concl>).a |≠ s ⇒ a |≠ <M @ N, x::concl>
15. a: Assignment
16. ∃s∈(IH <M @ N, x::concl>).a |≠ s
⊢ a |≠ <M @ (⌈∼⌉x::N), concl>
```

This subgoal is proved by backchaining through the invertibility lemma for negation on the left and then backchaining through the hypotheses.

The case of a negation on the left required one application of the induction hypothesis. We list the other proof obligations for operators on the left without proof. Below, we examine the case for conjunction on the right in some detail to show a case requiring two applications of the induction hypothesis.

Conjunction on the left:

```
8. x1: Formula
9. x2: Formula
10. hyp = M @ (x1⌈∧⌉x2::N)
11. (ρ x1 + ρ x2) + 1 > 0
⊢ ↓(∀s∈(IH <x1::x2::(M @ N), concl>).(ρ s = 0)
  ∧ (∀s∈(IH <x1::x2::(M @ N), concl>).|= s  ⇒ |= <hyp, concl>)
  ∧ (∀a:Assignment.
        ∃s∈(IH <x1::x2::(M @ N), concl>)
          a |≠ s ⇒ a |≠ <hyp, concl>))
```

Disjunction on the left:

```
10. hyp = M @ (x1⌈∧⌉x2::N)
11. (ρ x1 + ρ x2) + 1 > 0
⊢ ↓(∀s∈(IH <x1::x2::(M @ N), concl>).(ρ s = 0)
  ∧ (∀s∈(IH <x1::x2::(M @ N), concl>).|= s  ⇒ |= <hyp, concl>)
  ∧ (∀a:Assignment
    ∃s∈(IH <x1::x2::(M @ N), concl>)
       a |≠ s ⇒ a |≠ <hyp, concl>))
```

Implication on the left:

```
10. hyp = M @ (x1⌈⇒⌉x2::N)
11. (ρ x1 + ρ x2) + 1 > 0
⊢ ↓(∀s∈(IH <x2::(M @ N), concl> @ IH <M @ N, x1::concl>)
           (ρ s = 0)
  ∧ (∀s∈(IH <x2::(M @ N), concl> @ IH <M @ N, x1::concl>).
           |= s ⇒ |= <hyp, concl>)
  ∧ (∀a:Assignment
        ∃s∈(IH <x2::(M @ N), concl> @ IH <M @ N, x1::concl>).
                a |≠ s ⇒ a |≠ <hyp, concl>))
```

The conclusion contains a non-atomic formula: If the list of hypotheses is empty or contains only atomic formulas we consider whether the conclusion list contains any non-atomic formula. In this case, we suppose it does. Like the case above, we instantiate and decompose the lemma `list_exists_decomposition` to get a name for the non-atomic formula occurring in the list. After that we must prove the following:

```
 4. ∃f∈concl.(ρ(f) > 0)
 5. M: Formula List
 6. f: Formula
 7. N: Formula List
[8]. concl = M @ (f::N) ∈ Formula List ∧ ρ(f) > 0
⊢ {L:Sequent List|
    ↓((∀s∈L. ρ(s) = 0)
    ∧ ((∀s∈L. |= s ) ⇒ |= <hyp, concl>)
    ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ <hyp, concl>))}
```

In this case the witness term for the set type in the conclusion is the following.

```
case f:
 ⌜x⌝ ⟶ [];
 ⌜¬⌝x ⟶ (IH(<x::hyp, M @ N>));
 x1⌜∧⌝x2 ⟶ (IH(<hyp, x1::(M @ N)>) @ IH(<hyp, x2::(M @ N)>));
 x1⌜∨⌝x2 ⟶ (IH(<hyp, x1::x2::(M @ N)>));
 x1⌜⇒⌝x2 ⟶ (IH(<x1::hyp, x2::(M @ N)>));
```

As above, the proof of the well-formedness goal asserting that this term denoted a sequent list is straightforward. We are left with the following four proof obligations.

Negation on the right:

```
8. x: Formula
9. concl = M @ (⌈∼⌉x::N)
10. ρ x + 1 > 0
⊢ ↓(∀s∈(IH <x::hyp, M @ N>).(ρ s = 0)
  ∧ (∀s∈(IH <x::hyp, M @ N>).|= s  ⇒ |= <hyp, concl>)
  ∧ (∀a:Assignment
        ∃s∈(IH <x::hyp, M @ N>).a |≠ s ⇒ a |≠ <hyp, concl>))
```

Conjunction on the right:

```
8. x1: Formula
9. x2: Formula
10. concl = M @ (x1⌈∧⌉x2::N)
11. (ρ x1 + ρ x2) + 1 > 0
⊢ ↓(∀s∈(IH <hyp, x1::(M @ N)> @ IH <hyp, x2::(M @ N)>)
              (ρ s = 0)
  ∧ (∀s∈(IH <hyp, x1::(M @ N)> @ IH <hyp, x2::(M @ N)>)
              |= s  ⇒ |= <hyp, concl>)
  ∧ (∀a:Assignment
       ∃s∈(IH <hyp, x1::(M @ N)> @ IH <hyp, x2::(M @ N)>)
              a |≠ s ⇒ a |≠ <hyp, concl>))
```

Disjunction on the right:

```
10. concl = M @ (x1⌈∨⌉x2::N)
11. (ρ x1 + ρ x2) + 1 > 0
⊢ ↓(∀s∈(IH <hyp, x1::x2::(M @ N)>).(ρ s = 0)
  ∧ (∀s∈(IH <hyp, x1::x2::(M @ N)>).|= s  ⇒ |= <hyp, concl>)
  ∧ (∀a:Assignment
        ∃s∈(IH <hyp, x1::x2::(M @ N)>)
           a |≠ s ⇒ a |≠ <hyp, concl>))
```

Implication on the right:

```
10. concl = M @ (x1⌈⇒⌉x2::N)
11. (ρ x1 + ρ x2) + 1 > 0
⊢ ↓(∀s∈(IH <x1::hyp, x2::(M @ N)>).(ρ s = 0)
   ∧ (∀s∈(IH <x1::hyp, x2::(M @ N)>).|= s ⇒ |= <hyp, concl>)
   ∧ (∀a:Assignment
         ∃s∈(IH <x1::hyp, x2::(M @ N)>)
            a |≠ s ⇒ a |≠ <hyp, concl>))
```

These cases are discharged by instantiating the induction hypothesis with the appropriate sequent(s) (depending on the propositional proof rule for that case) and then appealing to the corresponding soundness and invertibility lemmas.

4.3 Analysis of the Normalization Proof and Extract

The extract of the proof of normalization appears as in Figure 4.2. The extracted program is essentially the functional program written to follow the tableau algorithm. It closely matches the pseudo-code presented in Gallier's textbook [Gal86a].

In contrast, the version of the proof not using the set type in its specification and not using the derived measure induction principle is virtually unreadable, being larger by a factor of about five. More precisely, after eliminating all newlines and compressing remaining whitespace to a single space, the ratio is closer to 6 : 1, and with all whitespace removed the ratio is similar. Measuring the number of terms in each raw extract, we get a ratio of 5 : 1 and the ratio of the number of subterms after an application of `Reduce` to each term is the same.

The measure induction eliminates all reference to the measure from the extract. This is a crucial savings in the computational efficiency of the resulting program, especially since the measure function plays no part in the actual computation.

The reader will note that the proof of normalization given above is not pure; however, it does provide for a compact presentation. In this proof, the witnesses compute the lists of normalized sequents directly. The corresponding pure proof proceeds instead by a step of case analysis on the formula `f`, establishing the context for instantiating the induction hypothesis with the hypothesis sequent(s) specified by the propositional rules, and then constructing the appropriate list to discharge the conclusion from these parts.

4.3.1 Applications

In [Gal86b, chapter 3] Gallier presents a search procedure nearly identical to the one extracted from the normalization lemma proved here. He notes two interesting applications of the procedure.

Recall that a proposition is in *conjunctive normal form* (CNF) if it is a conjunction $C_1 \wedge \cdots \wedge C_n$ where each C_i is a disjunction of propositional variables or their negations. A proposition is in *disjunctive normal form* (DNF) if it is a disjunction $D_1 \vee \cdots \vee D_n$ where each D_i is a conjunction of propositional variables or their negations.

Given a proposition P, the normalization procedure can be used to construct logically equivalent propositions P_C and P_D where P_C is in CNF and P_D is in DNF.

To construct the CNF form P_C, simply compute the normalization of the sequent `<[],`P`>`. For each sequent S_i (of the form `<[`$h_1, \cdots, h_n$`], [`$c_1, \cdots c_m$`]>`) in the list of normalized sequents, take C_i to be $\neg h_1 \vee \cdots \vee \neg h_n \vee c_1 \vee \cdots \vee c_m$. The proof that $P \Leftrightarrow P_C$ is quite easy, by using the definition of validity and the properties ensured by the normalization procedure to show the two forms agree on all assignments.

Similarly, to construct the DNF form P_D simply compute the normalization of the sequent `<`P`,[]>`. For each sequent S_i (of the form `<[`$h_1, \cdots, h_n$`], [`$c_1, \cdots c_m$`]>`) in the list of normalized sequents, take D_i to be $h_1 \wedge \cdots \wedge h_n \wedge \neg c_1 \wedge \cdots \wedge \neg c_m$. The proof that $P \Leftrightarrow P_D$ is similar to the proof for the CNF form.

The proof formalized and presented above has been used at Cornell to teach propositional logic to upper level undergraduate students taking CS 472. Students have reported that they found the formalized Nuprl presentation clearer than that of the classic text used for the course by Smullyan [Smu68] Students find the interactive access to all the definitions very helpful in learning the material; if they forget what an operator does they can simply click on it and its definition is displayed. The exposition here makes explicit certain algorithms, for example the computation of sequent rank, that are implicit in the definitions given by Smullyan. The libraries for these theories are available on the Nuprl web-page as well.

```
λG.(letrec normalize(S) =
   let <hyp,concl> = S in
     case ∃f∈hyp.(ρ(f) > 0)
     of inl(%2) =>
          let M,f@0,N = (ext{list_exists_decomposition}
                          (Formula)(λ₂f.ρ(f) > 0)(hyp)(%2)) in
           case f@0:
             ⌈x⌉ ⟶ [];
             ⌈∼⌉x ⟶ (normalize(<M @ N, x::concl>));
             x1⌈∧⌉x2 ⟶ (normalize(<x1::x2::(M @ N), concl>));
             x1⌈∨⌉x2 ⟶ (normalize(<x1::(M @ N), concl>)
                            @ normalize(<x2::(M @ N), concl>));
             y1⌈⇒⌉y2 ⟶ (normalize(<y2::(M @ N), concl>)
                            @ normalize(<M @ N, y1::concl>));
       | inr(%3) =>
           case ∃f∈concl.(ρ(f) > 0)
           of inl(%5) =>
                let M,f@0,N = (ext{list_exists_decomposition}
                                (Formula)(λ₂f.ρ(f) > 0)(concl)(%5)) in
               case f@0:
                 ⌈x⌉ ⟶ [];
                 ⌈∼⌉x ⟶ (normalize(<x::hyp, M @ N>));
                 x1⌈∧⌉x2 ⟶ (normalize(<hyp, x1::(M @ N)>)
                                @ normalize(<hyp, x2::(M @ N)>));
                 x1⌈∨⌉x2 ⟶ (normalize(<hyp, x1::x2::(M @ N)>));
                 y1⌈⇒⌉y2 ⟶ (normalize(<y1::hyp, y2::(M @ N)>));
           | inr(%6) => <hyp, concl>::[]  )
      (G)
```

Figure 4.2: Extract of the Normalization Lemma

Chapter 5

Decidability of Intuitionistic Propositional Logic

5.1 Introduction

In this chapter a formal theory of the intuitionistic propositional calculus is described, including a formal proof of decidability in Nuprl. With the proof of decidability as our focus, we describe formal developments of: a proof theory; the tableau construction; and a theory supporting the use of Kripke counter-examples as evidence of unprovability. The development is based on Underwood's proof [Und93, Und94] and closely follows the presentation by Aitken, Constable, and Underwood in [ACU]. The program resulting from the proof is an implementation of an intuitionistic tableau algorithm.

5.1.1 Intuitionistic proof systems

Sequent proof systems for classical and intuitionistic logic were first presented by Gentzen [Gen69]. Somewhat surprisingly, restricting sequents in the proof rules to having at most one formula in the succedent is enough to move from classical to intuitionistic logic. Many intuitionistic sequent proof systems have been proposed since, including a number of multi-conclusion calculi [Kle52, Dum77, Dra87, Wal90]. Multi-conclusion calculi are closely related to tableau systems, and in [Avr93] Avron characterizes the relation.

The calculus *MJ* presented in Figure 5.1 is essentially the propositional fragment of Dragalin's [Dra87, pg.11] multi-conclusion sequent calculus. The form of our rules differs from Dragalin's in two ways:

i.) The rules presented here are stated so as to allow the active formula (the formula being eliminated or introduced) to occur anywhere in the antecedent or consequent of the sequent. Dragalin stipulates this, but it is not reflected in his rules.

ii.) We do not eliminate the active formula from the hypotheses (except in the $\Rightarrow r$ rule where it is required for soundness). This simplifies the formal definition of proofs and is justified by Theorem 3.1.5 [Dra87, pg.13] which says the contraction rule is admissible.

Intuitionistic propositional logic differs from classical propositional logic in that undirected application of the proof rules is not guaranteed to terminate. This issue does not arise in the classical case, since every application of a rule eliminates

$$\frac{}{M, \lceil\texttt{false}\rceil, N \vdash C}\ (\texttt{false}\,l)$$

$$\frac{}{M, a, N \vdash M', a, N'}\ (Ax)$$

$$\frac{H \vdash a, M, a\lceil\vee\rceil b, N}{H \vdash M, a\lceil\vee\rceil b, N}\ (\vee r1)$$

$$\frac{a, M, a\lceil\vee\rceil b, N \vdash C \quad b, M, a\lceil\vee\rceil b, N \vdash C}{M, a\lceil\vee\rceil b, N \vdash C}\ (\vee l)$$

$$\frac{H \vdash b, M, a\lceil\vee\rceil b, N}{H \vdash M, a\lceil\vee\rceil b, N}\ (\vee r2)$$

$$\frac{a, M, a\lceil\wedge\rceil b, N \vdash C}{M, a\lceil\wedge\rceil b, N \vdash C}\ (\wedge l1)$$

$$\frac{H \vdash a, M, a\lceil\wedge\rceil b, N \quad H \vdash b, M, a\lceil\wedge\rceil b, N}{H \vdash M, a\lceil\wedge\rceil b, N}\ (\wedge r)$$

$$\frac{b, M, a\lceil\wedge\rceil b, N \vdash C}{M, a\lceil\wedge\rceil b, N \vdash C}\ (\wedge l2)$$

$$\frac{M, a\lceil\Rightarrow\rceil b, N \vdash a, C \quad b, M, a\lceil\Rightarrow\rceil b, N \vdash C}{M, a\lceil\Rightarrow\rceil b, N \vdash C}\ (\Rightarrow l)$$

$$\frac{a, H \vdash r}{H \vdash M, a\lceil\Rightarrow\rceil b, N}\ (\Rightarrow r)$$

Figure 5.1: System MJ

a formula; it is easy to see that the repeated application of the rules will terminate. This complexity is revealed in the proof rule for an implication occurring on the left side of a sequent; the implication itself cannot be deleted from from the hypotheses of the rule premises; there are intuitionistically valid propositions that depend on the reuse of the formula. To observe this phenomenon, consider the proof (shown in Figure 5.2) of the formula $((P \vee (P \Rightarrow \bot)) \Rightarrow \bot) \Rightarrow \bot$; this is a proof of $\neg\neg(P \vee \neg P)$ under the standard encoding of $\neg P$ as $P \Rightarrow \bot$. The proof requires two instances of the $\Rightarrow r$ rule applied to the same implication on the left side. Without the duplicated application, the proof cannot be completed. Thus, both the sequent proof rule for implication on the left, and the tableau rule for an implication assumed to be true, have a built in contraction. This implicit contraction in the $\Rightarrow r$ rule complicates termination arguments.

As early as 1952, Vorob'ev [Vor52, Vor70] considered intuitionistic propositional calculi having proof rules that naturally terminate, thereby simplifying the termination argument somewhat. More recently, Hudelmaier [Hud92], Lincoln, Scedrov, and Shankar [LSS91] and Dyckhoff [Dyc92] have all independently rediscovered this idea in contexts of various propositional calculi that suffer the same problem of having implicit contractions. These are the so-called contraction-free proof systems. They work by considering the structure of the antecedent of implications occurring on the left and by delaying application of some of the rules until as late as possible in the proof. Recently, Weich [Wei98a], has formalized a decidability proof for the implicational fragment of Hudelmaier's calculus for intuitionistic propositional logic.

5.1.2 The Tableau Construction

In his 1990 book [Wal90] Wallen remarked that there had been few efforts to automate proof search for intuitionistic logic. That can hardly be said to be true today; indeed, Wallen's book helped to spur significant interest in the area, both in matrix based methods and also in tableau methods. Tableau methods have received significant attention in the recent literature; indeed, since 1995 there has been a series of conferences in Europe devoted specifically to these methods [BHP95, MMMO96, Gal97, dS98], decision procedures for the propositional case being among those receiving attention. Tableau methods for proof search in intuitionistic logic go back to Beth [Bet59]. Fitting's book [Fit83], unfortunately out of print, is the classic reference.

Roughly speaking, tableau methods are those proof search methods that work by systematically exploring all consequences of an assumption. A tableau is a tree-like structure that records the development of the search, keeping track of those formulas assumed to true and those formulas assumed to be false. If we start by assuming that the formula we wish to verify is false, the tableau construction becomes a search for counter-examples. If a counter-example is found, then the formula initially assumed false must be provable.

To decide a formula ϕ, we start the tableau construction with a system containing a single node in which the formula ϕ is assumed to be false. Guided by the formula structure and contents of the nodes in the system, and justified by the Kripkian interpretation of the intuitionistic operators, the tableau construction proceeds by either: extending an existing node by adding new formulas to it;

splitting an existing node by extending it in two different ways; or extending the tableau system by adding a new node. The latter only occurs when the tableau rule is applied to an implication assumed to be false. In this way, all possible consequences are developed. If it occurs that a formula is assumed to be both *false* and *true* along some path in the developing tableau, then that path leads to a contradiction and is *closed*. If a path is developed to the point where further application of the tableau rules would only result in redundant structure being added to the tableau, then we stop development along that path. We call these paths *open*. If all the paths through the tableau for ϕ are closed then ϕ is provable; *i.e.* if assuming ϕ is false always leads to a contradiction, then there must be a proof of ϕ. Using the tableau constructed in this way we are able to construct a proof of ϕ

It is easy to check whether a path is closed. The complexity of the procedure arises in determining when further development of an open path is redundant. Underwood's proof [Und94], formalized for the first time here, provides a new termination argument based on a lexicographic ordering of tableau systems bounding the number of formulas that can be added to any node and bounding the number of nodes that can be added to a system.

The classical theory presented in Chapter 4 is a sub-theory of the one presented here. In the classical tableau construction, since every rule is local, a system would always contain a single node and would never be extended.

5.1.3 Kripke Counter-examples as Evidence of Unprovability

In the proof of decidability for the classical case given in Chapter 4, the representation of counter-examples was simply accomplished by considering three-valued assignments. Although not as fine, Boolean valuations would have also served that purpose. In the intuitionistic case, it is a well known negative result that no finite valuation captures intuitionistic propositional logic [Dra87]. Thus, models for intuitionistic logic are necessarily more complex. Following the account given by Underwood in [Und90, Und94], we use Kripke models to witness the unprovability of a formula. This interpretation is not without some subtlety which we address below, but first we introduce the models.

In 1963 Kripke [Kri63] presented graph-based model construction for modal logic. Gödel had already shown a modal interpretation of intuitionistic propositional logic: based on Gödel's interpretation, Kripke was able to extend his construction to the intuitionistic case [Kri65].

A Kripke model $\mathcal{K}$ is a triple of the form $(\Sigma, \leq, \Vdash_a)$ where: Σ is a (non-empty) set of states; $\leq$ is a reflexive and transitive relation on the states; and $\Vdash_a$, the atomic forcing relation, is a decidable relation on $\Sigma \times \texttt{Var}$ characterizing the variables forced (true) in a state of $\mathcal{K}$. Thus, $\sigma \Vdash_a x$ holds when variable x is true in state σ. The atomic forcing relation is monotone with respect to the partial ordering on states, *i.e.* $\forall \sigma.\ \sigma \Vdash_a x \ \Rightarrow\ \forall \sigma' \geq \sigma.\ \sigma' \Vdash_a x$.

The notion of *truth* in a Kripke structure is defined by the forcing relation. The atomic forcing relation on variables determines a forcing relation ($\Vdash$) on formulas with respect to the Kripke model. For a state σ in a given Kripke structure $\mathcal{K}$ and a formula ϕ, *σ forces ϕ in $\mathcal{K}$* is written $\sigma : \mathcal{K} \Vdash \phi$ (or if $\mathcal{K}$ is understood by context we simply write $\sigma \Vdash \phi$). The atomic forcing relation uniquely determines the forcing relation. If $\Vdash_a$ is the atomic forcing relation for a Kripke model $\mathcal{K}$, the forcing relation it induces is defined as follows.

$$
\begin{array}{lll}
\sigma \Vdash x & \text{iff} & \sigma \Vdash_a x \\
\sigma \Vdash \texttt{false} & \text{iff} & \texttt{False} \\
\sigma \Vdash \phi \wedge \psi & \text{iff} & \sigma \Vdash \phi \ \text{ and } \ \sigma \Vdash \psi \\
\sigma \Vdash \phi \vee \psi & \text{iff} & \sigma \Vdash \phi \ \text{ or } \ \sigma \Vdash \psi \\
\sigma \Vdash \phi \Rightarrow \psi & \text{iff} & \text{ for all } \sigma' \text{ such that } \sigma' \geq \sigma, \text{ if } \sigma' \Vdash \phi \ \text{ then } \sigma' \Vdash \psi
\end{array}
$$

We write $\mathcal{K} \Vdash \phi$ if, for every state $\sigma \in \Sigma$, $\sigma \Vdash \phi$. A formula is *Kripke valid* if it is forced in all Kripke models $\mathcal{K}$. A Kripke structure in which a formula ϕ is not forced is called a *Kripke counter-example for ϕ.*

Kripke's semantics for intuitionistic logic is sometimes motivated by considering Brouwer's notion of mathematics as a system of (growing) knowledge, what he called the *creative subject.* For such an account see Van Dalen [vD94]. Under this account, the states in a Kripke structure correspond to the mental states or stages of mathematical development of an idealized mathematician. Knowledge is monotone: once a mathematical fact is established, it holds in all future states. This epistemological structure forms the basis for the ordering of states in the Kripke model and plays an important part in the definition of truth via forcing in Kripke models.

However, Kripke semantics is *not faithful* to intuitionistic semantics. Smorynski [Smo73] and Dummett [Dum77] discuss this in some detail. Recall that, under Heyting's interpretation of Brouwer's intentions [TvD88], $\phi \Rightarrow \psi$ is intuitionistically valid when there is a construction transforming evidence for ϕ into evidence for ψ. But this is where the justification for Kripke semantics alluded to above breaks down. To see this, consider the forcing condition for implication. A formula of the form $(\phi \Rightarrow \psi)$ is forced at state σ if at some state σ', where $\sigma' > \sigma$, $\sigma' \Vdash \phi$ and $\sigma' \Vdash \psi$. This certainly does not capture the intended meaning of intuitionistic implication, since under the Kripkian interpretation, ϕ and ψ need not have any relation other than that they both occur as true formulas in some state later in the ordering.

Although they do not provide faithful intuitionistic semantics, following Underwood [Und94, pg.11–15], Kripke models serve as evidence of the unprovability of certain formulas, and not simply as abstract algebraic structures. In fact, we define a function below which maps systems (encodings of paths in the tableau search) to Kripke models. In the decidability proof, this function (`K`), is used to map failed (open) paths in the tableau search to Kripke models which can then be interpreted as evidence for unprovability. Thus, tableau construction and Kripke models are closely related. Failed tableau searches yield Kripke counter-examples. This use of Kripke models as counter-examples to intuitionistic provability has received attention elsewhere [PD95, Hud97].

More complete expositions on Kripke semantics for intuitionistic logic can be found in many sources [Fit69, TvD88, vD94, NS94].

5.1.4 Statement of the Theorem

By the nature of the constructive interpretation, a proof of a disjunction ($P \vee Q$) must indicate which of P or Q was proved and also must give evidence for its truth. Thus, the computational content of the Nuprl proof of intuitionistic decidability takes a propositional formula as its input and returns evidence for it truth or evidence for its absurdity.

```
    ∀f:Formula. is_valid(f)  ∨ {c:counter_example | c refutes f}
```

The reader may note that the shape of the theorem is essentially the same as that of the classical decidability theorem proved in Chapter 4.

In any case, we do not prove this theorem directly, but instead prove a more general theorem having enough structure to support our inductive proof. The more general theorem does not apply directly to formulas, but applies to tableau systems (lists of tableau nodes) satisfying an "eligibility" condition. Such structures have type `ESystem` (for *eligible system*). Evidence for the provability of an `ESystem` takes the form of a formal proof in a multi-conclusion sequent calculus. Evidence for its absurdity takes the form of a Kripke counter-example. Formally stated, the theorem we eventually prove in this chapter is the following:

```
* THM multi_decide
∀S:ESystem
   (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)} )
   ∨ {K:K{i}| ∃f:Node ⟶ Σ(K)
        ∀N:{N:Node| N∈S}
          ∀F:Formula. (F∈(T(N)) ⇒ forces(K,f N,F))
                   ∧ (F∈(F(N)) ⇒ not_forces(K,f N,F))}
```

To decide a formula ϕ, we will apply the computational content of this more general theorem to an eligible system containing a single node in which ϕ is assumed to be false. Should ϕ turn out to be provable, the result is a pair consisting of a tableau node and a proof of that node regarded as a sequent. Since the computational content of the theorem is intended to be applied to systems consisting of single nodes which contain a single formula, this evidently corresponds to a proof of the sequent `<[],[`ϕ`]>`. Should ϕ turn out not to be provable, the result is a Kripke counter-example. Kripke counter-examples here take the form of Kripke models defined over tableau nodes `N∈S` such that every formula in the true portion of the node (`T(N)`) is forced and every element in the false portion of the node (`F(N)`) is not forced. Since we will be applying the extracted program to initial systems consisting of a single nodes containing a single formula assumed to be false, the formula is not forced in the resulting Kripke model and so it serves as a counter-example.

5.2 Type Theoretic Formalization

In this section we present the formalization of the problem in type theory. This includes a the development of `Formula`, `Node`, and `Sequent` types, a type of `Kripke` models, and a `Proof` type. The definitions presented here are largely taken from Underwood [Und94] and Aitken, Constable and Underwood [ACU].

5.2.1 Variables and Formulas

For the purposes of this presentation, variables are taken to be elements of Nuprl's `Atom` type. We hide this in an abstraction declaring the type `Var` to be the type `Atom`. The only essential feature of the variable type is that it be a discrete type, *i.e.* that it have a decidable equality. Without adjusting the existing proofs it would be possible to substitute a more complex discrete type for the type `Atom` used here. Specifically, it would be possible to specify a more complex class of terms to stand for our variables.

Propositional formulas are defined by a Nuprl recursive type.

```
* ABS Formula
Formula  ≝   rec(FF.Var | Unit | FF × FF | FF × FF | FF × FF)
```

Thus a formula is either: a variable (element of the type `Var`); a constant, interpreted as `false`; a pair of formulas, representing a conjunction; a pair of formulas, representing a disjunction; or a pair of formulas, representing an implication. Intuitionistic negation $\neg P$ is just an encoding for $P \Rightarrow$ `False` and so we do not include it explicitly in our formula type. Neither do we include equivalence ($P \Longleftrightarrow Q$), which is defined to be $P \Rightarrow Q \ \wedge Q \Rightarrow P$.

It should be remarked that the `Formula` type is discrete.

A display form and an abstraction are defined for each element of the `Formula` type as follows:

```
* ABS fvar      ⌜F⌝  ≝   inl F
* ABS ffalse    ⌜false⌝  ≝   inr (inl ·)
* ABS fand      (p⌜∧⌝q)  ≝   inr inr (inl <p, q> )
* ABS f_or      (p⌈∨⌉q)  =def  inr inr inr (inl <p, q> )
* ABS fimp      (p⌈⇒⌉q)  =def  inr inr inr inr <p, q>
```

Case analysis over formulas is provided by the `formula_case` operator.

5.2.2 Sequents, Nodes, and Systems

We define sequents as pairs of formula lists.

```
* ABS Sequent
Sequent  =def  Formula List × Formula List
```

`Sequent` is a discrete type. This follows from the discreteness of the `Formula` type.

Nodes of the tableau system are given by the type `Node`. Like sequents, they are pairs of formula lists.

```
* ABS Node     Node  =def  Formula List × Formula List
```

The elements in the first component of a node are those formulas assumed to be true, the elements in the second component are those elements assumed to be false. For a node `N` we will refer to these components by writing `T(N)` for `N.1` and `F(N)` for `N.2`. We will sometimes refer to formulas occurring in the true part of node `N`, *i.e.* in `T(N)`, as a *positive occurrences.* We refer to formulas occurring in the false part of `N` (`F(N)`) as *negative occurrences.*

Also, note that `Node` is a discrete type.

Nodes are coerced to type `Sequent` by the identity operator.

```
* ABS NtoS     asSequent(N)  =def  N
```

There is a potential point of confusion about positive and negative parts of nodes and sequents. Because of the underlying semantic interpretation of nodes (in the search for counter-examples), when nodes are coerced to sequents under the mapping `asSequent`, the polarity of positive and negative parts switches. Thus a positive occurrence of a formula in a node becomes a negative occurrence in a sequent, and vice versa. The nomenclature of polarity has no bearing on the formal development but is useful terminology, even if potentially confusing.

A `System` is a non-empty list of nodes.

```
* ABS System
System  ≝   Node List+
```

Here, `List+` is the type of non-empty lists.

For the purposes of the proof we will define (below) a class of *eligible systems* (`ESystem`) as a subtype of `System`.

5.2.3 Kripke Semantics

The type theoretic characterization of Kripke models presented here follows the presentation of [ACU]. A Kripke (model) is a dependent triple consisting of a type (of states), a reflexive and transitive relation on the states, and an atomic forcing function.

```
* ABS Kripke
K{i}  ≝
    T:𝕌
    × R:{R:(T × T) ⟶ ℙ |
           ∀a,b,c:T. R(<a, a>) ∧
```

```
                    R(<a, b>) ⇒ R(<b, c>) ⇒ R(<a, c>)}
  × {af:T → Var → ℙ |
      ∀a:T. ∀v:Var.
        af(a)(v) ⇒ (∀b:T. R(<a, b>) ⇒ af(b)(v))}
```

The first element of the triple is the carrier type which is interpreted as the set of states of the Kripke model. Note that it is not always possible to simply model a set by a type because they differ on their membership relations (set membership is always a well-formed proposition while type membership is only a well-formed proposition when true): these differences are not material to the development here. The following abstraction encodes the selector referring to this component.

`* ABS K_state` $\quad \mathbf{\Sigma}$`(K)` $\overset{\text{def}}{=}$ `K.1`

The second component of the Kripke model is a reflexive and transitive relation on states. This relation presents the structure of Kripke model by relating states to one another. The relation is formally modeled here as a set type of propositional functions enjoying the appropriate properties. The selector for this component of the model is defined as follows.

`* ABS K_rel` $\quad \leq\{$`K`$\}$ $\overset{\text{def}}{=}$ `K.2.1`

When `K` is understood by context, for states `s` and `s'` we will use the display `s`$\leq$`s'` instead of $\leq\{$`K`$\}$`(<s,s'>)`.

The third element of the triple is the *atomic forcing* relation specifying those atomic formulas which are true in a given state. Thus, it is a proposition on states `T` and variables (`Var`) that is monotone with respect to the state relation `R`. That is, if a variable is forced by `af` in some state `s`, then it is forced in all states `s'` such that `s≤s'`. For every variable `v` forced at state `s`, the function `K.af(s)(v)` evaluates to `True`. Those variables `v'` such that `¬K.af(s)(v')` holds are not forced at `s`. As expected, the Nuprl selector for the atomic forces relation for a Kripke model is defined as follows:

```
* ABS K_af        K.af  ≝  K.2.2
```

Triples inhabiting the type `Kripke` may be called *Kripke structures*.

5.2.3.1 The Forces and Not_Forces Relations

As stated, the main theorem requires definitions of both *forces*, and its complement *not forces*. The reader may already realize that we cannot simply define the complementary notion by taking the constructive negation of the definition of forcing. Examining the definition of forcing reveals a conjunction in the condition for conjunctive formulas; $\neg(P \wedge Q)$ does not, in general, constructively imply $\neg P \vee \neg Q$. Also, the universal quantifier in the case of implication is problematic, thus $\neg\forall x : T.P[x]$ does not constructively imply $\exists x : T.\neg P[x]$.

Based on these observations we adopt Underwood's method of [Und93] and define the forces and not-forces relations simultaneously by mutual recursion. Definition by mutual recursion is not supported by Nuprl tactics (although there is no technical reason it cannot be) and we use the pairing trick to implement it here.

```
* ABS forcing_pair
<forces,not_forces>{K}  ≝
    (letrec f_nf(s)(f) =
      case f:
        ⌜x⌝ ⟶ <K.af(s)(x), ¬(K.af(s)(x))>;
    ⌈false⌉ ⟶ <False, True>;
    a⌈∧⌉b ⟶ <(f_nf(s)(a)).1 ∧ (f_nf(s)(b)).1,
              (f_nf(s)(a)).2 ∨ (f_nf(s)(b)).2>;
    a⌈∨⌉b ⟶ <(f_nf(s)(a)).1 ∨ (f_nf(s)(b)).1,
              (f_nf(s)(a)).2 ∧ (f_nf(s)(b)).2>;
    a⌈⇒⌉b ⟶ <∀s':Σ(K). ≤{K}(<s, s'>) ⇒
                  (f_nf(s')(a)).2 ∨ (f_nf(s')(b)).1,
              ∃s':Σ(K). ≤{K}(<s, s'>) ∧
                  (f_nf(s')(a)).1 ∧ (f_nf(s')(b)).2>;
)
```

Thus, for a Kripke structure `K`, the function `<forces,not_forces>{K}` is defined as a mutually recursive pair by `f_nf`. The arguments to the function are a state *s* and a formula *f*. The function definition is best understood if the term `f_nf(`*s*`)(`*f*`).1` is read as ``` ``s forces f'' ``` and the term `f_nf(`*s*`)(`*f*`).2` is read as ``` ``s does not force f'' ```. For a fixed state `s` the computation proceeds by case analysis on *f*. In the case where *f* is a variable the result is determined by the atomic forcing relation. If the formula *f* is the constant ⌈false⌉ then, for all states the pair of Nuprl constants `<False, True>` is returned, *i.e.* ⌈false⌉ forces `False`, and it does not force (`not_forces`) `True`. If the argument *f* is a conjunction of the form `a⌈∧⌉b`, then it is forced at `s` when both `a` and `b` are. Oppositely, `a⌈∧⌉b` is not forced at `s` if either `a` is not forced at `s` or if `b` is not forced at `s`; `a⌈∨⌉b` is not forced at `s` if `a` is not forced at `s`, and if `b` is not forced at `s`. Finally, an implication `a⌈⇒⌉b` is forced at a state `s` if for every state `s'∈Σ(K)`, if `(s≤s')`, `a` is not forced at `s'` or `b` is forced at `s'`. An implication `a⌈⇒⌉b` is not forced at a state `s` if there is some state `s'∈Σ(K)`, `(s≤s')`, `a` is forced at `s'` and `b` is not forced at `s'`. This gives the Nuprl encoding of the standard notion of forcing in a Kripke structure.

Using this definition we define the `forces` and `not_forces` relations as follows.

```
* ABS forces
forces(K,S,f)  ≝   (<forces,not_forces>{K}(S)(f)).1
* ABS not_forces
not_forces(K,S,f)  ≝   (<forces,not_forces>{K}(S)(f)).2
```

Eventually, we are interested in viewing tableau systems as Kripke structures. The following function serves to map systems into Kripke models.

```
* ABS K_structure
K(S)  ≝   <{N:Node| N∈S} , λ<n,m>.T(n)⊆T(m), λN,x.⌜x⌝∈T(N)>
```

Thus, under the interpretation, states of the corresponding Kripke model consist of the type whose members are those nodes in the system. The ordering on pairs of nodes is defined by sublist inclusion on the formulas assumed to be true in the nodes. The atomic forcing function for a state `N` and a variable `x` is defined by membership of the atomic formula `⌜x⌝` among formulas assumed true at `N`.

The well-formedness goal shows that Systems `S` do indeed map to Kripke models under `K`.

```
* THM K_structure_wf  ∀S:System. K(S) ∈ K{i}
```

In the case of a failed tableau search, culminating in a system `S`, the corresponding Kripke structure `K(S)` will serve as the counter-example.

Since we are interested in the possibility of reflecting our decision procedure into Nuprl, it is better for provability to be represented by either a Nuprl extract term or, more abstractly, by a formal proof. We choose the latter representation, motivating the following section.

5.2.4 A Formal Proof Type

One form of evidence for the validity of a formula is a proof in a sound and complete sequent calculus for propositional intuitionistic logic. In the decidability proof for classical case presented in Chapter 4 we showed how the soundness and invertibility of the sequent proof rules contributed to the the proof; here we will formally define a proof type and, in the case the formula to be decided is provable, a formal proof of that fact will be returned as evidence.

Proofs are formally modeled in two stages. A `pre_proof` is a recursive type representing the shape (tree structure) of a proof. A predicate is then defined to determine when `pre_proofs` are well-formed according to the proof rules of system *MJ*, *i.e.* when their structure corresponds to what would be accepted as a sequent proof. The `Proof` type is then defined as the subtype of `pre_proofs` that are well-formed.

```
* ABS pre_proof
pre_proof  =def  rec(P. Sequent
                   | Sequent × Sequent × P
                   | Sequent × Sequent × P × Sequent × P)
```

Thus, a `pre_proof` is either: a single sequent, *i.e.* if well-formed, it is an axiom or a sequent containing ⌜`false`⌝ in its hypothesis list; or it is a triple consisting of two sequents and a `pre_proof`, *i.e.* if well-formed it is a `pre_proof` whose root was derived by a proof rule with a single hypothesis and whose hypothesis is verified by the accompanying `pre_proof`; or it is a quintuple containing three sequents and two proofs, *i.e.* if it is well-formed it is a proof in which the last rule applied was a rule having two hypotheses, each of which is verified by the corresponding `pre_proofs`.

We define selectors for the various types of proof nodes and a `pre_proof` case analysis operator.

```
* ABS pre_proof_axiom
s\  =def  inl s
* ABS pre_proof_rule1
C\<H,p>  =def  inr (inl <C, H, p> )
* ABS pre_proof_rule2
C\<H1,p1>,<H2,p2>  =def  inr inr <C, H1, p1, H2, p2>
* ABS pre_proof_case
case P:
      h\ ⟶ Axiom[h];
      c1\<h1,p1> ⟶ Rule1[c1; h1; p1];
      c2\<h2,p2>,<h3,p3> ⟶ Rule2[c2; h2; p2; h3; p3];
 =def  case P of
      inl(h) => Axiom[h] | inr(P) => case P of
      inl(r2) => let c1,h1,p1 = r2 in Rule1[c1; h1; p1] |
      inr(r3) => let c2,h2,p2,h3,p3 = r3 in
                   Rule2[c2; h2; p2; h3; p3]
```

Using the case analysis we define operators for manipulating the hypotheses and conclusions of a `pre_proof`.

```
* ABS hyp
Hyps(p)  ≝  case p:
           c\ ⟶ [];
           c\<h,p'> ⟶ (h::[]);
           c\<h,p>,<h',p'> ⟶ (h::h'::[]);
* ABS concl
Concl(p)  ≝  case p:
              c\ ⟶ c;
              c\<h,p> ⟶ c;
              c\<h,p>,<h',p'> ⟶ c;
```

The proof rules of the calculus are defined by the two definitions, one for rules having a single hypothesis and another for rules having two hypotheses.

```
* ABS proof_rule1
c\h is a rule instance  ≝
  ∃a,b:Formula.
     ((a⌈∨⌉b) ∈ Concl(c) ∧ h = <Hyps(c),a::Concl(c)>)
     ∨ ((a⌈∨⌉b) ∈ Concl(c) ∧ h = <Hyps(c),b::Concl(c)>)
     ∨ ((a⌈⇒⌉b) ∈ Concl(c) ∧ h = <a::Hyps(c), b::[]>)
     ∨ ((a⌈∧⌉b) ∈ Hyps(c) ∧ h = <a::Hyps(c), Concl(c)>)
     ∨ ((a⌈∧⌉b) ∈ Hyps(c) ∧ h = <b::Hyps(c), Concl(c)>)
```

The clauses of `proof_rule1` correspond to the five rules ($\vee r1$, $\vee r2$, $\Rightarrow r$, $\wedge l1$, and $\wedge l2$) of system *MJ* having only one hypothesis (see Figure 5.1). Consider the case when `c\h` is in instance of the $\Rightarrow r$ rule. There must exist `a` and `b` such that the formula `(a⌈⇒⌉b)` occurs in the conclusion of sequent `c`, and `h` must be the sequent `<a::Hyps(c), b::[]>`. The equality used is the type equality for sequents (defined as pairs of formula lists) and so order counts; this is not the semantic (permutation) equality on sequents. The reader can verify by inspection that these clauses match the rules of system *MJ*.

The rules ($\vee l$, $\wedge r$, and $\Rightarrow r$) are characterized by the following definition.

```
* ABS proof_rule2
c\<h1,h2> is a rule instance  ≝
  ∃a,b:Formula
    ((a⌜∧⌝b) ∈ Concl(c)  ∧ h1 = <Hyps(c), a::Concl(c)>
                         ∧ h2 = <Hyps(c), b::Concl(c)>)
    ∨ ((a⌜∨⌝b) ∈ Hyps(c)  ∧ h1 = <a::Hyps(c), Concl(c)>
                          ∧ h2 = <b::Hyps(c), Concl(c)>)
    ∨ ((a⌜⇒⌝b) ∈ Hyps(c)  ∧ h1 = <Hyps(c), a::Concl(c)>)
                          ∧ h2 = <b::Hyps(c), Concl(c)>
```

Now we can define well-formedness for pre-proofs. A pre-proof is well-formed if:

i.) its leaves are all instances of the `false`*l* rule or the *Ax* rule, and

ii.) every non-leaf node matches a conclusion of some rule instance and its children match the premises of that rule.

This characterization is formalized by the following recursive function.

```
* ABS well_formed
 p is a Proof  ≝
  (letrec isap P =
     case P:
       c\ ⟶ (∃f∈Hyps(c).(f∈Concl(c)) ∨ ⌜false⌝∈Hyps(c));
       c\<h,p> ⟶ (c\h is a rule instance ∧
                      h = Concl(p) ∧
                      isap p);
       c\<h,p>,<h',p'> ⟶ (c\<h,h'> is a rule instance ∧
                              h = Concl(p) ∧
                              h' = Concl(p') ∧
                              isap p ∧
                              isap p');
  ) p
```

This definition gives the means to define the proof type.

`* ABS proof      Proof` $\overset{\text{def}}{=}$ `{p:pre_proof| p is a Proof}`

We also formalize the notion of a proof `P` proving a sequent `S`.

`* ABS proves     P proves S` $\overset{\text{def}}{=}$ `Concl(P) = S`

5.2.5 Eligible Systems and System Completeness

The tableau construction is defined inductively, starting with an initial system containing a single node. We do not explicitly define a tableau type or a type of tableau rules: the tableau is implicit in the structure of the inductive proof and the rules are implicit in the proof steps instantiating the inductive hypothesis. Systems, incrementally expanded during the tableau construction, correspond to paths in the tableau. Should we collect the set of expanded systems, we could reconstruct an explicit tableau, but there is no need. Even though we do not explicitly define tableaux or tableau rules, we will sometimes refer to a step of tableau development as the application of a tableau rule.

There is a close correspondence between the steps of tableau construction and the proof rules of system *MJ*. For each proof rule there is a corresponding step of tableau development. For proof rules having a single premise there is a corresponding tableau development step in which an existing node is extended or, in the case of $\Rightarrow r$, the tableau system itself is extended by the addition of a new node. For proof rules having two premises, the corresponding tableau step extends an existing node in the tableau in two different ways, invoking the induction hypothesis (unfolding a step of recursion) on these extended systems. This bifurcation of systems corresponds to a branching in the tableau structure. We call the tableau steps corresponding to rules other than the $\Rightarrow r$ rule *local* rules, as they only extend existing nodes.

When a node has been developed as far as possible under the local rules we say it is *node complete.* Having defined node completeness, we focus our attention on *eligible systems*, systems restricted to contain at most one member that is not node complete. Tableau systems containing all possible node extensions induced by occurrences of $\Rightarrow r$ are called *system complete.*

The underlying tableau construction starts with an eligible system. If all nodes are complete and the system is complete, the tableau is complete. Since the system is eligible, it contains at most one node incomplete node. This node is developed as far as possible under the local rules, each step preserving the eligibility of the system. Once the sole incomplete node has been completed, the system is examined to see if it is system complete; if so we have developed the system as far as possible and the tableau is complete; if not then the system is extended by applying the $\Rightarrow r$ rule to some previously unextended node thereby extending the system. This extension preserves eligibility. This procedure is repeated until the system is complete. The termination argument for this procedure is given below.

Now, we give type theoretic definitions for node completeness, eligibility, and system completeness.

5.2.5.1 Node Completeness

A node is *node complete* when further development of that node under the local tableau rules adds no new information. We formalize this condition by the following definition.

```
* ABS node_complete
  nComplete(N)  ≝
    ∀f∈(T(N)).case f:
                    ⌜x⌝ ⟶ True;
                    ⌜false⌝ ⟶ True;
                    a⌜∧⌝b ⟶ (a∈(T(N)) ∧ b∈(T(N)));
                    a⌜∨⌝b ⟶ (a∈(T(N)) ∨ b∈(T(N)));
                    a⌜⇒⌝b ⟶ (a∈(F(N)) ∨b∈(T(N)));
    ∧ ∀f∈(F(N)).case f:
                       ⌜x⌝ ⟶ True;
                       ⌜false⌝ ⟶ True;
                       a⌜∧⌝b ⟶ (a∈(F(N)) ∨ b∈(F(N)));
                       a⌜∨⌝b ⟶ (a∈(F(N)) ∧ b∈(F(N)));
                       a⌜⇒⌝b ⟶ True;
```

Note that this is a decidable property.

```
* THM decidable__node_complete  ∀N:Node. Dec(nComplete(N))
```

5.2.5.2 Eligible Systems

Given our definition of node completeness, we define the subclass of systems that are eligible. In [ACU], a system is *eligible* if it contains at most one node that is not complete. Here, we strengthen the eligibility condition, so a system is eligible if either every node in the system is complete or if the only incomplete node is at the head of the list.

```
* ABS eligible
Eligible(S) ==
 (∀N∈S. nComplete(N))
 ∨ let N::rest = S in ¬nComplete(N) ∧ (∀N'∈rest. nComplete(N'))
```

The strength of this definition of eligibility is not as fully utilized as it might be in the main proof.

The incomplete node in an eligible system (if there is one) will be called the *eligible node*.

5.2.5.3 System Completeness

New nodes are added to a system when a negative occurrence of an implication is decomposed.

A node `N` containing a negative occurrence of an implication of the form `a⇒b` is *subsumed* by any node `N'` if `T(N)⊂T(N')` and `a∈T(N')` and `b∈F(N')`.

We only wish to extend a system with a new node when it does not already contain a node which subsumes it. Otherwise, applying the decomposition tableau rule is redundant. A system is *complete* when every node containing a negative occurrence of an implication is subsumed by some node already in the system.

The following predicate defines system completeness.

```
* ABS system_complete
sComplete(S)  ≝
 ∀N∈S.
   ∀f∈F(N).
     case f: ⌈x⌉ ⟶ True;
             ⌈false⌉ ⟶ True;
             a⌈∧⌉b ⟶ True;
             a⌈∨⌉b ⟶ True;
             a⌈⇒⌉b ⟶ ∃N'∈S.(T(N) ⊆ T(N') ∧ a∈T(N') ∧ b∈F(N'));
```

The following lemma, an immediate consequence of the definition of system completeness, characterizes incomplete systems.

```
* THM not_system_complete
∀S:ESystem
  ¬sComplete(S)
    ⇒ (∃N∈S
         ∃a,b:Formula. (a⌈⇒⌉b)∈F(N)
                     ∧ (∀N'∈S . ¬a∈T(N') ∨ ¬b∈F(N')))
```

5.2.6 Termination

Underwood's termination argument for the construction, as presented in [ACU], is based on a lexicographic ordering of two measures on systems. Ultimately they depend on the fact that tableau construction has the *subformula property*, *i.e.* in a tableau construction from an initial system S, only subformulas of formulas already occurring in S ever appear in the tableau.

Roughly, the first measure (i1) is on the number of nodes that may ever be added to a system. The second measure (i2) is on the number of formulas that may ever be added to a node. These measures are calculated by computing conservative upper bounds on the sizes of the respective structures and then taking the difference between these bounds and the actual sizes of the objects being constructed as the measure. Since nodes and systems grow during the tableau construction phase, the difference decreases. Thus, at each step of the tableau construction process, one or the other measure decreases, which is enough to show termination. The bounds are never achievable in an actual tableau development and so we terminate the process when all nodes are complete and when the system is complete.

The lexicographic measure is defined as follows.

```
*ABS  System_lt
  S < S'  ≝  i1(S) < i1(S') ∨ (i1(S) = i1(S') ∧ i2(S) < i2(S'))
```

Under the lexicographic measure induction principle used in the proof, the measures do not contribute to the computational content of the program extracted from the proof. They have not been formalized in Nuprl. Indeed, it would be possible to verify the properties required of them in another system, even a classical system. In [CU96], Underwood and Caldwell present arguments that would allow properties of the measure functions to be verified in a classical system like PVS [OS95]. This is possible because the measures do not contribute to the computational content.

The stipulated lemmas related to the correctness of the measure functions and the steps of tableau development are listed in Appendix A. They have not been proved in Nuprl, but they are assumed in the proof presented below. Stipulated lemmas are added in Nuprl by creating the requisite theorem object and then declaring it to be true by `Fiat`. Unfortunately, the current implementation of Nuprl does not indicate uses of `Fiat`; however, in Version 5 of the Nuprl system, occurrences of stipulated lemmas will be noted in the library.

Each of the stipulated lemmas characterizes one step of tableau development. We present one here.

```
*THM and_1_positive_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈∧⌉b)∈(T(N)) ⇒ ¬(a∈(T(N))) ⇒
    (<a::(T(N)), F(N)>::remove(N;S)) ∈ {k:ESystem| k < S}
```

The informal proofs of the stipulated lemmas are justified by arguments given in [ACU]. In every case (except the `implies_negative_rule`) the argument that the extended system is eligible is based on the fact that the element removed is the incomplete node, and it is replaced in the system by the extended node. It does not matter if this node is complete or not, since the new system is eligible if the original one was. In the case of `implies_negative_rule`, by assumption, every node in the system `S` is complete and so the extended system can have at most one incomplete node, which has been added at the head of the list `S`, thus the system is eligible.

The reader can easily see that, in every case, the measure of the extended system is less than the measure of the system `S`. See [ACU] for a more detailed argument to this effect.

5.3 The Formal Proof

In this section we present an informal account of the proof to orient the reader. This is followed by the formal proof of intuitionistic decidability which is followed by the formal proof of the base case construction.

5.3.1 An Informal Account

We wish to prove the following theorem.

```
* THM multi_decide
∀S:ESystem
    (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)} )
    ∨ {K:K{i}| ∃f:Node ⟶ Σ(K)
        ∀N:{N:Node| N∈S}
          ∀F:Formula. (F∈(T(N)) ⇒ forces(K,f N,F))
                   ∧ (F∈(F(N)) ⇒ not_forces(K,f N,F))}
```

The proof is by induction on eligible systems, *i.e.* on systems having at most one node that is not node complete. The induction principle is the lexicographic measure induction presented in Chapter 3, and we apply it here using the measure functions i1 and i2 defined above. Recall that the first measure decreases with every node added to the system while the second decreases as formulas are added to the eligible node. The resulting induction hypothesis asserts that the the theorem holds for all systems below S in the lexicographic ordering.

Consider an arbitrary eligible system S; either it contains an eligible node or not. Suppose there is one; then since eligible nodes are expanded in place by adding subformulas of formulas already occurring in S, the tableau expansion steps for these rules reduce the second measure. The proof rules $\vee r1$, $\vee r2$, $\wedge l1$, and $\wedge l2$ correspond to local tableau steps and all have one premise. In these cases, the induction hypothesis is instantiated with the system constructed from S by extending the eligible node with subformulas as specified by the corresponding proof rule. The proof rules $\vee l$, $\wedge r$, and $\Rightarrow l$ all have two premises and so we instantiate two copies of the induction hypothesis; one with the system constructed by expanding the eligible node with the subformulas specified in the left premise of the corresponding proof rule; and the other with a system created by expanding the eligible node by adding subformulas as specified by the right premise of the corresponding rule. In each case, the result of instantiating the induction hypothesis is either a node-proof pair for the extended (lexicographically smaller) system or it is a Kripke counter-example for the extended system. Whenever a Kripke counter-example exists, it serves to refute the S as well. In the case node-proof pairs result from the instantiated induction hypotheses, they are used to identify a node in S and to construct a proof for it. The details of these constructions are elaborated below in the presentation of the formal proof.

Now, suppose there is no eligible node in S. Then we consider whether the system is complete or not. If it is not complete then there is some node containing an occurrence of $\Rightarrow r$ which has not been accounted for in the system. This is the case that distinguishes the intuitionistic case from the classical case. Recall from Chapter 4 the classical proof rule for implies on the right.

$$\frac{q,\ hyp \vdash r,\ M,\ N}{hyp \vdash M,\ q\lceil\Rightarrow\rceil r,\ N}$$

The antecedent of the premise of this rule retains the formulas in `M` and `N`. The corresponding proof rule for implication on the right in system *MJ* eliminates all the formulas in `M` and `N` from the antecedent of the premise.

$$\frac{q, H \vdash r}{H \vdash M, q\lceil\Rightarrow\rceil r, N} \Rightarrow r$$

Thus, the intuitionistic proof rule effectively chooses which of the formulas among `M,r,N` is to be proved. If the wrong choice is made, the proof may not succeed. The search procedure implemented by the intuitionistic tableau methods takes this impermutability of inference steps into account by introducing new tableau nodes into the system each time an implication on the right (that has not been expanded) is encountered. By adding new nodes in this way, different systems represent the search for different possible proofs, each resulting from the application of an $\Rightarrow r$ rule at a different point in the proof. The classical tableau rules are all local and thus all rules are permutable so this consideration does not arise. Based on this discussion we see that one or more nodes may cause a system to be incomplete, because it contains one or more nodes having implications on the right that have not been fully explored in the tableau construction yet.

Continuing with the proof; choose one of the nodes causing the system to be incomplete, call it `N`. We decompose the induction hypothesis by extending `S` with a new node constructed from `N` and accounting for the application of the $\Rightarrow r$ rule. This extended system is lower in the lexicographic ordering of systems since the measure `i1` is reduced whenever a node is added to `S`. As above, the instantiation of the induction hypothesis results either in a node-proof pair or a Kripke counter-example for the expanded system. These structures are used to construct the same for the system `S`.

Finally, if all nodes are complete and the system is complete, then we are in the base case where one of a node-proof pair or a Kripke counter example is constructed directly without reference to the induction hypothesis. If the system contains a node that, viewed as a sequent, is an axiom, then that node is returned paired with the instance of the axiom rule. If not, then a Kripke counter-model is constructed by applying the function `K` to the system, producing a counter-model of the form

```
<{N:Node| N∈S} , λ<n,m>.T(n)⊆T(m), λN,x.⌈x⌉∈T(N)>
```

The details of the verification that this is indeed a Kripke counter-example satisfying the properties specified of it in the main theorem are left until the presentation of the formal proof below.

This completes the informal proof of decidability the intuitionistic case.

5.3.2 Intuitionistic Decidability

In this section we present the formal proof in some detail.

```
* THM multi_decide
∀S:ESystem
    (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)} )
    ∨ {K:K{i}| ∃f:Node ⟶ Σ(K)
        ∀N:{N:Node| N∈S}
          ∀F:Formula. (F∈(T(N)) ⇒ forces(K,f N,F))
                    ∧ (F∈(F(N)) ⇒ not_forces(K,f N,F))}
```

The proof is by lexicographic measure induction on the eligible system `S`. The induction principle itself is justified by the lexicographic measure induction lemma presented in Chapter 3. Invoking the tactic `LexOrderMeasureInd` automatically instantiates it with the proper induction hypothesis. After the induction step, Nuprl displays the following proof state.

```
1.  S: ESystem
2. ∀k:{k:ESystem| k < S}
     (∃N:{N:Node| N∈k} . {p:Proof| p proves asSequent(N)})
     ∨ {K:K{i}| ∃g:Node ⟶ Σ(K)
         ∀N:{N:Node| N∈k}
           ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                     ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
⊢ (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)})
   ∨ {K:K{i}| ∃g:Node ⟶ Σ(K)
       ∀N:{N:Node| N∈S}
         ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                   ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
```

Thus, we may assume that there is either a proof or a Kripke counter example for eligible systems lexicographically below `S`.

To save space in the presentation of the proofs: we will write $\mathcal{C}$ to denote the conclusion of this sequent; we will write $\mathcal{IH}$ as an abbreviation for the term appearing in hypothesis (2) above; and we will only indicate new or changed hypotheses. Redisplaying this sequent under these conventions it appears as:

1. `S: ESystem`
2. $\mathcal{IH}$

$\vdash \mathcal{C}$

The proof proceeds by first deciding if all nodes in `S` are node complete. If not then some local tableau rule is applicable, and then one or two instances of $\mathcal{IH}$ are instantiated with the node(s) corresponding to those generated by the appropriate tableau expansion rule. The result of the instantiation is either a proof or a Kripke counter-example for the inductively smaller case(s), which are in turn used to construct a proof or counter-example satisfying the main goal of the theorem at that point in the proof.

If all the nodes in the system are node complete, then no local tableau rules apply to any node in `S`. Now it must be decided whether the system is complete. If not, then a new node is added to the system as specified by the $\Rightarrow r$ tableau rule and the induction hypothesis is invoked on the extended system, the result of which is used to discharge this case.

If the system is complete, then no tableau rule applies. This is the base case and is discharged by appeal to the following lemma.

```
* THM decidability_base
∀S:ESystem
    sComplete(S)
    ⇒ ∀N∈S.nComplete(N)
    ⇒ (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)})
       ∨ {K:K{i}| ∃g:Node → Σ(K)
           ∀N:{N:Node| N∈S}
             ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                       ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
```

The proof of the base case contains the construction of the initial proof or Kripke model asserted to exist by the theorem. The proofs of the inductive steps specify how these initial structures are extended by each call to the induction hypothesis. We delay the proof of this important theorem until after we have completed the main theorem.

To decide whether any local tableau rules apply we invoke the following tactic: `Decide` ⌜∀N∈S.nComplete(N)⌝. This results in two cases. We consider the negative case first (*i.e.* the case where there is an incomplete node in `S`).

5.3.2.1 `S` contains an incomplete node

A direct consequence of a node not being complete is that some local tableau rule must apply. This idea is captured by the following lemma.

```
* THM not_node_complete
∀N:Node
    ¬nComplete(N)
    ⇒ (∃a,b:Formula
         ((a⌜∧⌝b)∈(T(N)) ∧ (¬a∈(T(N)) ∨ ¬b∈(T(N))))
         ∨ ((a⌜∨⌝b)∈(T(N)) ∧ ¬a∈(T(N)) ∧ ¬b∈(T(N)))
         ∨ ((a⌜⇒⌝b)∈(T(N)) ∧ ¬a∈(F(N)) ∧ ¬b∈(T(N)))
         ∨ ((a⌜∧⌝b)∈(F(N)) ∧ ¬a∈(F(N)) ∧ ¬b∈(F(N)))
         ∨ ((a⌜∨⌝b)∈(F(N)) ∧ (¬a∈(F(N)) ∨ ¬b∈(F(N)))))
```

We take up the proof of decidability after having named the incomplete node.

```
  3. ¬∀N∈S.nComplete(N)
  4. N: {N:Node| N∈S}
  5. ¬nComplete(N)
  ⊢ 𝒞
```

Forward chaining through the lemma `not_node_complete` with hypothesis (5) results in five subgoals, one for each local tableau rule and characterized by one of the clauses of the lemma. Each clause corresponds to a case where the consequences of some formula assumed to be true or false have not yet been accounted for in the tableau. We will call such formula occurrences *eligible*.

An eligible conjunction occurs positively:

```
6. a: Formula
7. b: Formula
8. (a⌈∧⌉b)∈(T(N))
9. ¬a∈(T(N)) ∨ ¬b∈(T(N))
 ⊢ 𝒞
```

By hypothesis (9), at least one of `a` or `b` is not among the positive formulas of `N`. To proceed we split on hypothesis (9). Guided by the rules for $\wedge l1$ and $\wedge l2$ respectively: in the first case we instantiate the induction hypothesis (2) with the system `<a::(T(N)), F(N)>::remove(N;S)` ; and in the second case with `<b::(T(N)), F(N)>::remove(N;S)`. Since the proofs of the two cases are nearly identical, we present the first in some detail, leaving the other to the imagination of the reader.

The instantiation of the induction hypothesis results in a well-formedness obligation of the following form.

```
9. ¬a∈(T(N))
⊢ <a::(T(N)), F(N)>::remove(N;S) ∈ {k:ESystem| k < S}
```

This is proved by appeal to `and_1_positive_rule` presented above.

The main thread of the proof continues with the following subgoal.

```
    10. (∃N:{N1:Node| N1∈(<a::(T(N)), F(N)>::remove(N;S))}
              {p:Proof| p proves asSequent(N)})
          ∨ {K:K{i}| ∃g:Node → Σ(K)
               ∀N:{N1:Node| N1∈(<a::(T(N)), F(N)>::remove(N;S))}
                 ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                           ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
  ⊢ 𝒞
```

Decomposing the disjunct in hypothesis (10) results in two subgoals: in one, there is a proof of some node in the system `<a::(T(N)),F(N)>::remove(N;S)`, and in the other there is a Kripke counter-example for some node in the system.

First, consider the case where there is a proof.

```
10. N1: Node
11. N1∈(<a::(T(N)), F(N)>::remove(N;S))
12. p: Proof
[13]. p proves asSequent(N1)
⊢ ∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)}
```

By (11) we know that either `N1` is `<a::(T(N)), F(N)>` or it's a member of the list `remove(N;S)`. In the first case, we use `N` as witness for the existential in the conclusion resulting in the following proof state.

```
11. N1 = <a::(T(N)), F(N)>
12. p: Proof
[13]. p proves asSequent(N1)
⊢ {p:Proof| p proves asSequent(N)}
```

We construct a witness for `p` using the $\wedge l1$-rule as a model. Specifically, the witness is the pre-proof term `asSequent(N)\<asSequent(N1),p>`. The subgoal induced by this decomposition obliges a proof that this term is indeed a proof and that it proves `N`. The reader can verify this for himself by referring to the definition of the proof type given above and by referring to the $\wedge l1$-rule.

Now, suppose instead that `N1` is in `remove(N;S))`. Then `p` is already a proof of some node in `S` and so we eliminate the existential `N` in the conclusion with `N1` and use `p` to witness the proof.

Now consider the case where, instead of a proof, we have a Kripke counter-example, then we must show:

```
10. K: K{i}
11. g: Node ⟶ Σ(K)
12. ∀N:{N1:Node| N1∈(<a::(T(N)), F(N)>::remove(N;S))} .
      ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                 ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
⊢ ∃g:Node ⟶ Σ(K)
    ∀N:{N:Node| N∈S}
      ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                 ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
```

Here we must supply a function that maps nodes to states of `K` such that every formula in the nodes of `S` is forced or not forced according to its membership in the positive or negative part. By the induction hypothesis, we already have a function `g` mapping nodes in the system `<a::(T(N)), F(N)>::remove(N;S)` to states of `K`. Since this step of tableau development has replaced the node `N` with `<a::(T(N)), F(N)>`, we construct `g'` to first check if the argument node is equal to `N` and if so we map it to whatever state `g(<a::(T(N)), F(N)>)` does; otherwise, we just apply `g` to the argument. The following lambda term witnesses this function.

```
λn.if (n = N) then g(<a::(T(N)), F(N)>) else g(n) fi
```

We apply this witness and then decompose the universal quantifiers in the conclusion. Reducing the applications of the witness in the conclusion and case splitting on whether `N1 = N` or not results in two subgoals. The first having the following form:

```
14. (N1 = N)
⊢ (f∈(T(N1)) ⇒ forces(K,g(<a::(T(N)), F(N)>),F))
   ∧ (f∈(F(N1)) ⇒ not_forces(K,g(<a::(T(N)), F(N)>),F))
```

and, the other having the form:

```
14. ¬(N1 = N)
⊢ (f∈(T(N1)) ⇒ forces(K,g(N1),f))
   ∧ (f∈(F(N1)) ⇒ not_forces(K,g(N1),f))
```

Both of these subgoals are discharged by backchaining through hypothesis (11). This completes this branch of the proof. **An eligible disjunction occurs positively:** This case requires two instances of the induction hypothesis so we pick up the proof after having copied it.

```
1. S: ESystem
2. 𝓘𝓗
3. 𝓘𝓗
4. ¬∀N∈S.nComplete(N)
5. N: {N:Node| N∈S}
6. ¬nComplete(N)
7. a: Formula
8. b: Formula
9. (a⌈∨⌉b)∈(T(N))
10. ¬a∈(T(N))
11. ¬b∈(T(N))
 ⊢ 𝓒
```

Guided by the $\vee l$-rule, we instantiate the first occurrence of $\mathcal{IH}$ with the system `<a::(T(N)), F(N)>::remove(N;S)`. This results in a well-formedness goal to show that this system is both eligible and lexicographically below `S`.

```
⊢ (<a::(T(N)), F(N)>::remove(N;S)) ∈ {k:ESystem| k < S}
```

This is proved by backchaining through the stipulated lemma `or_1_positive_rule` shown in Appendix A.

Taking up the main line of the proof we see.

```
12. (∃N:{N1:Node| N1∈(<a::(T(N)), F(N)>::remove(N;S))}
         {p:Proof| p proves asSequent(N)})
      ∨ {K:K{i}|
          ∃g:Node ⟶ Σ(K)
            ∀N:{N1:Node| N1∈(<a::(T(N)), F(N)>::remove(N;S))}
               ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                      ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
 ⊢ 𝒞
```

Evidently, by hypothesis (12), there are two cases: either there is a proof of a node in the system; or there is a Kripke counter-example for this system.

We consider the second case first, *i.e.* the case that there is a Kripke model `K` which is counter-example. To complete this branch of the proof there is no call to instantiate the second instance of $\mathcal{IH}$. Having already witnessed the existential in the conclusion with `K`, the proof appears as follows.

```
12. K: K{i}
13. g: Node ⟶ Σ(K)
14. ∀N:{N1:Node| N1∈(<a::(T(N)), F(N)>::remove(N;S))}
     ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
```

```
                  ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
 ⊢ ∃g:Node → Σ(K)
   ∀N:{N:Node| N∈S}
     ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                 ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
```

To eliminate the existential in the conclusion we appeal to the argument given above (in the case an eligible conjunction occurs positively) and supply the witness defined to be the following function term.

```
λn.if (n = N) then g(<a::(T(N)), F(N)>) else g(n) fi
```

This results in two subgoals, both of which are solved by backchaining through the hypotheses. This completes the case where the instantiation of the induction hypothesis with the system `<a::(T(N)), F(N)>::remove(N;S)` induced a Kripke counter-example.

Now we consider the case where splitting the induction hypothesis results in the assertion that a proof exists.

```
12. N1: Node
13. N1∈(<a::(T(N)), F(N)>::remove(N;S))
14. p: Proof
[15]. p proves asSequent(N1)
 ⊢ 𝒞
```

Now, either `N1` is the node `<a::(T(N)), F(N)>`, or it occurs as a member of `remove(N;S)` (hence it was in `S`). Consider the second case first.

If `N1` was already in `S` (but was not `N`) then, since `p` is a proof of `N1`, `N1` and `p` are witnesses for the first disjunct of the conclusion. This is a case where the tableau step did not contribute to the end result.

If `N1` is the node `<a::(T(N)), F(N)>`, then we instantiate the second instance of the induction hypothesis with `<b::(T(N)), F(N)>::remove(N;S)`. This gives the following well-formedness goal.

```
 ⊢ (<b::(T(N)), F(N)>::remove(N;S)) ∈ {k:ESystem| k < S}
```

It is proved by appeal to the stipulated lemma `or_2_positive_rule` shown in Appendix A.

Continuing with the main subgoal resulting from the second instantiation of the induction hypothesis, we see the following.

```
16. (∃N:{N1:Node| N1∈(<b::(T(N)), F(N)>::remove(N;S))}
            {p:Proof| p proves asSequent(N)})
          ∨ {K:K{i}| ∃g:Node ⟶ Σ(K)
              ∀N:{N1:Node| N1∈(<b::(T(N)), F(N)>::remove(N;S))}
                ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                          ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
 ⊢ 𝒞
```

By (16), either there is a proof or a Kripke counter-example. In the case there is a proof, either it is a proof of `<b::(T(N)), F(N)>` or it is a proof of some already existing node of `S`. The second case is proved, as it was above, by providing `N2` and `p` as witnesses.

Instead, assume the proof `p1` is of `<b::(T(N)), F(N)>`, *i.e.*:

```
16. N2: Node
17. N2∈(<b::(T(N)), F(N)>::remove(N;S))
18. p1: Proof
[19]. p1 proves asSequent(N2)
 ⊢ ∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)}
```

We discharge the existential with the witness `N` and, following $\vee l$-rule, witness the proof in the set type by the pre-proof term

```
asSequent(N)\<asSequent(N1),p>,<asSequent(N2),p1>
```

That it is a proof is verified by unfolding the definition and verifying that it does indeed match a rule instance.

Now, suppose instead that there was a Kripke counter-example `K` for the system `<b::(T(N)), F(N)>::remove(N;S))`. We pick up the proof after having witnessed the existential in the conclusion by `K`.

```
16. p proves asSequent(N1)
17. K: K{i}
18. g: Node ⟶ Σ(K)
19. ∀N:{N1:Node| N1∈(<b::(T(N)), F(N)>::remove(N;S))}
      ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
 ⊢ ∃g:Node ⟶ Σ(K)
    ∀N:{N:Node| N∈S}
      ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
```

We supply the following witness to eliminate `g` in the conclusion.

```
λn.if (n = N) then g(<b::(T(N)), F(N)>) else g(n) fi
```

As above, this induces two subgoals, each of which is proved by backchaining through the hypotheses. This complete the proof of the case where an eligible disjunction occurs in the positive part of the eligible node.

So far, we have shown two cases, one requiring one instance of the induction hypothesis, and the other requiring two. The proofs of the remaining cases where a local tableau rule applies are similar to one or the other of these two presented so far. These two cases serve as models for those remaining and so we gloss over the formal details of the proofs in the remaining cases.

An eligible implication occurs positively:

```
8. (a⌈⇒⌉b)∈(T(N))
9. ¬a∈(F(N))
10. ¬b∈(T(N))
 ⊢ 𝒞
```

In the case of a positive implication, the proof requires two instances of the induction hypothesis. Use the system `(<b::(T(N)), F(N)>::remove(N;S))` to instantiate the first instance. The well-formedness goal induced by this proof step is discharged by backchaining through the stipulated lemma `imp_1_positive_rule` shown in Appendix A. The instantiation of the induction hypothesis results either in a node proof pair or a Kripke counter-example. If it is a counter-example, this branch of the proof is done since that counter-example serves for the conclusion as well. If it is a node proof pair of the form `<N1,p1>`, we check to see if `N1` is a member of `remove(N;S)`. If so, then `p1` is a proof of some node in `S` and this branch of the proof is easily completed. Otherwise, `N1` is the node `<b::(T(N)), F(N)>`. In this case a second copy of the induction hypothesis is instantiated, this time with the system `(<T(N), a::(F(N))>::remove(N;S))`. The well-formedness goal for this step is discharged by the stipulated lemma `imp_2_positive_rule` in Appendix A. Again, the instantiated induction hypothesis gives a node proof pair or a Kripke counter-example. As before, the same counter-example serves for `S` completing the proof in that case. If, on the other hand, a node proof pair, say `<N2,p2>` has been returned, then either `N2` is `<T(N), a::(F(N))>` or it is in `remove(N;S)`. In the first case, we construct a proof of `N` by the construction `asSequent(N)\<asSequent(N1),p1>,<asSequent(N2),p2>`. In the second, `N2` is in `S` and `p2` is a proof of `N2` satisfying this goal. This completes the case where an eligible implication occurs positively.

An eligible conjunction occurs negatively:

```
6. a: Formula
7. b: Formula
8. (a⌈∧⌉b)∈(F(N))
9. ¬a∈(F(N))
10. ¬b∈(F(N))
 ⊢ 𝒞
```

This case requires two instances of the induction hypothesis. The first is instantiated with the system `(<(T(N)),a::F(N)>::remove(N;S))`. The well-formedness goal induced by this proof step is discharged by appeal to the stipulated lemma `and_1_negative_rule` shown in Appendix A. The instantiation of the induction hypothesis results either in a node proof pair or a Kripke counter-example. If it is a counter-example, this branch of the proof is done since that counter-example serves for the conclusion as well. If it is a node proof pair of the form `<N1,p1>`, we check to see if `N1` is a member of `remove(N;S)`. If so, then `p1` is a proof of some node in `S` and this branch of the proof is complete. Otherwise, `N1` is the node `<T(N), a::F(N)>`. In this case a second copy of the induction hypothesis is instantiated, this time with the system `(<T(N), b::(F(N))>::remove(N;S))`. The well-formedness goal for this step is discharged by the stipulated lemma `imp_2_positive_rule`. Again, the instantiated induction hypothesis gives a node proof pair or a Kripke counter-example. The counter-example serves for for `S` completing this branch of the proof. If, on the other hand, a node proof pair, say `<N2,p2>` has been returned, then either `N2` is `<T(N), b::(F(N))>` or it is in `remove(N;S)`. In the first case,

```
asSequent(N)\<asSequent(N1),p1>,<asSequent(N2),p2>
```

is a proof of `N` matching then $\wedge r$-rule. In the second, `N2` is in `S` and `p2` is a proof of `N2` satisfying this goal. This completes the case where an eligible conjunction occurs negatively.

An eligible disjunction occurs negatively:

```
 8. (a⌈∨⌉b)∈(F(N))
 9. ¬a∈(F(N)) ∨ ¬b∈(F(N))
  ⊢ 𝒞
```

In the case of a negative occurrence of an eligible disjunction , either `¬a∈(F(N))` or `¬b∈(F(N))`. The cases are symmetrical. In the first the induction hypothesis is instantiated with `(<T(N), a::F(N)>::remove(N;S))`, in the second with `(<T(N), b::F(N)>::remove(N;S))`. The well-formedness goals generated by these instantiations are discharged by the lemmas `or_1_negative_rule` and `or_2_negative_rule` shown in Appendix A. In both cases, the instantiated induction hypotheses assert the existence of either a node proof pair (say `<N1,p1>`) or a Kripke counter-example. For both instantiations, the counter-example refuting the extended system serves to refute `S`, thereby satisfying the goal of the theorem. Now consider the branch of the proof where `p1` is a proof of `N1` and where the node `N1` is either `<T(N), a::F(N)>` or it is in `remove(N;S)`. If `N1` is a member of `remove(N;S)`, then it is a member of `S`. Since `p1` is a proof of `N1`, choosing `N1` and `p1` as the witnesses for the first disjunct of the conclusion completes this branch of the proof. Alternately, if `N1 = <T(N), a::F(N)>` then `asSequent(N)\<asSequent(N1),p1>` is a proof of `N` since it is an instance of the proof rule $\vee r1$-rule. The case for `(<T(N), b::F(N)>::remove(N;S))` is symmetric and matches the $\vee r2$-rule.

This completes the proof of the case where `S` contains an incomplete node. Next we consider the case where all nodes are complete but the system is not system complete.

5.3.2.2 `S` is not system complete

We reiterate the entire proof state at this point.

```
1.  S: ESystem
2. ∀k:{k:ESystem| k < S}
     (∃N:{N:Node| N∈k} . {p:Proof| p proves asSequent(N)})
     ∨ {K:K{i}| ∃g:Node ⟶ Σ(K)
         ∀N:{N:Node| N∈k}
           ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                     ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
3. ∀N∈S.nComplete(N)
4. ¬sComplete(S)
 ⊢ 𝒞
```

Forward chaining through the lemma `not_system_complete` we add the following five hypotheses telling us that there is an eligible negative occurrence of an implication in some node of the system.

```
5. N: {N:Node| N∈S}
6. a: Formula
7. b: Formula
8. (a⌈⇒⌉b)∈(F(N))
9. ∀N':{N':Node| N'∈S} . ¬a∈(T(N')) ∨ ¬b∈(F(N'))
 ⊢ 𝒞
```

Decomposing the induction hypothesis with system `<a::(T(N)),b::[]>::S` yields two subgoals, the first showing that, under the preexisting hypotheses, this system is both an eligible system and is below `S` in the lexicographic ordering.

```
 ⊢ (<a::(T(N)), b::[]>::S) ∈ {k:ESystem| k < S}
```

This subgoal is discharged by backchaining through the stipulated lemma `imp_negative_rule`, which is informally justified by noting that `S` is node complete and so the system `(<a::(T(N)), b::[]>::S)` is eligible, and by appeal to the definitions of the measure `i1`.

The second subgoal, carrying the main thread of the proof, is as follows.

```
2. ∀N∈S.nComplete(N)
3. ¬sComplete(S)
4. N: {N:Node| N∈S}
5. a: Formula
6. b: Formula
7. (a⌈⇒⌉b)∈(F(N))
8. ∀N':{N':Node| N'∈S} . ¬a∈(T(N')) ∨ ¬b∈(F(N'))
9. (∃N:{N1:Node| N1∈(<a::(T(N)), b::[]>::S)} .
           {p:Proof| p proves asSequent(N)})
       ∨ {K:K{i}| ∃g:Node ⟶ Σ(K)
            ∀N:{N1:Node| N1∈(<a::(T(N)), b::[]>::S)}
              ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                          ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
 ⊢ 𝒞
```

Decomposing the disjunction in (10) (and resulting subterms) results in two subgoals: the first asserting that there is a proof of some node `N1` in the extended system `(<a::(T(N)), b::[]>::S)`; and the second asserting that there is a Kripke counter-example for some node in the extended system.

There is a proof of a node in the extended system:

```
9. N1: Node
10. N1∈(<a::(T(N)), b::[]>::S)
11. p: Proof
[12]. p proves asSequent(N1)
  ⊢ ∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)}
```

By hypothesis (10) we know that `N1` is either `<a::(T(N)), b::[]>` or it is some other node already in `S`. If it is a node in `S`, use `N1` and `p` to witness the existentials in the goal. If `N1` is `<a::(T(N)), b::[]>` then we need to construct a proof. First we discharge the existential with `N` (thereby unhiding hidden hypotheses) and yielding the following subgoal.

```
12. p proves asSequent(<a::(T(N)), b::[]>)
 ⊢ {p:Proof| p proves asSequent(N)}
```

We discharge this proof obligation, completing this branch of the proof, by appeal to the following lemma.

```
* THM imp_negative_proof
∀G:Sequent. ∀a,b:Formula.
  (a⌈⇒⌉b)∈Concl(G)
    ⇒ (∀p:Proof. p proves <a::Hyps(G), b::[]>
     ⇒ (∃p':Proof. p' proves G))
```

There is a Kripke counter-example for a node in the extended system:

```
 9. K: K{i}
 [10]. ∃g:Node ⟶ Σ(K)
          ∀N:{N1:Node| N1∈(<a::(T(N)), b::[]>::S)}
            ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                      ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
 ⊢ {K:K{i}| ∃g:Node ⟶ Σ(K)
      ∀N:{N:Node| N∈S} .
        ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                  ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
```

Decomposing the set type in the conclusion with witness `K` unhides hypothesis (10). Decomposing it and then using `g` to witness the second existential, followed by some house-keeping steps, results in the following proof state.

```
 10. g: Node ⟶ Σ(K)
 11. ∀N:{N1:Node| N1∈(<a::(T(N)), b::[]>::S)}
        ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                  ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))
 12. N1: {N:Node| N∈S}
 13. f: Formula
 ⊢ (f∈(F(N1)) ⇒ forces(K,g(N1),f))
    ∧ (f∈(T(N1)) ⇒ not_forces(K,g(N1),f))
```

Backchaining through the hypotheses (and then discharging the well-formedness goal `N1∈(<a::(T(N)), b::[]>::S)`) completes this branch of the proof.

It also completes the case where all nodes in the system are complete, but the system was not complete. In the remaining case, all nodes in the system are complete as is the system.

This completes the proof of the theorem.

5.3.3 The Base Case

The base case of the decidability theorem occurs when both the node and the system are complete. Then, either there is a proof of some node in the system, or there is a Kripke counter-example for the system. Thus, it is in the base case that the initial proofs and counter-examples are constructed.

The overall strategy for the proof is to check if there are any nodes in `S` that, when viewed as sequents, are instances of one of the axioms. If not, it is shown that the system encodes a Kripke counter-example. Above, we presented a mapping `K:System →Kripke`. The main body of this proof is to show that for eligible systems `S` not containing any instances of axioms, `K(S)` is in fact a Kripke counter-example for `S`.

Recall the statement of the lemma.

```
* THM decidability_base
∀S:ESystem
    sComplete(S)
    ⇒ ∀N∈S.nComplete(N)
    ⇒ (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)})
       ∨ {K:K{i}| ∃g:Node → Σ(K)
∀N:{N:Node| N∈S}
  ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
            ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
```

First we decide if there is any node in the system having a non-empty intersection among those formulas it assumes true and those it assumes false. We pick up the proof after having executed the `Decide` tactic on the following term.

```
∃N:{N:Node| N∈S}.T(N)∩F(N)
```

Here, `M∩N` is a proposition that is true when the lists `M` and `N` share a common element. Clearly the existence of such a node is a decidable property and the proof obligation to show that is automatically discharged by the system based on the lemmas `decidable__exists` and `decidable__is_intersection`.

This step of the proof yields two subgoals, the first being

```
 1. S: ESystem
 2. sComplete(S)
 3. ∀N∈S.nComplete(N)
 4. N:{N:Node| N∈S}
 5. T(N)∩F(N)
⊢ (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)})
  ∨ {K:K{i}|
     ∃g:Node → Σ(K)
      ∀N:{N:Node| N∈S} . ∀f:Formula.
        (f∈(T(N)) ⇒ forces(K,g(N),f)) ∧
        (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
```

In this case, there is a node `N` in the system that, when viewed as a sequent, is an instance of an axiom, *i.e.* some hypothesis is among the conclusions. We proceed by choosing to prove the first disjunct of the conclusion using N as the witness for the node and the pre-proof term (asSequent(N)\) to witness the proof in the set type. The proof that this term is indeed a proof of N is trival.

To save space below, we will display the goal of this sequent as $\mathcal{C}$ from here on. Continuing with the proof, consider the case where there is no node that is an axiom of this form.

```
4. ¬(∃N:{N:Node| N∈S} . T(N)∩F(N))
⊢ 𝒞
```

Now we decide if there is a node N in S containing ⌜false⌝ among T(N), *i.e.* we decide whether there is an instance of the constant ⌜false⌝ among the nodes of S. This results in two subgoals, the first being:

```
4. ¬(∃N:{N:Node| N∈S} . T(N)∩F(N))
5. ∃N:{N:Node| N∈S} . ⌜false⌝∈T(N)
⊢ 𝒞
```

In this case there is a node N in S that is an axiom. Using N as a witness and a pre-proof term of the form (asSequent(N)\) discharges the first conjunct of the conclusion. This case is concluded by arguing that this pre-proof term is indeed a well-formed proof of N by showing it is an instance of the axiom rule.

In the other case, ⌜false⌝ is not among the hypotheses of the nodes of S; we conclude that S contains no instances of axioms in S.

```
4. ¬(∃N:{N:Node| N∈S} . T(N)∩F(N))
5. ¬(∃N:{N:Node| N∈S} . ⌜false⌝∈T(N))
⊢ 𝒞
```

To complete the proof we must construct a Kripke counter-example from `S`, and so choose to prove the the second disjunct of the conclusion. We use the Kripke model `K(S)` as witness for the set type. We also know there is some node `N` such that `N∈S` (since systems are non-empty). After these steps the proof appears as follows.

```
6. N: Node
7. N∈S
⊢ ∃g:Node ⟶ Σ(K(S))
   ∀N:{N:Node| N∈S} . ∀f:Formula.
     (f∈T(N) ⇒ forces(K(S),g(N),f)) ∧
     (f∈F(N) ⇒ not_forces(K(S),g(N),f))
```

Eliminating the existential with the function mapping nodes of `S` to themselves, and all others to `N`, results in the following (main) subgoal (after having reordered the universal quantifiers and stripping one off).

```
8. f: Formula.
⊢ ∀N':{N:Node| N∈S}
   (f∈(T(N')) ⇒ forces(K(S),(λx.if x∈S then x else N fi)(N'),f))
   ∧ (f∈(F(N')) ⇒
        not_forces(K(S),(λx.if x∈S then x else N fi)(N'),f))
```

From here the proof is by induction on the structure of the formula `g`.

5.4 Remarks on the proof and Extract

It should be remarked that this particular structuring of sequents, as pairs of lists of the form `<M@f::N,c>` or `<h,M@f::N>`, has proved itself well suited to implementation in Nuprl. It avoids the problems associated with formula ordering in the parts of the sequent: in our proof we manage to avoid all reasoning about permutation equivalence of lists that might otherwise be required. In system *MJ*, multiplicity is not at issue because, unlike the classical case, the premises of all the rules (except the $\Rightarrow r$ rule) are monotonic in their conclusions, *i.e.* the conclusions of the rules are sub-sequents of their hypotheses. However, the only rule for which the principle formula *must* be maintained in a premise is in the left premise of the $\Rightarrow l$ rule.

The proofs produced here are in a multi-succedent calculus allowing multiple formulas on the right. To incorporate the extracted procedure into Nuprl, either via reflection, or by running the procedure and returning a tactic justification, the multi-succedent calculus proofs must be transformed into the single succedent calculus implemented by the Nuprl rules. Egly and Schmidtt [ES98] give cut-free translations of multi-succedent proofs into single succedent proofs preserving reasonable extracts.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{}{\mathtt{P},(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash \mathtt{P},(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)),\bot}\,(\boldsymbol{Ax})
        }{\mathtt{P},(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash (\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)),\bot}\,(\vee r1)
        \qquad
        \dfrac{}{\bot,\ \mathtt{P},\ (\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash \bot}\,(\boldsymbol{Ax})
      }{\mathtt{P},(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash \bot}\,(\Rightarrow l)
    }{(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash \mathtt{P} \Rightarrow \bot,\ ((\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)),\bot}\,(\Rightarrow r)
  }{(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash ((\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)),\ \bot}\,(\vee r2)
  \qquad
  \dfrac{}{\bot,(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash \bot}\,(\boldsymbol{Ax})
}{
  \dfrac{(\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot \vdash \bot}{\vdash ((\mathtt{P} \vee (\mathtt{P} \Rightarrow \bot)) \Rightarrow \bot) \Rightarrow \bot}\,(\Rightarrow r)
}\,(\Rightarrow l)
$$

Figure 5.2: A Proof of $\neg\neg(P \vee \neg P)$ in system MJ

Chapter 6

Conclusion

The work reported on here represents the early stages of an applied research program devoted to the synthesis of programs by extraction from constructive proofs. This is research that has many unexplored, yet important, topics.

The extract resulting from the proof of intuitionistic decidability presented in proof in Chapter 5 is not as clear as it could be. Careful study of the extracted program reveals that there is room for the introduction of abstractions which would make the extracted program clearer and would result in a shorter proof. Also, it would be interesting and reasonably easy to modify the current proof to implement Dyckhoff's contraction-free system.

The integration into Nuprl of the extracted decider for intuitionistic propositions is an immediate goal. The extracted program can easily be re-coded in ML as as part of a tactic to decide propositional fragments of Nuprl's type theory. The resulting tactic would fail, returning the Kripke model as evidence against the validity of a formula should it turn out not to be valid; alternatively, it would use the formal proof returned by the procedure to construct a Nuprl tactic, which it could then apply to discharge the goal.

Another line of development that should eventually be explored is the reflection of this decision procedure into Nuprl. The reflection work [ACHA90, ACU] was the motivation for the proof outlined in [ACU]. In [Har95] Harrison argued for the kind of integration proposed above (as a tactic) and against reflection as being unnecessary; until the experiment is performed, this question cannot really be answered.

In [Und94] Underwood shows how the tableau proof of intuitionistic decidability can be extended to a semi-decision procedure for the first order case by using co-inductive types. In [Men88] Mendler presents the rules for co-inductive types but they need to be implemented, this is an ideal application for them.

The Nuprl system supports an intersection type (see Chapter 2) which can be seen to be the dual to the set type. The author has experimented with the intersection type as a means to further eliminate unwanted computational content, and it can be used to eliminate layers of lambda abstraction from extract terms whenever the abstracted variable does not occur in the body of the term. Better and more complete integration of the set type and the intersection type with the existing tactic library holds great promise for achieving better extracts with less effort in the future.

Currently, it is not entirely trivial to translate programs extracted form Nuprl proofs into other languages. The Nuprl extracts are terms of the Nuprl computation system which has lazy evaluation semantics. For the purposes of program extraction, the development of methodology to extract programs guaranteed to terminate under an eager semantics is an interesting research topic. One approach would be to develop a set of eager proof rules. A refiner extracting programs in Scheme or perhaps Boyer and Moore's ACL2 should be quite easy and would be a very useful addition to the program extraction tool-box.

In joint work with Gent and Underwood [CGU99], the author applied Nuprl to synthesize a sophisticated search algorithm known in the literature as Conflict-directed backjump [Pro93]. The system applied to the problem was a classical extension of Nuprl. The classical system provided the means to extract a non-local control operator, Scheme's `call-cc`. The successful results of that project suggest further developments along that path.

Appendix A

Stipulated Lemmas

The following stipulated lemmas characterize the tableau construction.

```
*THM and_1_positive_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌜∧⌝b)∈(T(N)) ⇒ ¬(a∈(T(N))) ⇒
    (<a::(T(N)), F(N)>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM and_2_positive_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌜∧⌝b)∈(T(N)) ⇒ ¬b∈(T(N)) ⇒
    (<b::(T(N)), F(N)>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM or_1_positive_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌜∨⌝b)∈(T(N)) ⇒ ¬a∈(T(N)) ⇒
    (<a::(T(N)), F(N)>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}
```

```
*THM or_2_positive_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈∨⌉b)∈(T(N)) ⇒ ¬b∈(T(N)) ⇒
    (<b::(T(N)), F(N)>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM imp_1_positive_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈⇒⌉b)∈(T(N)) ⇒ ¬b∈(T(N)) ⇒
    (<b::(T(N)), F(N)>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM imp_2_positive_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈⇒⌉b)∈(T(N)) ⇒ ¬a∈(F(N)) ⇒
    (<T(N), a::(F(N))>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM and_1_negative_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈∧⌉b)∈(F(N)) ⇒ ¬a∈(F(N)) ⇒
    (<T(N), a::(F(N))>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM and_2_negative_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈∧⌉b)∈(F(N)) ⇒ ¬b∈(F(N)) ⇒
    (<T(N), b::(F(N))>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM or_1_negative_rule
```

```
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈∨⌉b)∈(F(N)) ⇒ ¬a∈(F(N)) ⇒
    (<T(N), a::(F(N))>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM or_2_negative_rule
 ∀S:ESystem. ¬∀N∈S.nComplete(N) ⇒
  ∀N:{N:Node| N∈S}. ¬nComplete(N) ⇒
   ∀a,b:Formula. (a⌈∨⌉b)∈(F(N)) ⇒ ¬b∈(F(N)) ⇒
    (<T(N), b::(F(N))>::remove(=₂;N;S)) ∈ {k:ESystem| k <(i1,i2) S}

*THM imp_negative_rule
 ∀S:ESystem ∀N∈S. nComplete(N) ⇒ ¬sComplete(S) ⇒
  ∀N:{N:Node| N∈S}
   ∀a,b:Formula . (a⌈⇒⌉b)∈(F(N)) ⇒
    ∀N':{N':Node| N'∈S} . ¬a∈(T(N')) ∨ ¬b∈(F(N'))) ⇒
    (<a::(T(N)), b::[]>::S) ∈ {k:ESystem| k <(i1,i2) S})
```

Appendix B

Extract of intuitionistic decidabilty proof

This transformed program is equivalent to the program extracted from the intuitionistic decidabilty proof.

```
λS.(letrec tableau(S) =
  if ∀N∈S.nComplete(N) then
    if sComplete(S) then
      ext{decidability_base}(S)(·)(·)
    else let <N,a,b,mp,_> = (ext{not_system_complete}(S)(·)) in
            case tableau(<a::T(N), b::[]>::S)
            of inl(<N1,p1>) =>
              inl(if (N1 = <a::T(N), b::[]>)
                  then <N, let <p',_> =
                             (ext{imp_right_proof}
                               (N)(a)(b)(mp)(p1)(Ax)) in p'>
                  else <N1, p1>)
            | inr(K) => inr(K)
```

```
else
  let <N,t> = (∃N:{N:Node | N∈S}. ¬nComplete(N))
    let <a,b,op_type> = (ext{not_node_complete}(N)(t)) in
      case op_type of inl(<_,V14>) => case V14
         of inl(_) =>
           case tableau(<a::T(N), F(N)>::remove(N;S))
           of inl(<N1,p1>) =>
             inl(if (N1 = <a::T(N), F(N)>)
                 then <N, mk_proof(N,<N1,p1>)>
                 else <N1, p1>)
           | inr(K) => inr(K)
         | inr(_) =>
             case tableau(<b::T(N), F(N)>::remove(N;S))
             of inl(<N1,p1>) =>
               inl(if (N1 = <b::T(N), F(N)>)
                   then <N, mk_proof(N,<N1,p1>)>
                   else <N1, p1>)
             | inr(K) => inr K
      | inr(V13) => case V13
         of inl(_) =>
           case tableau(<a::T(N), F(N)>::remove(N;S))
           of inl(<N1,p1>) =>
               if (N1 = <a::T(N), F(N)>) then
                 case tableau(<b::T(N), F(N)>::remove(N;S))
                 of inl(<N2,p2>) =>
                   inl(if (N2 = <b::T(N), F(N)>)
                        then <N, mk_proof(N,<N1,p1>,<N2,p2>)>
                        else <N2, p2>)
                 | inr(K) => inr(K)
               else  inl(<N1, p1>)
           | inr(K) => inr(K)
         | inr(V15) => case V15
```

```
of inl(_) =>
  case tableau(<b::T(N),F(N)>::remove(N;S))
  of inl(<N1,p1>) =>
    if (N1 = <b::T(N), F(N)>) then
      case tableau(<T(N), a::F(N)>::remove(N;S))
      of inl(<N2,p2>) =>
        inl(if (N2 = <T(N), a::F(N)>)
            then <N, mk_proof(N,<N1,p1>,<N2,p2>)>
            else <N2, p2>)
      | inr(K) => inr(K)
    else  inl(<N1, N2>)
  | inr(K) => inr(K)
| inr(V17) => case V17
    of inl(_) =>
      case tableau(<T(N), b::F(N)>::remove(N;S))
      of inl(<N1,p1>) =>
        if (N1 = <T(N), b::F(N)>) then
          case tableau(<T(N), a::F(N)>::remove(N;S))
          of inl(<N2,p2>) =>
            inl(if (N2 = <T(N), a::F(N)>)
              then <N, mk_proof(N,<N2,p2>,<N1,p1>)>
              else <N2, p2>)
          | inr(K) => inr(K)
        else inl(<N1, p1>)
      | inr(K) => inr(K)
    | inr(V19) => let <_,V21> = V19 in
          case V21
          of inl(_) =>
            case
             tableau(<T(N), a::F(N)>::remove(N;S))
            of inl(<N1,p1>) =>
              inl(if (N1 = <T(N), a::F(N)>)
                            then  <N, mk_proof(N,<N1,p1>)>
                            else <N1, p1>)
                      | inr(K) => inr(K)
                    | inr(_) =>
                        case tableau(<T(N), b::F(N)>::remove(N;S))
                        of inl(<N1,p1>) =>
                          inl(if (N1 = <T(N), b::F(N)>)
                              then <N, mk_proof(N,<N1,p1>)>
                              else <N1, p1>)
                        | inr(K) => inr(K)
) (S)
```

BIBLIOGRAPHY

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.

[ACU] William Aitken, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. To Appear *Journal of Symbolic Computation*.

[All87a] Stuart F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1987.

[All87b] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[Avr93] Arnon Avron. Gentzen-type systems, resolution and tableaux. *Journal of Automated Reasoning*, 10(2):265–281, April 1993.

[Bar81] Henk P. Barendregt. The lambda calculus: its syntax and semantics. In *Studies in Logic*, volume 103. Amsterdam:North-Holland, 1981.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[BC85] J.L. Bates and Robert L. Constable. Definition of micro-PRL. Technical Report 82–492, Cornell University, Computer Science Dept., Ithaca, NY, 1985.

[Bet59] E. W. Beth. *The Foundations of Mathematics*. North-Holland, 1959.

[BHP95] Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors. *Theorem Proving with Analytic Tableaux and Related Methods*, volume 918 of *Lecture Notes in Artificial Intelligence*. Springer, 1995.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[Cal97] James Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference Automated Software Engineering*. IEEE Computer Society, 1997.

[CC98] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In *Preprint*, 1998.

[CCF$^+$95] Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Muñoz, Chetan Murthy, Catherine Parent, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1995.

[CGU99] James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. To Appear in: *Theoretical Computer Science*, 1999. Special Issue on Proof Search in Type Theoretic Languages.

[CH85] Thierry Coquand and G. P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL '85, Lecture Notes in Computer Science, Vol. 203*. Springer-Verlag, 1985.

[CH88a] Thierry Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CH88b] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CH90] Robert L. Constable and Douglas 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.

[CM85] Robert L. Constable and N.P. Mendler. Recursive definitions in type theory. In *Proceedings of the Logics of Programming Conference*, pages 61–78, January 1985. Cornell TR 85–659.

[Con71] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[Con83] Robert L. Constable. Mathematics as programming. In *Proceedings of the Workshop on Programming and Logics, Lectures Notes in Computer Science 164*, pages 116–128. Springer-Verlag, 1983.

[Con85] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Annals of Mathematics, Vol. 24*, pages 21–37. Elsevier Science Publishers, B.V. (North-Holland), 1985. Reprinted from *Topics in the Theory of Computation*, Selected Papers of the International Conference on Foundations of Computation Theory, FCT '83.

[Con86] Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CT98] R. Constable and The Nuprl Group. The Nuprl web pages. `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`, 1998.

[CU96] James Caldwell and Judith Underwood. Classical tools for constructive proof search. In Didier Galmiche, editor, *Proceedings of the CADE-13 Workshop on Proof search in type-theoretic languages.*, Rutgers N.J., July 1996.

[CZ84] Robert L. Constable and D.R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[deB70] N. G. deBruijn. The mathematical language Automath: Its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration,* Lecture Notes in Mathematics, Vol. 125, pages 29–61. Springer-Verlag, 1970.

[DFH$^+$93a] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide.* INRIA, Version 5.8, 1993.

[DFH$^+$93b] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner.

The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[Dra87] A. G. Dragalin. *Mathematical Intuitionism: Introduction to Proof Theory*, volume 67 of *Translations of Mathematical Monographs*. American Mathematical Society, 1987.

[DS77] M. Davis and J. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. Technical Report 12, Courant Institute of Mathematical Sciences, New York, 1977.

[dS98] H. de Swart, editor. *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1397 of *Lecture Notes in Artificial Intelligence*. Springer, 1998.

[Dum77] Michael 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.

[ES98] Uwe Egly and Stephan Schmitt. Intuitionistic proof transformations and their application to constructive program synthesis. In *Proceedings of the Fourth International Conference on Artificial Intelligence and Symbolic Computation, AISC'98*, Plattsburg, N.Y., 1998.

[Fef79] Solomon Feferman. Constructive theories of functions and classes. In *Logic Colloquium '78*, pages 159–224. North Holland,, 1979.

[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, 1983.

[Gal86a] J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper and Row, 1986.

[Gal86b] J. H. Gallier. *Logic for Computer Science, Foundations of Automatic Theorem Proving*. Harper and Row, NY, 1986.

[Gal97] D. Galmiche, editor. *Theorem Proving with Analytic Tableaux and Related Methods*, number 1227 in Lecture Notes in Artificial Intelligence. Springer, 1997.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969. Originally published 1935.

[Gir86] J-Y. Girard. The system F of variable types: Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

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

[Göd65] Kurt Gödel. On intuitionistic arithmetic and number theory. In Davis, M., editor, *The Undecidable*, pages 75–81. Raven Press, 1965.

[Har60] R. Harrop. Concerning formulas of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)B(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, 25(1):27–32, March 1960.

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[Hay94] S. Hayashi. Singleton, union, and intersection types for program extraction. *Information and Computation*, 109:174–210, 1994.

[Hed91] M. Hedberg. Normalizing the associative law: An experiment with Martin-Löf's type theory. *Formal Aspects of Computing*, 3:218–252, 1991.

[HN88] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic.* Foundations of Computing. MIT Press, Cambridge, MA, 1988.

[How88] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.

[How93] Douglas J. Howe. Reasoning about functional programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, Berlin, 1993. Springer Verlag.

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

[Hud97] J. Hudelmaier. A note on Kripkean countermodels for intuitionistically unprovable sequents. In W. Bibel, U. Furbach, R. Hasegawa, and M. Stickel, editors, *Seminar on Deduction*, February 1997. Dagstuhl report 9709.

[Jac95a] Paul Jackson. The Nuprl proof development system, version 4.2 reference manual and user's guide. Computer Science Department, Cornell University, Ithaca, N.Y. Manuscript available at `http://www.cs.cornell.edu/Info/Projects/NuPrl/manual/it.html`, July 1995.

[Jac95b] Paul B. Jackson. *Enhancing the Nuprl proof development system and applying it to computational abstract algebra.* PhD thesis, Cornell University, 1995.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics.* van Nostrand, Princeton, 1952.

[Kri63] S. Kripke. Semantical analysis of modal logic I. *Zeit. für Math. Logic u. Grund. der Math.*, 9:67–96, 1963.

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

[Les81] J. Leszczylowski. An experiment with Edinburgh LCF. In W. Bibel and R. Kowalski, editors, *5th International Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 170–181, New York, 1981. Springer-Verlag.

[LP92] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[LSS91] P. Lincoln, A. Scedrov, and N. Shankar. Linearizing intuitionistic implication. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 51 – 62, Amsterdam, Netherlands, 1991. IEEE Computer Society Press.

[Luo89] Zhaohui Luo. ECC, an extended calculus of construction. In *Proceedings of the 4th Symposium on Logic in Computer Science*, pages 385–395, Pacific Grove, CA, June 1989.

[Luo94] Zhaohui Luo. *Computation and Reasoning, A Type Theory for Computer Science.* Oxford University Press, New York, 1994.

[Mag95] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[Men79] Elliott Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand, second edition, 1979.

[Men88] P.F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, Ithaca, NY, 1988.

[ML73] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[MMMO96] P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors. *Theorem Proving with Analytic Tableaux and Related Methods*, volume 1071 of *Lecture Notes in Artificial Intelligence.* Springer, 1996.

[MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In Springer-Verlag, editor, *Types for Proofs and Programs*, volume 806 of Lecture Notes in Computer Science, pages 213–237, 1994.

[Moh86] Christine Mohring. Algorithm development in the Calculus of Constructions. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 84–91. IEEE, 1986.

[MW90] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming: Volume II: Deductive Systems.* Addison Wesley, 1990.

[Nor81] Bengt Nordstrom. Programming in constructive set theory: Some examples. In *Proceedings 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 290–341. Portsmouth, England, 1981.

[Nor93] Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.

[NPS90] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

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

[OS95] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Computer Science Laboratory, SRI International, Menlo Park, CA. Draft Manuscript available at `http://www.csl.sri.com/~shankar/shankar-drafts.html`, October 1995.

[Pau86a] Lawrence Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[Pau86b] Lawrence Paulson. Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, 2:63–74, 1986.

[PD95] L. Pinto and R. Dyckhoff. Loop-free construction of counter-models for intuitionistic propositional logic. In *Symposia Gaussiana*, pages 225–232, Berlin, New York, 1995. Walter de Gruyter and Co.

[PM89] Christine Paulin-Mohring. Extracting $F_w's$ programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 89–104, 1989.

[PMW93] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.

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

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, pages 268–299, 1993.

[Sal89] Anne Salvesen. On specifications, subset types and interpretation of propositions in type theory. In *Proceedings of the Workshop on Programming Logic*, pages 209–230, Båstad, Sweden, May 1989. Programming Methodology Group, University of Göteborg and Chalmers University of Technology.

[Sco70] D. Scott. Constructive validity. In D. Lacombe M. Laudelt, editor, *Symposium on Automatic Demonstration*, volume 5(3) of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, New York, 1970.

[Sha85] N Shankar. Towards mechanical metamathematics. *J. Automated Reasoning*, 1(4):407–434, 1985.

[Smi89] S.F. Smith. *Partial Objects in Type Theory.* PhD thesis, Cornell University, Ithaca, NY, 1989.

[Smo73] C. A. Smorynski. Applications of Kripke models. In Troelstra [Tro73], pages 324–391.

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

[SS87] Anne Salvesen and Jan M. Smith. The strength of the subset type in intuitionistic type theory. In *Proceedings of the Workshop on Programming Logic*, pages 327–332, Marstrand, Sweden, October 1987. Programming Methodology Group, University of Göteborg and Chalmers University of Technology.

[SS88] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 384–391, Edinburgh, Scotland, 5–8 July 1988. IEEE Computer Society.

[Thé98] Laurent Théry. A Certified Version of Buchberger's Algorithm. In H. Kirchner and C. Kirchner, editors, *15th International Conference on Automated Deduction*, LNAI 1421, pages 349–364, Lindau, Germany, July 5–July 10, 1998. Springer-Verlag.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.

[Tho92] Simon Thompson. Are subsets necessary in Martin-Lof type theory? In J. P. Myers Jr. and M. J. O'Donnell, editors, *Constructivity in Computer Science*, volume 613 of *Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, January 1992.

[Tro73] A. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Mathematics*, volume 344 of *Lecture Notes in Mathematics.* Springer-Verlag, 1973.

[TvD88] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction, Vol. I.* North-Holland, Amsterdam, 1988.

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

[Und93] Judith 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.

[Und94] J. Underwood. *Aspects of the Computational Content of Proofs.* PhD thesis, Cornell University, 1994.

[Und95] Judith Underwood. Tableau for intuitionistic predicate logic as metatheory. In Baumgartner et al. [BHP95].

[vD94] D. van Dalen. *Logic and Structure.* Springer-Verlag, third edition, 1994.

[Vor52] N. N. Vorob'ev. The derivability problem in the constructive propositional calculus with strong negation. *Doklady Akademii Nauk SSSR*, 58:689–692, 1952. (In Russian.).

[Vor70] N. N. Vorob'ev. A new algorithm for derivability in the constructive propositional calculus. *Translations of the American Mathematical Society*, 94(2):37–71, 1970.

[Wal90] L. A. Wallen. *Automated Deduction in Non-Classical Logics.* MIT Press, 1990.

[Wei98a] K. Weich. Decision procedures for intuitionistic propositional logic by program extraction. In de Swart [dS98], pages 292–306.

[Wei98b] K. Weich. Private communication. February 1998.