Extracting Constructive Content from Classical Proofs

Chetan Murthy
Ph.D Thesis

90-1151
August 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
Extracting Constructive Content From Classical 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

Chetan Murthy

August 1990
Extracting Constructive Content From Classical Proofs

Chetan Murthy, Ph.D.
Cornell University 1990

This thesis is concerned with the relationship between classical and constructive mathematics. It is well-known that in many constructive logics, we can interpret mathematical sentences as program specifications, and we can interpret a constructive proof of such a sentence as a program which meets this specification. It is also well-known that many classical logics do not have this property, as shown by Brouwer's counterexamples to some theorems of analysis. Kreisel and Friedman showed that for certain classes of sentences (\(\Pi_2^0\)), the classical theories conservatively extend their constructive counterparts, and furthermore give effective translations from classical proofs to constructive proofs.

This thesis consists of two parts. In the first, we describe our implementation of Friedman's translation results, and their use in translating Higman's Lemma, a nontrivial theorem of combinatorics. To do this, we delineate a subtheory of a constructive type theory (Nuprl) for which Friedman's translation is guaranteed to succeed. We also extend the Nuprl type theory with impredicative \(\Pi\)-quantification, and use this to provide a classical proof of Higman's Lemma, which we go on to mechanically translate to a constructive proof.

In the second part we discuss connections that we have discovered between Friedman's translation and a standard compilation technique, continuation-passing-style (CPS) translation. We demonstrate that a classical proof of a \(\Pi_2^0\) sentence \(\Phi\) is a program which meets the specification \(\Phi\). We demonstrate that we can consistently give algorithmic content to the only constructively problematic rule of classical logic, the rule of double-negation elimination. This algorithmic content is the nonlocal control operator \(C\) (a relative of call-with-current-continuation). Moreover, we show that Friedman's translation is exactly a CPS-translation on the classical "program" (with \(C\)), converting it into a pure functional program (without \(C\)).

Our work provides a semantic account of Friedman's translation, in terms of its effect on programs, making the connections (and the differences) between classical and constructive systems clearer and more precise. Moreover, we provide the first steps towards integrating nonlocal control operators into a type-theoretic explanation of computation.
Biographical Sketch

Chet Murthy was born in Bangalore, India, on January 11, 1965. In May 1986, he received a Bachelor of Science in Electrical Engineering degree from Rice University. In September 1989, he received a Master of Science degree from Cornell University. In May 1989, he bench-pressed 200 pounds.
To my mother.
Acknowledgements

This thesis would never have come about without the help of so many people. David Basin, Prakash Panangaden and Tim Teitelbaum started it all off by encouraging me to get involved with Nuprl in the first place. Doug Howe, the big brother of the Nuprl group, helped me uncountably many times, and his insight into problems is much-appreciated. James Russell, my officemate for four years, was a great role model as a person, and a great collaborator, idea-evaluator, and friend. Dean Krafft and Doug Flanagan worked hard to get me the CPU cycles I needed. Chapter 5 is a simple extension of Nax Mendler’s thesis, and his advice and assistance was critical to my being able to push thru the results therein. I benefited from conversations with Tim Griffin and Gabriel Stolzenberg. Elizabeth Maxwell and Jan Batzer were always there to help with things that I knew nothing about, and were of life-and-death importance, like dealing with the administration of Cornell. Tim Teitelbaum and Bard Bloom provided very useful comments on drafts of this thesis. Bill Aitken spent many long hours with me contemplating the deep meaning of various constructivist slogans and ideas. The National Science Foundation and the Office of Naval Research gave me a fellowship and research support while I did this research, for which I am very grateful. Finally, and most importantly, I have my advisor, Robert Constable, to thank for getting me to this place. It is said that graduates from the Nuprl group always yearn to return to the environment from which they came - to the womb which they left to make their way in the world. And with good reason. He is a good friend, a caring man, and most certainly the best teacher I have ever had.

This thesis was typeset with Leslie Lamport’s LaTeX package for Donald Knuth’s TeX typesetter, and used Paul Taylor’s diagrams package.
# Table of Contents

1 Introduction ............................................. 1
   1.1 Reasoning about Computation .................. 2
   1.2 Functional Programming and Constructive Reasoning .... 4
   1.3 Nonfunctional Programming and Nonconstructive Reasoning ... 5
      1.3.1 The Research Contributions of This Work .......... 11
      1.3.2 Comparisons with Other Work ................ 12
   1.4 An Overview ...................................... 13

2 Motivation: A Description of Higman's Lemma ............ 16
   2.1 Preliminaries .................................... 16
   2.2 A Classical Proof of the Lemma .................. 16
      2.2.1 The Main Proof .............................. 17
   2.3 An Analysis of the Proof ......................... 18

3 The Nuprl System ...................................... 20
   3.1 The Nuprl Type Theory ........................... 20
      3.1.1 Terms, Typehood, and Evaluation ............ 20
      3.1.2 Evidence and Judgments .................... 24
      3.1.3 An Example Proof .......................... 26
   3.2 The Nuprl Development Environment ............... 27
      3.2.1 Tactics .................................. 27
      3.2.2 Notation ................................ 28
      3.2.3 Libraries ................................ 28
      3.2.4 The MetaLanguage ......................... 28
   3.3 The Nonarithmetic Reduction Rules ................ 29

4 The Formalization of Higman's Lemma in Nuprl ............ 30
   4.1 Preliminaries .................................. 30
   4.2 The First Classical Proof ...................... 31
   4.3 What's Wrong with the Axiom of Choice? .......... 33
   4.4 The Final Classical Proof ...................... 34
   4.5 A Retrospective ................................ 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>A Proof of Consistency of Impredicative II-Types in Nuprl</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Motivation</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>Extending Mendler’s Work</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Semantics of Predicative Nuprl</td>
<td>41</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Ground Relations</td>
<td>41</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Type Systems</td>
<td>42</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Truth and Soundness</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>Simple Impredicative II-Types</td>
<td>45</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Ground Relations and Ground Types</td>
<td>46</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Type Systems</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Parameterized Impredicative II-Types</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>The Translations on Proofs and Types</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>The Double-Negation Translation</td>
<td>55</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Other Properties of Double-Negation Translations</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>The Friedman A-Translation</td>
<td>57</td>
</tr>
<tr>
<td>6.3</td>
<td>Putting It All Together</td>
<td>58</td>
</tr>
<tr>
<td>7</td>
<td>Justifying the Translations in Nuprl</td>
<td>60</td>
</tr>
<tr>
<td>7.1</td>
<td>An Overview of the Translation Results</td>
<td>61</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Preconditions for Translation</td>
<td>62</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Translation and Conservative Extension</td>
<td>64</td>
</tr>
<tr>
<td>7.2</td>
<td>Classical Nuprl</td>
<td>65</td>
</tr>
<tr>
<td>7.3</td>
<td>Propositions vs. Types</td>
<td>65</td>
</tr>
<tr>
<td>7.4</td>
<td>Nuprl^\circ Defined</td>
<td>67</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Membership and Nuprl^\circ Proofs</td>
<td>68</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Proposition-hood</td>
<td>69</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Nuprl^\circ At Last</td>
<td>73</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Technical Results Concerning Nuprl^\circ</td>
<td>74</td>
</tr>
<tr>
<td>7.5</td>
<td>Double-Negation Translation on Terms, Sequents, and Proofs</td>
<td>75</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Some Results about Double-Negation Translation</td>
<td>75</td>
</tr>
<tr>
<td>7.5.2</td>
<td>The Translation on Proofs</td>
<td>79</td>
</tr>
<tr>
<td>7.6</td>
<td>A-Translation in Nuprl^\circ</td>
<td>85</td>
</tr>
<tr>
<td>7.6.1</td>
<td>A-Translating Terms and Sequents</td>
<td>85</td>
</tr>
<tr>
<td>7.6.2</td>
<td>Some Results about A-Translation</td>
<td>85</td>
</tr>
<tr>
<td>7.6.3</td>
<td>A-Translation on Proofs</td>
<td>87</td>
</tr>
<tr>
<td>7.7</td>
<td>Extensions and Implementation</td>
<td>89</td>
</tr>
<tr>
<td>7.8</td>
<td>Summary</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>The Implementation of Proof Translations in Nuprl</td>
<td>91</td>
</tr>
<tr>
<td>8.1</td>
<td>The Translation Project</td>
<td>91</td>
</tr>
<tr>
<td>8.2</td>
<td>A High Level Overview of Proof Translation</td>
<td>92</td>
</tr>
<tr>
<td>8.3</td>
<td>Comparisons with Other Theorem-Proving Systems</td>
<td>93</td>
</tr>
</tbody>
</table>
8.4 The Actual Proof Translation Experiment ........................................ 94
8.5 Computational Metatheory and Efficient Translation ......................... 95
8.6 Exploiting Gratuitous Parallelism .................................................. 95
8.7 Efficient Translation of Tactic Expressions ...................................... 96

9 Double-Negation, Compilation, and Continuations .............................. 98
9.1 Overview of The Next Two Chapters .............................................. 98
9.2 Continuations, Evaluation, and CPS Compilation ............................... 100
  9.2.1 Continuations Throughout History ........................................ 101
9.3 CPS-Compilation, Semantic Equivalence, and Definitional Interpreters 102
9.4 Connections With Our Work ......................................................... 103
9.5 A Tractable Example Theory ....................................................... 105
  9.5.1 The Programming Language .................................................. 105
  9.5.2 The Constructive Logic ....................................................... 106
  9.5.3 Extending to Classical Logic ................................................. 106
  9.5.4 Some Simple Results .......................................................... 107
  9.5.5 CPS-Translation and Double-Negation Translation for Constructive Witnesses ...................................................... 107
  9.5.6 Semantic Equivalence for $LJ \rightarrow$ .................................... 110
  9.5.7 The CPS-Translation for Classical Witnesses ............................. 113
  9.5.8 Semantic Equivalence for $LK \rightarrow$ .................................... 114
  9.5.9 Summing Up This Trivial Example: A Breather ........................... 117
9.6 Constructive Arithmetic: A Programming Language and its Logic .......... 118
  9.6.1 The Programming Language .................................................. 119
  9.6.2 The Logic of Arithmetic in Nuprl .......................................... 122
  9.6.3 The Proof Rules of HA ....................................................... 123
9.7 Classical Arithmetic: The Extension ............................................. 127
  9.7.1 New Logical Features .......................................................... 128
9.8 Classical Evaluation Semantics: New Constraints for Total Correctness 128
9.9 Call-by-Name (Almost) CPS-Translation and Kolmogorov Translation 130
  9.9.1 The CPS-Translation on Types: Preliminaries ............................ 131
  9.9.2 The CPS-Translation on Programs ......................................... 133
  9.9.3 The Compatible Kolmogorov/A-Translation on Proofs ................... 139
  9.9.4 Equivalence of The Translations: Conservative Extension Revisited ...................................................... 139
9.10 Semantic Equivalence: Value-Preserving CPS-Compilation ................. 140
  9.10.1 Motivation ........................................................................ 141
  9.10.2 The Colon Translation ....................................................... 141
  9.10.3 The Main Proof ................................................................ 149
9.11 Extensions: Primitive Recursive Function Symbols .......................... 154
9.12 A Recap of the Chapter ............................................................... 155
10 Double-Negation Translation: Extensions 156
  10.1 Other Translations and Evaluation Strategies .................. 156
    10.1.1 The Kuroda Translation: Call-by-Value .................... 158
    10.1.2 The Pervasive Kolmogorov Translation ...................... 164
  10.2 Evaluation, Typing, and Proof Translation .................... 165
  10.3 Negative Results ............................................. 167
  10.4 Extension to Higher-Order Theories ........................... 168
  10.5 Conclusions ................................................. 169

11 Conclusions and Directions for Future Research 170

Bibliography 174

A The Nuprl Rules 179

B Kinding Rules for Logic in Nuprl 191
List of Figures

1.1 A Simple Binary Tree Summer ........................................... 6
1.2 A Modified Binary Tree Summer: The Ugly Version .............. 7
1.3 Modified Binary Tree Summer: The Pretty Version .............. 8

3.1 Terms ............................................................................. 21
3.2 The II-Formation Rule ..................................................... 26
3.3 II-Type Member Introduction Rule .................................... 26
3.4 The Nonarithmetic Reduction Rules ................................. 29

6.1 The Gödel Translation ....................................................... 55
6.2 The Kolmogorov Translation ............................................. 55
6.3 The Friedman A-Translation ............................................. 58

9.1 The Reduction Rules For Arithmetic Programs .................... 121
Chapter 1

Introduction

In recent years there has been great interest in the applicability of constructive logics and type theories to the problems of program specification and development. Much of this interest has focused specifically around the problems of reasoning about programs at a high level, that is, without translating them into some machine-level description. Of equal (if not greater) interest is the goal of developing the proof of correctness (the explanation) for a program alongside the program itself. This desire to reason about programs at a high level has also been one of the motivating factors behind the development of functional programming languages. While functional languages began as relatively simple programming formalisms like the lambda-calculus [Bar84], they grew rapidly, with the addition of many new data-structuring ideas, until today we have functional languages with recursive data types, data abstraction [Mac85,CW85], polymorphism [Mil84], inheritance [Car84, JM88,Wan87], lazy evaluation and streams [HP90].

Though each of these extensions was translatable back into the lambda-calculus, and hence amenable to the same reasoning as pure lambda-calculus programs, the introduction of these programming extensions was followed by the introduction of new principles of proof which allowed programmers to argue directly about their programs, rather than first translating them into the simpler language.

The interest in constructive formal systems stemmed from the fact that these extended reasoning systems related programs in a straightforward way to a constructive interpretation of program specifications [ML82,Con86,Gir72,Mac86,Men87]. Quite naturally, much interest has been focused on constructive formal systems, both in terms of their connections to programming, and also as systems for mathematical reasoning.

For various reasons which will become apparent in this thesis, there are nonfunctional programming language features which are not amenable to direct reasoning in purely constructive formal systems. The goto-like C (pronounced "control") operator [FFED86], a relative of call-with-current-continuation from Scheme [RC86] is an example. Our work shows, however, that these constructs are amenable to direct reasoning in classical formal systems.
Our work began as an attempt to understand the connections between constructive and classical expressions of algorithms. Building on the work of Griffin [Gri90], Plotkin [Pla75], and others, we discovered that classical formal systems are systems for direct reasoning about nonfunctional programs. Our findings can be summarized as follows:

- We can give a consistent algorithmic interpretation to the rule of double-negation elimination, the classical axiom, in much the same manner as the other rules of logic are given algorithmic interpretations by constructive realizability, and this algorithmic interpretation is exactly $\mathcal{C}$.

- One can employ classical reasoning to provide total-correctness proofs for nonfunctional programs using $\mathcal{C}$.

At its core, our work deals with the problem of valid ways of expressing (and reasoning about) computation. It is well-known that all computation can be coded using Turing machines, and that number theory is sufficient for almost all reasoning about computation. Few, however, would jump to the conclusion that we need not concern ourselves with expressive reasoning systems and expressive programming systems, just as few would advocate programming in assembly language. Moreover, most would agree that a truly expressive programming system requires an expressive reasoning system. Thus, when we wish to add features to our programming language, and still reason about the correctness of programs, we must often add power to our logic. The question is how we can do this systematically. This thesis began as an attempt to understand the second task; to wit, what was to be gained (from the standpoint of programming) by adding classical axioms to a constructive logic used to derive programs. Partway thru our project, we discovered that adding such an axiom allowed us to incorporate facts about $\mathcal{C}$ in a very natural way. As a result, we found that from standard formal proofs in classical number theory we could extract computations in the same way that we extracted them from standard proofs in constructive number theory, and that under specific, well-defined circumstances, these computations were provably correct.

This chapter serves as a general introduction to the problems we wish to address. We will begin with a general discussion of the problem of reasoning about computation, and the enterprise of constructivism. From this, we will turn to descriptions of the two different problems which we solve, and then sketch out the remainder of the thesis.

1.1 Reasoning about Computation

If we want to derive programs systematically, we need a calculus of programs. One might think that this problem is easily solved by using a calculus for the underlying mathematics. After all, a computer can be mathematically modelled by a Turing
machine, which can be formalized in number theory, and we can then write things such as

$$\exists n, t. f(a) \succ_{t} n,$$

which says that the program $f$ on input $a$ terminates in $t$ steps, and reduces to $n$. If we now want to know what $n$ is, we know we can evaluate $f(a)$ to find out. But suppose we had instead proven:

$$\exists g. \forall a. (\exists n, t. g(a) \succ_{t} n) \Leftrightarrow \Phi(a, n),$$

which tells us that there exists a program which meets a particular specification. In this case, depending upon how we proved the sentence, we might have no idea how to find the $g$ in question. Whereas in the first case we had an algorithm provided for finding $n$ (start computing $f(a)$), in the second case we might have no earthly idea where to begin to look for $g$. This problem is not restricted to cases where the existence of a program is in question. Consider any undecidable proposition $\Phi$, and the sentence

$$\exists n. (n = 0 \Rightarrow \Phi) \land (n \neq 0 \Rightarrow \neg\Phi).$$

(1.1)

We think there exists a $n$, but we cannot say what value $n$ has. So does it make sense to say that $n$ exists? And even if we can say that $n$ exists, we certainly cannot say that $n$ can be computed. We can find other examples, for instance, the following classic example:

There are two irrational numbers, $a$ and $b$ such that $a^b$ is rational.

Here is a nonconstructive proof of it. Consider $\sqrt{2}^{\sqrt{2}}$; it is either rational or not. If it is, then take $a = \sqrt{2}$ and $b = \sqrt{2}$. Otherwise, take $a = \sqrt{2}^{\sqrt{2}}$

and $b = \sqrt{2}$, then $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$.

This proof, again, does not allow us to say what the $a, b$ are. But we can provide a proof which does:

We can exhibit two algorithms $A$ and $B$ for computing irrational numbers, $a$ and $b$ respectively, such that $a^b$ is rational. Here is a proof which suffices for both theorems. Take $a = \sqrt{2}$ and $b = 2 \cdot \log_2(3)$. Notice that there are algorithms $A$ and $B$ to compute these numbers.

The feature which separates the second proof from the first is that when we claim something exists, the proof provides an algorithm to compute it. Likewise, when we claim that a disjunction holds, the proof provides an algorithm to decide which branch of the disjunction holds. Certainly in standard mathematical reasoning these restrictions are not adhered to, and, as a result, when we are given a proof which purports to show that an object with certain properties exists, we cannot in general use the proof to derive a program which will construct this
object. When we reason about computation, we would like such an interpretation of existence, so that we can use mathematical reasoning to derive programs, and that is exactly what constructive reasoning gives us. This is not to say that classical mathematical systems are incapable of reasoning about computation; rather, that in constructive systems the notions of existence and computability are identified (when we say "there exists", we mean exactly "we can compute"), and that there are certain aesthetic and practical advantages to such an identification.

Having decided upon a computational view of existence, we can give consistent computational interpretations of all the logical connectives. To prove the sentence \( \exists x \in \mathbb{N}. P \) constructively, we must give a \( t \in \mathbb{N} \), and a proof of \( P \) when \( t \) is substituted for \( x \), that is, of \( P[t/x] \). Likewise, a proof of \( \forall x \in A. B(x) \) would be a means of computing, given \( t \in A \), a proof of \( B[t/x] \). To give one more example, a proof of \( P \lor Q \) would be a proof of either \( P \), or of \( Q \), along with a means of distinguishing which of the two was given.

Putting all this together, a proof of \( \forall x \in A. \exists y \in B. R(x, y) \) would be a means of computing, given \( X \in A \), a satisfactory \( Y \in B \), and a proof of \( R(X, Y) \). Of course, we know this as a function \( A \rightarrow B \) which meets certain correctness criteria, and carries with it evidence of its correctness. In presenting a constructive proof, though, one does not usually manipulate program expressions directly (though in Nuprl, for instance, one can). Rather, one simply proves a theorem constructively in much the same way as one does classically (but without recourse to the classical axiom). Thus, the proof of correctness of a program is developed right alongside the program itself; the program is automatically extracted from the proof of correctness. This approach has obvious merits for program synthesis, since it increases the integration of specification, programming, and verification.

Let us turn now from the reasoning system to the programs we extract from proofs.

### 1.2 Functional Programming and Constructive Reasoning

As we have described, a constructive proof always provides a way to construct an object which is proven to exist. This property of constructive proofs is referred to informally as the evidence property. Another nice property of most constructive reasoning systems is that the computations extracted from proofs are functional programs. For instance, given a constructive proof of \( \forall x. \exists y. R(x, y) \) in Heyting Arithmetic, we know this proof will be a functional program which, given a suitable \( X \), will compute the \( Y \), and a proof that \( Y \) is actually related to \( X \) by \( R(R(X, Y)) \). Consider now a proof of \( \text{Int} \Rightarrow \text{Int} \). Such a proof would be a functional program which, given an integer, would compute another integer. But that program would also have type \( \text{Int} \rightarrow \text{Int} \) in a suitable type theory for programming. Likewise, given
a functional program for the integer square root function, $\text{sqrt}$, we can prove

$$\forall n \in \mathbb{N}. \text{sqrt}(n)^2 \leq n \leq (\text{sqrt}(n) + 1)^2$$  \hspace{1cm} (1.2)

which exactly characterizes the integer square root function. If, on the other hand, we had been given a proof of

$$\forall n \in \mathbb{N}. \exists r \in \mathbb{N}. r^2 \leq n \leq (r + 1)^2,$$  \hspace{1cm} (1.3)

we could extract a function from this, call it $\text{root}$, and then $\lambda n. \text{fst}(\text{root}(n))$ would be a function $\mathbb{N} \rightarrow \mathbb{N}$ such proposition 1.2 held of it. Thus, in a sense, a constructive proof of proposition 1.3 above already contains the square-root function which meets the specification in 1.2. When we extend this idea a bit further, we are led to the Curry-Howard isomorphism, which can be stated thus:

Propositions as types,
Proofs as programs

or as

$p$ is a proof of propositions $\Phi$,
$p$ is an inhabitant of the type $\Phi$,
$p$ is a program which meets specification $\Phi$.

This idea stands at the foundation of many efforts to construct systems for reasoning about functional programs [Con86, CH84, HN88]. These efforts have met with varying degrees of success. However, the fact of the matter is that many natural programming methodologies simply do not fit well into the functional programming framework. Likewise, there are many proof methods (most notably classical reasoning) which do not fit well into the constructive framework. In the next section, we turn to these, and attempt to motivate extensions to cope with these other ways of reasoning and programming.

1.3 Nonfunctional Programming and Nonconstructive Reasoning

Nonfunctional Programming

One of the maxims of the functional programming community is that there are many problems whose solutions are best expressed as functional programs. We feel this is very true. However, as with any particular programming methodology, there are many programming problems whose solutions are best expressed in nonfunctional programming languages. We will give an example adapted from Felleisen [Fel87], but there are many such examples.
let sum T =
    if leaf(T) then val(T)
    else val(T) + sum(leftson(T)) + sum(rightson(T))

Figure 1.1: A Simple Binary Tree Summer

Suppose we are handed a binary tree, where each node, both internal nodes and leaves, contains an integer. We are asked to sum the values of this tree. This problem admits a beautiful functional solution, shown in Figure 1.1.

Suppose we wanted to sum the values of the nodes in a tree, but return zero if any particular node contained zero for its value. Then we would have to rewrite the program above, perhaps as in Figure 1.2.

Note how the type of the function sum, which was $\text{Tree} \rightarrow \text{Int}$, changes to $\text{Tree} \rightarrow (\text{Error} + \text{OK(Int)})$. Nevertheless, the top-level type is the same. This complicates the program greatly. In addition, it would be nice if, when we detected the first instance of a zero, we could abort the computation, and return zero at top-level, rather than compute out the sums of all the subtrees, only to discard them, as is done in Figure 1.2. So suppose we had a "catch/throw", as in LISP. Then we could write the program in Figure 1.3.

Note that this program does exactly what we specified - given a tree, it computes the sum of the values of the nodes, and if it finds a zero anywhere in the tree, it returns zero. But this program is not expressible in a pure functional language, since such a language would not have catch/throw. There are other examples of nonfunctional programming techniques that are quite useful, also. For instance, we could write a chess program as a routine which generated, one-by-one, all the different n-step move combinations, and then computed a quality function for each. This would be the classic depth-first search strategy. On the other hand, we could write it as a breadth-first search, where we explicitly kept track of the tree as we grew it. The former search strategy has advantages in code complexity, whereas the latter can sometimes lead to faster searches (though sometimes to slower searches). One can code breadth-first searches in terms of coroutines, and since we can implement coroutines in a functional programming language, theoretically we could write our breadth-first search routine in a functional language. But in reality, implementing coroutines in functional languages is a painful task, and actually requires a global re-organization of the program, much as in the first example above.

This reorganization is related to the denotational interpretation of nonfunctional programs as expressed in the work of Scott, Strachey, Milne, and others [MS76]. There, the idea of a continuation is presented as capturing the computations needed to finish the program. That is, the continuation is the "rest of the program", and nonlocal control operations are represented by operations which modify continuations, i.e. discarding, replacing, saving copies of, etc. This work was later extended
let topSum T =
let sum T =
    if leaf(T) then
        if val(T)=0 then Error
        else OK(val(T))
    else let
        leftval = sum(leftson(T))
        rightval = sum(rightson(T))
    in
        if Error = leftval or Error = rightval then
            Error
        else
            let OK(v1) = leftval
                OK(v2) = rightval in
            OK(v1+v2)
    end
in
let result = sum(T) in
    if Error = result then 0
    else let OK(v)=result in
        v
    end
end

Figure 1.2: A Modified Binary Tree Summer: The Ugly Version
let topSum T =
catch K in
  let sum T =
    if val(T) = 0 then throw K 0
    else if leaf(T) then val(T)
    else val(T) + sum(leftson(T)) + sum(rightson(T))
  in
  sum(T)
end
end

Figure 1.3: Modified Binary Tree Summer: The Pretty Version

by Fischer [Fis72], who showed that one could give a translation from a lambda-calculus back into itself such that the resulting program explicitly maintained a representation of the current continuation. Finally, Felleisen and others [FFED86] used these translations to convert programs with nonlocal control operations into pure functional programs. The translation is referred to as a continuation-passing-style (CPS) translation, and one distinguishing feature of it, other than the fact that it can translate a nonfunctional language into a functional one, is that it generates horribly opaque code.

We did not use a CPS translation to generate the program in Figure 1.2; instead we chose a much simpler translation. Nevertheless, even our simple translation greatly complicated the program. One would much prefer to write the compact program (Figure 1.3), rather than the messy program (Figure 1.2), even though the second is written in the (mathematically) more elegant language.

The phenomenon we are observing here was described by Felleisen in his thesis as a conflict between expressiveness (i.e. ease of coding such things as coroutines and escapes) and mathematical simplicity (the ability to reason about our programs using strictly local reasoning; the ability to use reduction-rule reasoning, as in the lambda-calculus). Now, we would like to be able to reason about our programs with these nonfunctional operations (they are in fact nonlocal control operations) without resorting to "coding them up" in a functional language first.

Our desire is simply expressed: we wish to integrate programs containing these operations into a constructive interpretation. We wish to reason in some mathematical system, and when we give a proof of a sentence \( \Phi \), we wish to extract the program which stands as witness for \( \Phi \). Looked at another way, we wish to write down our programs with nonlocal control operations, and reason about them directly in a manner much as we reason about normal functional programs.

Our work provides a first step in this direction. We demonstrate that there is indeed a simple extension of constructive reasoning systems which suffices to reason
about programs with these nonlocal control operations. In the remainder of this section, we will approach the problem of describing this extension.

Nonconstructive Reasoning

We can make an observation about mathematical reasoning which is, in a sense, much like the observation made about nonfunctional programming. That is, there are many times when we prove a theorem in mathematics which seems to have real constructive content; nevertheless, no matter how hard we look at that theorem, we cannot see the algorithm. For example, the statement of Higman's Lemma is

\[ \forall f \in \mathbb{N}^+ \rightarrow \{1, \ldots, m\}^*.\exists i < j \in \mathbb{N}^+. f(i) \ll f(j). \]

This sentence tells us that for every infinite sequence of strings over some finite subrange of the integers, there are \( i < j \) such that the \( i \)-th string is less than the \( j \)-th string in a particular ordering (the definition of \( \ll \) is found in Chapter 2). It turns out that the ordering is decidable (we can write down a program to check if the order holds between two strings). The standard proof of Higman's Lemma, as we shall see in Chapter 2, is a classical proof. If classical proofs had the evidence properties we talked about earlier, we would expect that we could extract a function (or a program) which, given a description of an infinite sequence of strings, would pick out the two elements of that sequence which were related by \( \ll \). But since it is commonly held that classical proofs do not contain algorithmic content, we would look elsewhere for our program.

In fact, there is a program for Higman's Lemma. This algorithm will enumerate all pairs \( i < j \) in \textit{backwards-diagonal} order, an enumeration of integer pairs which covers \( \mathbb{N}^+ \times \mathbb{N}^+ \). For each pair, it will check \( f(i) \ll f(j) \). Thus, if we know that the sentence is true, then we know that there indeed exists a pair \( i, j \) satisfying the condition above, and we can find it by search.

There are two things to notice here. First, we didn't need to know anything about the proof other than the fact that it exists to define this algorithm. Second, the algorithm we defined is much more than simply the classical proof. Someone who did not know about the backwards-diagonal injection of \( \mathbb{N}^+ \times \mathbb{N}^+ \) into \( \mathbb{N}^+ \) would be unable to find the algorithm we found.

If we were working constructively, we would expect that the proof would be all that is necessary, and that the proof would contain every detail of the algorithm. We would expect that different proof strategies would lead to different algorithms, and that some would be more efficient than others. The fact that our algorithm takes nothing at all from the proof, and that, upon reading the proof, we cannot discern any algorithm therein, makes us wonder if the classical proof has any relevance to understanding the computation.

The example presented previously, showing the existence of two irrational numbers \( a, b \) such that \( a^b \) is rational, only increases our discomfort with classical reasoning, as there, even though we know that one of the two pairs of real numbers
"works", we have no idea which pair it is, and we can see no way in the proof of figuring out which pair is correct. Thus one would be partially justified in thinking that the classical proof had no computation in it.

In both these cases, what is missing is a leap of intuition (or a deep theorem) which would take us from the classical proof of existence to a program that computes the object which is proven to exist. These leaps are clearly not automatable, and so we might have reason to think that classical reasoning is just not suitable for reasoning about computation (in the same manner that constructive reasoning is), since such a critical part of the process is essentially un-formalizable.

In 1952, though, Georg Kreisel [Kre58] cleared up the matter, by showing that a classical proof of a $\Pi^0_2$ sentence (like Higman’s Lemma) indeed gave a constructive proof, when viewed "with one’s head cocked to the side". Kreisel’s proof involved functional interpretation, and as such was horribly hard to follow. Later, in 1977, Harvey Friedman [Fri78] showed that Kreisel’s proof could be condensed down to a simple syntactic argument, his A-translation. Friedman’s contribution was in seeing that one did not need to reason completely in the metatheory of a constructive system to find the computation which stood as evidence for a classical proof. Rather, one could just translate the classical proof in a straightforward manner into a constructive proof of the same sentence.

So, one would think, our fears were quieted. Given the classical proof of Higman’s Lemma, Friedman’s A-translation technique would give us a constructive proof, and so we would have the required computation, computed from the proof. But several workers in the field felt that the A-translation did not give them an intelligible computation. They felt that, though one could mechanically construct a formally constructive proof, that proof was completely opaque, and thus it was hard to persuade oneself that in fact the constructive proof was valid. Despite the existence of the metatheorem, some workers found it very difficult to believe that the A-translation could be producing a computation from a proof which (it was believed) did not contain computational content.

The Connection

The readers might have noticed a common thread running through the two stories above, namely the idea that translations can render programs and proofs unintelligible. In the first case, it was CPS translation which replaced nonlocal control operations with functional program versions of them, and made the resulting program much harder to read, understand, and prove correct. In the second case, it was the Friedman A-translation, which replaced uses of the classical axiom with constructive reasoning, and rendered the proof much harder to read, understand, and extract programs from.

The results in this thesis attempt to address these objections to, on the one hand, the use of nonlocal control constructs, and, on the other, the use of the classical axiom. We show that the proper reasoning system for programs with nonlocal
control constructs is classical; symmetrically, we show that the computational content of the classical axiom is a nonlocal control construct. In the process, we will see that

- every classical proof can be regarded as a computation
- the tools with which to view classical proofs as computations have been lurking in the programming languages community for over a decade
- the meaning of a classical proof as a computation is clear and direct.

We can summarize the major theoretical result of this thesis with a single commuting diagram:

\[
\begin{array}{c}
\vdash_{PRL+EM} \forall \exists R \\
\xrightarrow{\text{Translation}} \\
\lambda x.M \\
\xrightarrow{\text{CPS-Translation}} \\
\lambda x.(M\tau) \\
\end{array}
\]

where \(\text{ext}_K\) is the classical program extraction procedure, \([\_]_K\) is the classical operational semantics (with \(C\)), the constructive versions (without \(C\)) are subscripted with a \(J\), and \(b\) is a primitive value.

Now, one might think that we are attempting the impossible, since it is well-known that classical logics do not in general have existence properties (from a classical proof of \(\exists x \in N.P\) we cannot always extract the relevant integer). But what we will learn is that from classical proofs we can always extract programs; however, only sometimes are these programs correct. The conditions under which they are correct are exactly those under which Friedman’s translation succeeds, and are sufficient to describe all functions provably total in the particular theory in which we are working. This might seem a bit paradoxical. What we mean is that from every classical proof we can extract a program (of some sort). However, only sometimes does that program perform the expected task.

### 1.3.1 The Research Contributions of This Work

This work unifies and explains the work of researchers in (semi-) functional programming languages (who search for reasoning systems for programs with nonlocal
control), and of researchers in mathematics, who search for the computational interpretation of classical proofs. This thesis shows conclusively that the rule of double-negation elimination is the proof-theoretic form of the (nonlocal control) operator $\mathcal{C}$. Alternatively, we show that $\mathcal{C}$ is the algorithmic content of the rule of double-negation elimination. We demonstrate that the translation method of Friedman is simply a continuation-passing-style (CPS) translation.

In the process, we demonstrate that classical proofs have very real constructive content, only in a form which is different from that in pure functional programs, and that CPS-translation simply renders that content into pure functional form.

We also demonstrate that the method of Friedman $\lambda$-translation is a feasible method of generating constructive proofs from classical proofs, by using it to prove Higman’s Lemma.

Our research contributions can be described succinctly as follows:

1. We extend and enhance the understanding of classical and constructive reasoning, demonstrating the intimate relationship between nonlocal control operations and classical reasoning, and making programs with $\mathcal{C}$ amenable to simple, direct total-correctness proofs.

2. In the process, we show that double-negation translation and CPS-translation are the same translation, looked at in different ways. Moreover, we show that employing different double-negation translations to convert a classical proof into a constructive one is equivalent to using different operational semantics for evaluating the classical proof/program directly.

3. In the process of proving Higman’s Lemma automatically, we had to prove that impredicative universal quantification was sound in Nuprl. We did so by extending work of Mendler [Men88] in Chapter 5. This work was only incidental to our main thrust, but in itself constitutes a step forward in the power of the Nuprl type theory, as it is now possible to encode even more logic than was possible before.

### 1.3.2 Comparisons with Other Work

The work in this thesis builds upon the work of others in constructing and extending the Nuprl system and type theory, most notably [Con86,How88]. In addition, the work of Friedman [Fri78] laid the theoretical foundations upon which our entire translation effort was built. The motivations which eventually led to the work in Chapters 9 and 10 issue from Leivant [Lei85], who showed that one could view Friedman’s $\lambda$-translation as a simple constructive-content-preserving operation on a proof in a minimal logic. However, both Leivant and Friedman were mathematicians, so neither performed automatic proof translation to put into practice the theoretical results, nor did they care to give a characterization of the effect of the translation upon algorithmic witnesses which would be intelligible to a computer
scientist (after all, they were mathematicians). Rather, they were mainly concerned with the mathematical form and implications of the translations.

In the computer science community, Fischer [Fis72] and Plotkin [Plo75] began the task of understanding continuation-passing-style translation. They showed that CPS-translation preserved the values of integer-valued lambda-calculus programs. In foundational work that truly set the stage for our work, Griffin [Gri90] observed that one could consistently give the nonlocal operator $\mathcal{C}$ the type $\neg\neg(P) \Rightarrow P$. He went on to show that for simply-typed lambda-calculus, the CPS-translation greatly resembled a double-negation translation from classical to constructive implicational logic, and that it preserved the values of programs of ground type. However, Griffin was unaware of Friedman’s work on conservative extension and A-translation, and thus did not extend his work to total-correctness type theories, or make the fundamental connection between CPS-compilation and double-negation/A-translation (and not simply double-negation translation).

Jean-Louis Krivine [Kri90] came as close as anyone (other than Griffin) to some of the results in this thesis. He discovered that a version of the Gödel double-negation translation was a CPS-translation. This result will fall out of the more general analysis we present in Chapters 9 and 10. Meyer and Wand’ [?] also discovered that CPS-translation was a double-negation translation.

1.4 An Overview

This thesis reflects two separate threads of work going on today in programming languages. One can read this thesis as programming languages research with implications for constructivism, or research in constructive mathematics with implications for programming. I hope that the readers will bear this in mind as they read this work.

Before proceeding with a chapter-by-chapter overview, we will outline in broad strokes the work in this thesis. We motivate the work by examining the standard classical proof of Higman’s Lemma [Hig52], a famous theorem of combinatorics, related to Kruskal’s Tree Theorem [Gal87] and the Graph Minor Theorem [RS85]. Higman’s Lemma (and its relatives) figure prominently in logic (due to the systems of mathematics which are necessary for their formalization [Sim85]). Kruskal’s Theorem and the Graph Minor Theorem are not formalizable in any predicative theory of arithmetic [FRS87,Gal87]. Hence, their proofs exploit impredicativity in what we will call an “essential” manner. Moreover, the proofs are obtained via many nested arguments by contradiction. As we described earlier, Higman’s Lemma asserts the existence of a pair of integers with certain checkable properties. As we discussed before, the classical proof does not seem to give us a way of constructing those integers which are proven to exist. Rather, we have to make the additional discovery of a backwards-diagonal method of enumerating all pairs of integers, and use this to build an unbounded search procedure.
Introduction

But Friedman's results show that one did not need to make this additional discovery; that the classical proof already contains a computation, and that we can (theoretically) translate the classical proof of Higman's Lemma into a constructive proof. From this constructive proof, we would then extract a program which was a direct construction, as opposed to an unbounded search. This is exactly what we did in our research. We formalized a classical proof of Higman's Lemma, and proceeded to implement Friedman's translation scheme, using it to translate that classical proof into a constructive proof. In the process, we incurred a huge blowup in the size of the proof, some of it incidental, some of it unavoidable. The final proof consumed over 50 megabytes of disk space. When we finished, we were able to execute the program on a trivial input, though due to space limitations (the program itself, with no proof information, took up 12 megabytes of space on disk) we could not run it on any nontrivial inputs.

Through this proof development effort, we learned a great deal about the implementation choices which one must make in constructing a theorem-prover to handle such large problems. We also learned about some very simple methods of parallelizing the Nuprl system, which we will describe in Chapter 8.

When we learned of work by Griffin on a classical typing of the $\land$ operator from (semi)functional programming languages, we realized that the Friedman double-negation/A-translation was exactly a continuation-passing-style (CPS) compilation of the classical proof into a constructive proof. We discovered that we could assign programs to classical proofs in a direct manner, by giving the rule of double-negation elimination (equivalent to excluded middle) an algorithmic meaning. Then, we discovered that the double-negation/A-translation converted this "classical program" into a constructive program which evaluated to exactly the same value as the classical program (for certain kinds of specifications). Thus, for certain (very general) classes of programs, classical reasoning is completely admissible, and we can evaluate the "algorithmic extractions" of these classical proofs just as if they were constructive extractions.

It turns out that functional programming languages with $\land$ are non-Church-Rosser. This means that different evaluation strategies can be terminating, and yet yield different values for the same program. We discovered that double-negation/A-translation fixes (imposes) an evaluation order on programs with $\land$, so that regardless of which evaluation order the translated programs are evaluated in, they give the same result.

Our work provides a foundational account of the algorithmic nature of classical reasoning, in the same way that constructive realizability provided such an account for constructive reasoning. For the first time, we can begin with a classical proof and directly extract and evaluate a program with certainty that the program will perform in accordance with its specification.

This thesis is organized as follows:

- Chapter 2 discusses an informal classical proof of Higman's Lemma, which serves as the background for much of the next few chapters.
1.4 An Overview

- Chapter 3 describes the Nuprl type theory and theorem-proving environment.
- Chapter 4 discusses the formalization of Higman’s Lemma in Nuprl.
- Chapter 5 justifies the use of impredicative quantification which was necessary to prove Higman’s Lemma in Nuprl.
- Chapters 6 and 7 discuss double-negation translation and Λ-translation both for constructive logics in general and for the Nuprl type theory in particular, showing the range of applicability of these translations, and the areas where characteristics of type theory make translation difficult or well-nigh impossible.
- Chapter 8 reflects upon the experience we gained in the implementation of an automated proof translation engine which we used to translate our classical proof.
- Chapters 9 and 10 discuss the connections between Friedman’s translation result and CPS-compilation.
- We end with some conclusions on the entire enterprise and many directions for future research.

Chapters 7 and 5 are mostly technical, and can be skipped on a first reading. Those wishing to understand the connections between classical reasoning and programming can restrict their attention to Chapters 3, 6, 9, and 10.
Chapter 2

Motivation: A Description of Higman’s Lemma

In this chapter, we describe the classical proof Higman’s Lemma, which is the motivation for our research.

2.1 Preliminaries

Definition 2.1.1. Given a set $\Sigma$, a binary relation $\leq$ is a well-quasi-order (WQO) if and only if $\leq$ is a preorder on $\Sigma$, and for any infinite sequence $s_1, s_2, s_3, \ldots$ of elements of $\Sigma$ there are $i < j$ such that $s_i \leq s_j$.

Definition 2.1.2. Given strings $u = u_1u_2\cdots u_m$ and $v = v_1v_2\cdots v_n$ in $\Sigma^*$, we say that $u$ can be embedded in $v$ (written $u \ll v$) if and only if there is an injective, order-preserving mapping $g$ from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$ such that for all $i$ in $\{1, \ldots, m\}$, $u_i \leq v_{g(i)}$. Simply put, $u$ can be embedded in $v$ if we can erase some of the $v_i$ and produce a string which is pointwise above $u$. We call $u$ a substring of $v$, and $v$ a superstring of $u$.

Note that the string embedding $\ll$ is a preorder on $\Sigma^*$. Higman’s lemma states the following:

If $\leq$ is a WQO on $\Sigma$, then $\ll$ is a WQO on $\Sigma^*$.

When specialized to $\Sigma \equiv \{1, \ldots, m\}$, a nonempty subrange of the integers, Higman’s Lemma becomes:

$$\forall f \in \mathbb{N}^+ \to \{1, \ldots, m\}^*. \exists i < j \in \mathbb{N}^+. f(i) \ll f(j).$$

2.2 A Classical Proof of the Lemma

In this section, we sketch a classical proof of Higman’s Lemma. The classical proof of Higman’s Lemma is interesting because it contains a minimal bad sequence (MBS)
argument. The proof proceeds by assuming that Higman's Lemma is false, and arguing to a contradiction, by defining, via transfinite induction, a MBS. This sequence turns out not to be minimal, and we get a contradiction. This MBS argument turns up in Kruskal’s Theorem, of which Higman’s Lemma is but a small part. The version of the proof we present is due to Gallier [Gal87], and originally presented in this form by Nash-Williams [NW63].

**Lemma 2.2.1.** For any sequence \((a_i)_{i \geq 1}\) over a WQO set \(\Sigma\), there is a subsequence \(a_{\sigma(1)}, a_{\sigma(2)}, a_{\sigma(3)}, \ldots\) such that for all \(i\), \(a_{\sigma(i)} \leq a_{\sigma(i+1)}\) (i.e. the sequence \((a_{\sigma(i)})_{i \geq 1}\) is monotonically increasing).

*Proof:* Say that \(n\) is terminal iff \(\forall j < n. a_n \not\in a_j\). If infinitely many \(n\) were terminal, let \(a_{\gamma(1)}, a_{\gamma(2)}, a_{\gamma(3)}, \ldots\) be the subsequence of elements of \(a\) which are terminal. By the WQO property, we know there are some \(i < j\) s.t. \(a_{\gamma(i)} \leq a_{\gamma(j)}\). But then \(a_{\gamma(i)}\) is not terminal. So there are only a finite number of terminal elements.

Let \(N\) be the largest terminal element. Then

\[
\sigma(1) \equiv N + 1
\]

\[
\sigma(n + 1) \equiv \text{the least } k \text{ s.t. } \sigma(n) < k \land a_{\sigma(n)} \leq a_k.
\]

(2.1) \hspace{1cm} (2.2)

We can now define the infinite subsequence of \((a_i)_{i \geq 1}\) trivially. Let \(\sigma(1) = N\), and \(\sigma(i + 1) = k\) where \(k\) is the least integer such that \(\sigma(i) < k \land a_{\sigma(i)} \leq a_k\). \(\sigma\) is well-defined since every \(k > N\) is nonterminal.

### 2.2.1 The Main Proof

We now prove Higman’s Lemma.

**Theorem 2.2.1 (Higman’s Lemma)**

Given a well-quasi-order \((\Sigma, \leq)\), the ordering \((\Sigma^*, \ll)\) is a well-quasi-order also.

*Proof:* Assume that \(\ll\) is not a WQO on \(\Sigma^*\). Then there is at least one sequence \(s = s_1, s_2, s_3, \ldots\) such that for all \(i, j\) with \(i < j\), \(s_i \not\in s_j\). We will call such a sequence bad, and we will call non-bad sequences good. We define, by impredicative quantification across all bad sequences, a minimal bad sequence, \(t = t_1, t_2, t_3, \ldots\) as follows:

- Let \(t_1\) be a string of minimal length which starts a bad sequence.

- Given that \(t_1, t_2, t_3, \ldots, t_i\) have been defined, let \(t_{i+1}\) be an \((i + 1)\)-st string of minimal length from all bad sequences which start with \(t_1, t_2, t_3, \ldots, t_i\).

It is easily seen that the new sequence \(t\) is bad, since any prefix of it is also a prefix of a bad sequence. Note that there can be *no* instances of \(\epsilon\) (the empty string) in \(t\) since \(\epsilon\) may be embedded into any string. It is also clear that \(t\) is a minimal
sequence, in the sense that for any other bad sequence, \( x \), there exists a \( k \in \mathbb{N} \) such that \( x_i = t_i \) for \( i \leq k \), and \( |t_{k+1}| \leq |x_{k+1}| \).

Since all strings in \( t \) are non-null, let \( t_i = a_is_i \) where \( a_i \in \Sigma \) is the leftmost symbol of \( t_i \). The elements \( a_i \) define an infinite sequence \( a = (a_i)_{i \geq 1} \) in \( \Sigma^* \) and the elements \( s_i \) define an infinite sequence \( (s_i)_{i \geq 1} \) in \( \Sigma^* \). By the lemma above, we know there is a monotonically increasing subsequence \( a' = (a_{\sigma(i)})_{i \geq 1} \). We claim that the corresponding sequence \( s' = (s_{\sigma(i)})_{i \geq 1} \) is good. If not, there are two cases:

1. \( \sigma(1) = 1 \): The infinite sequence \( s' \) is bad, with \( |s_1| < |t_1| \), which contradicts the minimality of \( t \).

2. \( \sigma(1) > 1 \): The infinite sequence \( s' = t_1, t_2, t_3, \ldots, t_{\sigma(1) - 1}, s_{\sigma(1)}, s_{\sigma(2)}, s_{\sigma(3)}, \ldots \) is also bad, because \( t_k = a_k s_k \) for all \( k \geq 1 \), and \( t_i \ll s_{\sigma(j)} \) implies that \( t_i \ll t_{\sigma(j)} \) by the definition of \( \ll \). But \( |s_{\sigma(1)}| < |t_{\sigma(1)}| \), and this contradicts the minimality of \( t \).

Since the sequence \( s' = (s_{\sigma(i)})_{i \geq 1} \) is good, there exist positive integers \( i < j \) such that \( \sigma(i) < \sigma(j) \) and \( s_{\sigma(i)} \ll s_{\sigma(j)} \). We know \( a' \) is monotonically increasing, so we have

\[
   t_{\sigma(i)} = a_{\sigma(i)} s_{\sigma(i)} \ll a_{\sigma(j)} s_{\sigma(j)} = t_{\sigma(j)}.
\]

Hence, we conclude \( t \) is good, which is a contradiction.

\[ \blacksquare \]

### 2.3 An Analysis of the Proof

The important things to notice about this proof are

1. The use of excluded middle.

2. The use of the axiom of choice.

3. The use of impredicative quantification across the set of bad sequences.

The classical reasoning is rampant throughout the proof; the initial step, after all, is a proof by contradiction, and there are lots elsewhere. To rid ourselves of the classical reasoning requires dramatic changes to the proof [Fri78]; this operation is the focus of this thesis.

Likewise, the axiom of choice is used in the construction of the minimal bad sequence, and we will see that these uses of the axiom must be replaced with more strict definitions for which the axiom of collection are sufficient. This modification is trivial; it amounts to replacing the minimization over lengths of strings with minimization over some well order which respects string embedding (that is, if \( s \ll t \) then \( s \sqsubseteq t \) in this ordering). For \( \Sigma \equiv \{1, \ldots, m\} \), we can treat each string as a base \( m + 1 \) representation of an integer in order to get the total order we need.
This turns out to be a crucial step in the formalization of the classical proof, which we will turn to in Chapter 4.

Finally, the impredicative quantification in this proof is not removable without fundamentally changing the nature of the proof. This quantification shows up in our proof that the minimal sequence \( t \) we defined by induction is also bad. Then we are able to conclude a contradiction by constructing a yet-smaller bad sequence from \( t \). We construct \( t \) by quantification over all bad sequences, but we then show that it is a bad sequence itself, thus violating predicative stratification. While we could remove the impredicativity [RM90], Friedman showed that there are other theorems, most notably Kruskal’s Theorem, from which the impredicativity cannot be removed [Sim85]. So we will not attempt to remove the impredicativity; instead we choose to work within Nuprl extended with impredicative definitions (which we justify in Chapter 5).

The reader will note that Higman’s Lemma asserts that a particular search procedure always terminates; this search procedure (on input sequence \( s \)) enumerates pairs of positive integers \((i, j)\) in backwards-diagonal order, and tests for the substring embedding property \( s_i \preceq s_j \). The classical proof shows that this search will always terminate, but it gives us no clue whatsoever as to when. As mathematicians, we might be satisfied that the search terminates. As computer scientists, though we might be satisfied with the algorithm of unbounded search, we would like to know why the search terminates, in terms of something like a bound function which decreases at every iteration of the search loop, rather than simply that it does terminate.

Our effort, then, is directed at generating, automatically, a constructive proof of Higman’s Lemma from which an algorithm can be extracted. This proof is obtained by translation from the classical proof; thus we must formalize the classical proof before we do anything. Due to the size of the translated proof, we cannot say anything as of this writing regarding the actual algorithm inherent in it. However, we expect that as time goes by, further research will clear up these, and other, questions.
Chapter 3

The Nuprl System

In this chapter, we discuss the Nuprl type theory and proof refinement system. We will use the information in this description again and again as we describe translations of rules of the type theory and the implementation of these translations as Nuprl proof procedures. The chapter is divided up into two sections, the first describing the Nuprl type theory, and the second the Nuprl system implementation. Readers interested in more detailed information should consult [Con86].

3.1 The Nuprl Type Theory

Nuprl is a predicative constructive Martin-Lof [ML82] type theory. It is based on the ideas of computation and type. Intuitively, we think of our domain of discourse as being some set of terms. These terms are then endowed with an evaluation (computation) relation, \( \succ (\text{e.g. } (\lambda x.b)(a) \succ b[a/x]) \). This defines a computation system. Next, we define a type system by defining when a term denotes a type, and when two terms are equal in a particular type. Thus we define a partial mapping from terms to types, where a type is defined as a partial equivalence relation on terms. We then say that a term \( t \) is a member of a type \( T \) exactly when \( \langle t, t \rangle \) is in the partial equivalence relation associated with \( T \). We require that these relations (the equivalence relation inherent in a type) respect the evaluation relation. That is, if \( a = b \in T, a \succ a', b \succ b', T \succ T' \), then \( a' = b' \in T' \). Finally, we define the judgment forms, that is, the proof system, which must respect the definition of types. We describe only a portion of the Nuprl type theory. The rest can be found in appendix A. Finally, we list the reduction rules for nonarithmetic terms at the end of this chapter.

3.1.1 Terms, Typehood, and Evaluation

The terms of Nuprl can be divided up into two categories, depending on whether they are canonical when closed. A term is canonical when it is not a redex (e.g. \( \lambda x.x \)
3.1 The Nuprl Type Theory

<table>
<thead>
<tr>
<th>$x$</th>
<th>$n$</th>
<th>axiom</th>
<th>void</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\setminus x.b$</td>
<td>$a \leq b$</td>
<td>int</td>
<td>$Uk$</td>
</tr>
<tr>
<td>$x \ x$</td>
<td>$x : A # B$</td>
<td>$A \rightarrow B$</td>
<td>$x : A \rightarrow B$</td>
</tr>
<tr>
<td>${ x : A \mid B }$</td>
<td>${ A \mid B }$</td>
<td>$a = b \text{ in } A$</td>
<td>$A \mid B$</td>
</tr>
</tbody>
</table>

canonical if closed

| $\text{noncanonical if closed}$ |

<table>
<thead>
<tr>
<th>$a - b$</th>
<th>$a \cdot b$</th>
<th>$a / b$</th>
<th>$a \mod b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$^\wedge$</td>
<td>$^\wedge$</td>
<td>$^\wedge$</td>
<td>$^\wedge$</td>
</tr>
<tr>
<td>$-a$</td>
<td>$t(a)$</td>
<td>$a + b$</td>
<td>$\text{ind}(a; x, y, s; b; u, v, t)$</td>
</tr>
<tr>
<td>$^\wedge$</td>
<td>$^\wedge$</td>
<td>$^\wedge$</td>
<td>$^\wedge$</td>
</tr>
</tbody>
</table>

$x, y, u, v$ range over variables.

$a, b, s, t, A, B$ range over terms.

$n$ ranges over integers.

$k$ ranges over positive integers.

$i$ ranges over atom constants.

Variables written below a term indicate where the variables become bound.

"\wedge" indicates principal arguments.

---

Figure 3.1: Terms
is canonical, \((\lambda \, x.x)(1)\) is noncanonical). We will discuss the terms listed in Figure 3.1, pointing out the intuitive idea behind each term.

The integer type \((\text{int})^1\) is a type which contains all closed terms which evaluate to integers. Two integer expressions are equal exactly when they evaluate to the same "numeral". The forms \(a \, \text{op} \, b\), where \(\text{op}\) is one of \(+\), \(-\), \(*\), \(/\), and \(\text{mod}\), along with \(-a\) are also integer expressions with the expected evaluation relation. The form \(\text{ind}(a; \, x, \, y, s; \, b; \, u, \, v, t)\) is, loosely speaking, a computation form on integers. The subterm \(a\) is expected to be an integer, and if \(a\) is zero, \(\text{ind}(a; \, x, \, y, s; \, b; \, u, \, v, t) > b\). If \(a > 0\),

\[
\text{ind}(a; \, x, \, y, s; \, b; \, u, \, v, t) > t[a, \text{ind}(a - 1; \, x, \, y, s; \, b; \, u, \, v, t)]/u, v, [u, v].
\]

Likewise, if \(a < 0\), \(\text{ind}(a; \, x, \, y, s; \, b; \, u, \, v, t) > s[a, \text{ind}(a + 1; \, x, \, y, s; \, b; \, u, \, v, t)]/x, y, [x, y].\) We can think of this form as a "for-loop" of sorts, but it is much more powerful, in that we can define inductive predicates with it. Finally, the form \(a < b\) is another type, which is inhabited by the term \(\text{axiom}\) exactly when \(a, b\) are integers, and \(a\) evaluates to a numeral less than \(b\).

We can prove equalities over integers (using the I-type, to be discussed soon), prove inequalities, prove propositions by induction, form inductively defined propositions and predicates, and, in general, do most of the things with Nuprl integers that we can do with a formalization of the integers in any other logic.

The \(\Pi\)-type is a dependent function type. An expression \(x : A \rightarrow B\) is a \(\Pi\)-type when \(A\) is a type, and, for every \(a \in A\), \(B[a/x]\) is a type. At the end of this section, on page 23, we will discuss the predicative nature of Nuprl, which constrains the form of a \(\Pi\)-type. The members of \(x : A \rightarrow B\) as the functions \(f\) which, given \(a \in A\), produce a value \(f(a) \in B[a/x]\). Two functions are equal when they produce equal values for equal arguments (equality of functions is extensional). Thus we see how a \(\Pi\)-type is defined in terms of its constituent types.

There is only one rule of evaluation associated with the \(\Pi\)-type. It is the rule of beta-reduction. If we have an expression \(t(a)\), where \(t \in (x : A \rightarrow B)\), and \(a \in A\). then \(t(a) \in B[a/x]\). Moreover, if \(t > \lambda \, x. \, b\), then \(t(a) > b[a/x]\).

The \(\Sigma\)-type is a dependent product type \(^2\). An expression \(x : A \# B\) is a \(\Sigma\)-type exactly when \(A\) is a type, and, for every \(a \in A\), \(B[a/x]\) is a type. The members of \(x : A \# B\) are the pairs \((a, b)\) such that \(a \in A\) and \(b \in B[a/x]\). Two pairs are equal exactly when their respective parts are equal in the appropriate constituent types. The evaluation rules for \(\Sigma\)-types are in Appendix A.

The \(I\)-type is so-called because it is a type which encodes identity. It is written \(I(a, b, T)\), though usually pretty-printed as \(a = b \in T\). An expression \(a = b \in T\) is an I-type exactly when \(T\) is a type, and \(a, b\) are members of \(T\). \(I(a, b, T)\) has as a member the expression \(\text{axiom}\) exactly when \(a, b\) are equal members of \(T\). \(\text{axiom}\)

\(^1\)actually, the term \(\text{int}\) denotes the type of integers; for brevity, though, we will gloss over this fine difference in this general discussion

\(^2\)We refer to the \(\Pi\)-type as a dependent function type since it contains functions as members; similarly, the \(\Sigma\)-type contains pairs, and is hence a product type. Others, most notably category theorists, prefer to call the \(\Pi\)-type a dependent product type, and the \(\Sigma\)-type a dependent sum type
3.1 The Nuprl Type Theory

is a "dummy", in the sense that what is important about an I-type is not what its inhabitants are, but rather whether or not it has inhabitants. We refer to axiom as the trivial inhabitant. There are no intrinsic rules of evaluation for the I-type. There are rules of transitivity and symmetry for the I-type, and these together with the rules of inhabitation and typehood make it possible for us to regard the I-type as a strongly typed equivalence relation.

The union type is a binary disjoint union (or sum) type. An expression $A \mid B$ is a union type exactly when $A$ and $B$ are types. The members of a union type are expressions $inl(u)$ and $inr(v)$, where $u \in A$, and $v \in B$. The idea here is that the "tag" ($inl$ or $inr$) tells whether the value ($u$ or $v$) is in the left or right injection of the union. Two expressions $x, y$ are equal members of a union type $A \mid B$ when they inject in the same direction (both $inl$ or both $inr$) and the injected values are equal members of the appropriate type. Again, we leave off the rules of computation here.

The void type is the type which contains no members. It is used to encode the notion of falsehood, and the rules (to be discussed later) tell us that when we have an inhabitant of the void type in a proof, we can prove that any other type is inhabited. As an example of how void is used to encode falsehood, consider the type $1 \leq 2$. This type is encoded as $2 \times 1 \Rightarrow \text{void}$, and a possible inhabitant of this type is $\lambda x.\text{axiom}$. This inhabitant tells us that, given any $x$, an inhabitant of the type $2 \times 1$, $\text{axiom}$ is an inhabitant of void. The term $\lambda x.3$ is also a valid inhabitant. The idea is that, given counterfactual assumptions, any term is a member of any type, and the type system collapses. These ideas become formalized in the semantic account of the system, in Chapter 5 and [All87].

Finally, we come to the universe type, and the predicative nature of Nuprl. The universe type $U_i$ is a type which contains other types. $U_1$ is the universe type which contains simple types; those which do not make reference to collections of types. For instance, $\forall x \in \text{Int.}\exists y \in \text{Int.} y = x^2$ is an example of a simple type. $P : U_1 \rightarrow (P \rightarrow P)$ is an example of a type which is not simple, since it is defined in terms of $U_1$, a collection of types. The rules for universe types tell us how we can construct types from collections of types. These rules tell us how to construct types from the collection $U_4$, for instance. Moreover, they tell us when such a constructed type does not reside in $U_4$, but in $U_5$. The universe types are cumulative, so $U_i \in U_{i+1}$. Of course, the $i$ in the universe type is a constant, not an integer variable, so when we actually write down a universe type, we write down a numeral, as above. In the above discussion, whenever we stipulated that an expression was a type, we actually meant that the expression was provably a member of an appropriate universe. Let's re-do the II-type, to demonstrate.

The II-type is a dependent function type. An expression $x : A \rightarrow B$ is a II-type in $U_i$ when $A$ is a type in $U_i$, and, for every $a \in A$, $B[a/x]$ is a type in $U_i$. The members of $x : A \rightarrow B$ are the functions $f$ which, when applied to $a \in A$, produce a value $f(a) \in B[a/x]$. Two functions are equal when they produce equal values for equal arguments. Thus we see how a II-type is defined in terms of its constituent types.
Consider now the $\Pi$-type $P : U_1 \rightarrow (P \rightarrow P)$. We can "sugar" this type into $\forall P \in U_1 . (P \Rightarrow P)$, by regarding the "dependent" $\Pi$-type as a universal quantification, and the independent $\Pi$-type as an implication. One might ask what universe this type is in. The rules of Nuprl tell us that this type is in $U_2$, and not in $U_1$. This occurs because if we try to prove $(P : U_1 \rightarrow (P \Rightarrow P)) \in U_1$, we end up with the goal $U_1 \in U_1$. This goal is not provable in Nuprl, and it turns out that adding it as a rule to the system causes the theory to become inconsistent via Girard's paradox [Gir72, How88, Coq86, MR86, Rei86]. This motivated the definition of Nuprl as a predicative type theory.

A predicative type theory is one in which the objects of discourse must be defined before they can be talked about. In the preceding example, we had to define $U_1$ before we could quantify over it. What we really wanted to do, when we wanted to prove $(P : U_1 \rightarrow (P \Rightarrow P)) \in U_1$, was to extend the definition of $U_1$ as we were in the process of quantifying over it. Of course, that would mean that we weren't done defining $U_1$ yet, and we would have an impredicative system. The problem is that we wish to define a type by reference to a collection of types, of which our type is itself a member. This self-reference is the essence of impredicativity, and we eliminate it in Nuprl by labeling universes with indices, and not allowing a closed term to denote a type in $U_i$ if it contains universe-terms at levels $i$ or greater. The Nuprl semantics makes this precise, and extends it to open terms.

Note that we haven't said that an impredicative system is inconsistent. A particular means of arriving at an impredicative system, the addition of the rule $U_1 \in U_1$, leads to inconsistency. As we shall see later, there are other ways of adding impredicativity that do in fact lead to consistent systems. These methods, while admitting the kind of impredicative definition displayed above, rule out some other, more pathological, forms of impredicative definition.

### 3.1.2 Evidence and Judgments

Nuprl is a system for the construction of terms which are members of types. That is, it is a system for, first, proving that a term $T$ is a type, and second, constructing a member $t$ of that type $T$. We refer to $t$ as an inhabitant of or evidence for $T$. That is, taking the Curry-Howard [How80] isomorphism to heart, we regard $T$ as a statement of logic, and $t$ as the evidence for its truth. We will define the set of rules whereby we systematically construct inhabitants of a type.

In Nuprl, a judgment is a statement of the form $H \gg T \text{ ext } t$, and specifies, loosely, that, for any inhabitants of the type binding list $H$, the type $T$ is inhabited by $t$, where we substitute in the appropriate inhabitants for the free variables of $t$ (which appear in $H$).

Informal Definition 3.1.1 (Nuprl Proof Trees) A Nuprl proof tree is a tree of sequents each of the form $H \gg T \text{ ext } t$, such that every node falls into one of the following categories:
Unproven Leaf: If a node does not contain a justification (rule of inference), then it is an unproven leaf node. A proof tree which contains unproven leaves is incomplete.

Proven Leaf: If a node contains a justification, and that justification does not require any subsidiary judgments (e.g. the rule \( A \gg A \), then the node is a leaf node.

Internal: If a node contains a justification, and that justification requires subsidiary judgments be proven, then the node is called an internal node, and the subsidiary judgments are referred to as subgoals.

The method of proof construction is to start with an unproven sequent, apply some inference rule to it, derive zero or more subsidiary sequents, to which we recursively apply inference rules. At some point, we will run out of unproven subgoals, and hence we will have a complete proof.

Informal Definition 3.1.2 (Judgment) A judgment is an expression of the form \( H \gg T \text{ ext } t \). The judgment says that the following holds:

- Let the hypothesis list be \( x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n \) where in each \( T_i \) the only free variables come from \( x_1, \ldots, x_{i-1} \).
- Let \( T \) take all its free variables from \( H \).
- Then under the assumption that we have terms \( t_1 \in T_1, t_2 \in T_2[t_1/x_1], \ldots, t_n \in T_n[t_1, \ldots, t_n/x_1, \ldots, x_n] \), it is true that \( t[t_1, \ldots, t_n/x_1, \ldots, x_n] \) is a member of the type \( T[t_1, \ldots, t_n/x_1, \ldots, x_n] \).

A judgment captures the notion of inhabitation of a term with free variables which have been given types, or, equivalently, of inhabitation of \( T \) being dependent upon being given inhabitants of other types. The rules of Nuprl are expressed in terms of judgments, which leads to a formulation of the rules where they are all of the form:

\[
H \gg T \text{ ext } t \\
\text{ by } r \\
H_1 \gg T_1 \text{ ext } t_1 \\
H_2 \gg T_2 \text{ ext } t_2 \\
\vdots \\
H_n \gg T_n \text{ ext } t_n
\]

which admits two equivalent interpretations:

- \( H \) is a list of hypotheses, under which \( T \) is to be proven true, and \( t \) is a constructive witness for the truth of \( T \). The rule \( r \) is a refinement rule which produces the subgoals \( T_1, \ldots, T_n \) such that the truth of \( T \) under assumptions \( H \) can be reduced to the truth of the subgoals under their respective hypotheses.
\[ H \gg x : A \rightarrow B \text{ in } U_i \text{ by intro } \begin{array}{c} \text{[new } y \text{]} \\
\gg A \text{ in } U_i \\
y : A \gg B[y/x] \text{ in } U_i \end{array} \]

Figure 3.2: The \( \Pi \)-Formation Rule

\[ H \gg x : A \rightarrow B \text{ ext } \lambda y. b \text{ by intro at } U_i \begin{array}{c} \text{[new } y \text{]} \\
y : A \gg B[y/x] \text{ ext } b \\
\gg A \text{ in } U_i \end{array} \]

Figure 3.3: \( \Pi \)-Type Member Introduction Rule

- \( H \) is a list of typings of free variables. We wish to prove that \( t \) is a member of type \( T \). This involves proving that \( T \) is a well-formed type, and that \( t \) is a member of \( T \), all under the assumptions that the free variables of \( t, T \) are given typings by \( H \). The refinement rule \( r \) decomposes the typing judgement \( H \gg T \text{ ext } t \) into the subsidiary judgements, in such a manner that we can infer the truth of the top judgement from the truth of the subsidiary ones.

We obtain a proof of consistency for Nuprl by showing (in the semantics [All87]) that the truth of the subsidiary judgments implies the truth of the top judgment. In appendix A, we list the rules of Nuprl. Here we will list and explain two rules for the \( \Pi \)-type, shown in Figures 3.2 and 3.3. In Figure 3.2, in order to conclude that \( x : A \rightarrow B \) is a member of \( U_i \) (that is, a type at level \( i \)), we must show that \( A \) is a member of \( U_i \), and that under the assumption of \( A \)'s inhabitation by \( y, B[y/x] \) is in \( U_i \). In Figure 3.3, in order to conclude that \( \lambda y. b \) is a member of \( x : A \rightarrow B \), we must show that \( A \) is a member of \( U_i \) for some \( i \), and that, under the assumption that \( y \) is a member of \( A \), that \( b \) is a member of \( B[y/x] \).

### 3.1.3 An Example Proof

Let us now look at an example proof, to give a flavor of the use of the Nuprl ruleset. Suppose \( B, C \) are types; that is, we have proofs of \( \Gamma \gg B \in U_j \) and \( \Gamma \gg C \in U_j \). We will prove a typed version of extensionality, where, for readability, we leave off well-formedness goals (goals of the form \( H \gg T \in U_i \)).

\[ \Gamma, x : B \rightarrow C \gg x = \lambda y. x(y) \in B \rightarrow C \text{ by extensionality} \]
\[ (\ast) : B \gg x(z) = (\lambda y. x(y))(z) \in C \text{ by computation} \]
\[ \gg x(z) = x(z) \in C \text{ by elimination} \]
\[ \gg x \in B \rightarrow C \text{ by hypothesis} \]
\[ \gg z \in B \text{ by hypothesis} \]
\[ \gg \lambda y. x(y) \in B \rightarrow C \text{ by introduction} \]
\[ y : B \gg x(y) \in C \text{ (like (\ast))} \]
(the hypothesis rule justifies the judgment $\Gamma, x : A, \Gamma' \gg x \in A$).

### 3.2 The Nuprl Development Environment

Writing proofs in Nuprl, as in almost any formal proof development system, is a painful and laborious task, because formal proofs must be complete; nothing can be left as an exercise for the reader. To make working in these systems practical, most of them offer mechanized assistance in the form of proof editors, data-abstraction facilities, notational abbreviation facilities, heuristic reasoners, decision procedures, etc. The Nuprl proof development system offers these tools also. Some of these tools are:

- a window-oriented display interface, including a proof editor which allows the user to edit and modify their proof as if it were a tree which they were traversing.

- *Tactics*, programs which operate on proof trees in order to make progress toward completion.

- A notational *abstraction* facility, which allows the user to define their own abstract notions of, e.g. groups, rings, subsets of the integers, etc.

- A notation *abbreviation* facility, which allows the user to define their own display forms for various terms in the Nuprl type theory, or for their notational abstractions. (e.g. $\forall x \in A. B(x)$ for $x : A \rightarrow B(x)$).

#### 3.2.1 Tactics

The tactic library [How88,Bas89] in Nuprl is immense, and contains tactics which implement such common proof strategies as integer induction, generalization upon the value of a subterm (i.e. substituting a variable in for a subterm of a given term so that, for example, we may induct on that subterm), first-order-matching-driven lemma instantiation and hypothesis instantiation, propositional reasoning, backchaining, controlled forward chaining, simple analogical reasoning, intelligent term rewriting, and a host of other strategies.

The normal course of theorem-proving involves tactic invocation and new tactic construction to automate that portion of the work of proving a new theorem that is dull and repetitive, but also different from that work that was done to prove other theorems. Tactics are invoked from the proof editor, and extend the proof tree. Tactics can also be written (and there exist some in the standard collection) which transform a given proof tree in some manner. The author's own work falls in this class, and we will discuss these *transformation tactics* in chapter 8.
3.2.2 Notation

The Nuprl system allows the user to define notational abstractions via the type theory itself. In the type theory, we can define a function iterator, for example, as follows: Given a type $A$, a non-negative integer $n$, a value $a \in A$, and a function $f : A \rightarrow A$, let $IT(A, f, a, n) \equiv ind(n; u, v.a; a; x, y.f(y))$. Then we can prove the following facts:

- $IT(A, f, a, n) \in A$
- if $n = 0$ then $IT(A, f, a, n) = a \in A$
- if $n > 0$ then $IT(A, f, a, n) = f(IT(A, f, a, n - 1)) \in A$.

That is, $\lambda a.IT(A, f, a, n)$ is the $n$-th iterate of $f$ on its input argument. We can make this definition in Nuprl much as we did above, and routinely check that all the conditions we specified above hold, yielding lemmas. Then, the proofs of these conditions are automatically used by the standard tactic collection to “lift” reasoning in proofs containing uses of this iterator to a level where the iterator is abstract. We use instances of these lemmas to unfold or refold uses of the iterator, to verify consistent typings for uses of the iterator, etc. In later chapters, we will see numerous instances of defined notations. In each case, we defined the notation via this abstract definition facility.

The notational abbreviation facility in Nuprl is the counterpart of notational abstraction, in that the normal method of using the two is to first define an abstract construct, and define a “display form” for it with the notational abbreviation facility. It associates user-specified “pretty” display forms with terms in a proof, and tries to maintain the pretty-printing in the face of substitutions.

3.2.3 Libraries

Nuprl provides the user with the ability to create libraries of theorems, abstract definitions, notational abbreviations for these, and tactics. A library is a mapping from names to objects, and the user has complete control of the library contents.

In chapter 8 we will discuss how we used the information in this library to help automate the proof translation process.

3.2.4 The MetaLanguage

The current implementation of Nuprl is in CommonLisp, but the language which tactics are written in is ML [Pau87]. ML has been extended with abstract data types for Nuprl terms and proofs, and functions to operate upon these and also upon the lemma library. We will later describe some details of how we were able to make much of the task of proof translation automatic simply by manipulating the library from a program in much the same way as users manipulate it from their screens.
3.3 The Nonarithmetic Reduction Rules

The operational semantics of the core Nuprl language can be described in terms of a set of reduction rules. These rules tell us, given an occurrence of a term in a context, what that term reduces to. We call these rules reduction rules because they are independent of the context in which they are used. Let $\rightarrow_1$ denote one-step reduction. We list the nonarithmetic reduction rules in Figure 3.4.

\begin{align*}
\text{ind}(k_n;x,y.d;b;x,y.u) & \rightarrow_1 d[k_n,\text{ind}(k_{n+1};x,y.d;b;x,y.u)/x,y] \quad \text{if } n < 0 \\
\text{ind}(0;x,y.d;b;x,y.u) & \rightarrow_1 b \\
\text{ind}(k_n;x,y.d;b;x,y.u) & \rightarrow_1 u[k_n,\text{ind}(k_{n-1};x,y.d;b;x,y.u)/x,y] \quad \text{if } n > 0 \\
\text{list}\_\text{ind}(\text{nil};b;x,y,z.u) & \rightarrow_1 b \\
\text{list}\_\text{ind}(h.t;b;x,y,z.u) & \rightarrow_1 u[h,t,\text{list}\_\text{ind}(t;b;x,y,z.u)/x,y,z] \\
\text{decide}(\text{inl}(t);x.l;y.r) & \rightarrow_1 l[t/x] \\
\text{decide}(\text{inr}(t);x.l;y.r) & \rightarrow_1 r[t/y] \\
\text{spread}(<t_1,t_2>;x,y.t) & \rightarrow_1 t[t_1,t_2/x,y] \\
(\lambda x.b)(t) & \rightarrow_1 b[t/x] \\
\text{less}(k_n;k_m;t_1;t_2) & \rightarrow_1 t_1 \quad \text{if } n < m \\
\text{less}(k_n;k_m;t_1;t_2) & \rightarrow_1 t_2 \quad \text{if } n \geq m \\
\text{int}\_\text{eq}(k_n;k_n;t_1;t_2) & \rightarrow_1 t_1 \quad \text{if } n = m \\
\text{int}\_\text{eq}(k_n;k_m;t_1;t_2) & \rightarrow_1 t_2 \quad \text{if } n \neq m \\
\text{atom}\_\text{eq}(a;b;t_1;t_2) & \rightarrow_1 t_1 \quad \text{if } a = b \\
\text{atom}\_\text{eq}(a;b;t_1;t_2) & \rightarrow_1 t_2 \quad \text{if } a \neq b
\end{align*}

Figure 3.4: The Nonarithmetic Reduction Rules
Chapter 4

The Formalization of Higman's Lemma in Nuprl

In this chapter we discuss the formalization of the previously presented classical argument. Throughout this chapter, we will assume, unless specified otherwise, that excluded middle holds. Since we already knew that we would be translating the argument back into a constructive theory, we had to be especially careful in our formalization to prevent the inadvertent introduction of a non-translatable construct. In fact, we did introduce such a non-translatable construct, and we did indeed find that it was not translatable by the hard road of attempting a translation, failing, and having to reconstruct major portions of the formalization in order to sidestep the problem. In doing so, we came up hard against the predicative nature of the theory, as expected. In order to overcome this barrier, it was necessary to semantically justify a parameterized impredicative \( \Pi \)-type. To the reader who wishes to avoid the details of formalizing arguments in Nuprl, this chapter is perhaps superfluous. This chapter serves as a retrospective on a complex argument formalized in the Nuprl type theory.

In the following, we discuss two formalizations of the classical Higman's Lemma argument. The first turned out to be wrong, and the reasons that it is wrong are interesting and illuminating. First, though, some preliminaries.

4.1 Preliminaries

We chose the subrange \( \{1, \ldots, m\} \) for our set \( \Sigma \), and identified \( \Sigma^* \) with \( \Sigma \text{ list} \). We identified the class of functions in our first informal proof with the type of functions \( \mathbb{N}^+ \rightarrow \{1, \ldots, m\}^* \).

**Definition 4.1.1.** Let the set \( \text{Fun}(m) \) be the function space \( \mathbb{N}^+ \rightarrow \{1, \ldots, m\}^* \).

In the classical proof, we have implicitly available a way of picking the "smaller" of two values from \( \{1, \ldots, m\}^* \). In the informal proof, we used the function \textit{length},
defined in the obvious way on lists. In the formalized proof, we needed to put a well
order on \( \{1, \ldots, m\}^* \), and we did this by injecting \( \{1, \ldots, m\}^* \) into \( \mathbb{N} \), by treating
every member of \( \{1, \ldots, m\}^* \) as a base \( m + 1 \) integer. We called the injection \( \mu \),
and we required that

- \( x = \text{nil} \Leftrightarrow \mu(x) = 0 \)
- \( x \ll y \Rightarrow \mu(x) < \mu(y) \)
- \( \mu \) is injective.

The upshot of these definitions is to carry over the induction principle from the
natural numbers to the strings \( \{1, \ldots, m\}^* \). The informal classical proof using the
axiom of choice, described earlier, used the length function on tuples, which meets
the first two conditions above, but not the third. It turns out that when a \( \mu \) which
meets the third condition is used, we need not use the axiom of choice. As we will
discuss in section 4.3, the axiom of choice is not usable in proofs which we intend
to translate, so we had to use the \( \mu \) described above.

We codified the appropriateness of the \( \mu \) we picked by showing

\[
[\exists x. \Phi(x)] \Rightarrow [\exists x. \Phi(x) \land (\forall y. (\mu(y) \leq \mu(x) \land \Phi(y)) \Rightarrow x = y)]
\]  

This theorem allows us to find, for any decidable \(^1\) property \( \Phi \) that holds for some
\( x \in \{1, \ldots, m\}^* \), a unique, \( \mu \)-least \( x \in \{1, \ldots, m\}^* \) such that \( \Phi(x) \) holds for that
\( x \). Since in a classical theory, \( P \lor \neg(P) \) holds of any property, this allowed us to
carry out the minimization for any property of \( \{1, \ldots, m\}^* \).

### 4.2 The First Classical Proof

**Definition 4.2.1.** A function \( f \in \text{Fun}(m) \) is **bad**, written \( \text{Bad}(f) \), iff
\( \exists i < j \in \mathbb{N}^+. f(i) \ll f(j) \).

**Definition 4.2.2.** For two functions \( f, g \in \text{Fun}(m) \), \( n \in \mathbb{N} \), \( f =^n g \) iff
\( \forall i \in \{1, \ldots, n\}. f(i) = g(i) \).

**Definition 4.2.3.** We say \( f \) is minimal over \( 1, \ldots, n \), and write it \( \text{MinBad}(f, n) \), iff

\[
(n = 0) \quad \text{or} \quad (\text{Bad}(f) \land \text{MinBad}(f, n - 1))
\]

\[
\land \forall g \in \text{Fun}(m). [(\text{Bad}(g) \land f =^n g) \Rightarrow \mu(f(n)) \leq \mu(g(n))]\]

\(^1\)In this thesis a decidable proposition \( P \) is one for which we can show constructively \( P \lor \neg(P) \). When we work in a classical theory, we can do this for every proposition, and we sometimes say that every proposition is decidable in a classical theory, which simply means that excluded middle holds of every proposition.
That is, if $\text{MinBad}(f, 2)$, then $f(1)$ is minimal over all $g(1)$ such that $\text{Bad}(g)$. And $f(2)$ is minimal over all $g(2)$ such that $\text{Bad}(g)$ and $g(1) = f(1)$. When $\text{Bad}(f), \text{Bad}(g)$, for any $n$, we can show that

$$f \equiv_n g \Leftrightarrow (\text{MinBad}(f, n) \Leftrightarrow \text{MinBad}(g, n)). \quad (4.2)$$

In particular:

$$\forall g \in \text{Fun}(m). \text{MinBad}(g, 0).$$

In the following construction, we always assume $\exists f \in \text{Fun}(m). \text{Bad}(f)$, as was done in the informal classical proof of Higman’s Lemma. Now, we can show by induction (and excluded middle) that, for each $i$,

$$(\exists f. \text{MinBad}(f, i)) \Rightarrow \exists g. \text{MinBad}(g, (i + 1)).$$

The induction is used to find a $g$ such that $g(i)$ is $\mu$–minimal, using excluded middle at each step to decide whether the current $g(i)$ is $\mu$–minimal, or there is a function which would provide a smaller value. Then, based on that disjunction, we either use the inductively computed value or the new value. We next show

$$\forall i. \exists f. \text{MinBad}(f, i). \quad (4.3)$$

Moreover, since

$$\text{MinBad}(f, n + 1) \Rightarrow \text{MinBad}(f, n),$$

any function $f$ such that $\text{MinBad}(f, n)$ is also $\text{MinBad}(f, k)$ for all $k \leq n$. We can use the axiom of choice to prove, from (4.3):

**Lemma 4.2.1 (Existence of a Minimal Bad Sequence)** Given that there is a bad sequence, there exists a minimal bad sequence $f$ such that for every $i$, $\text{MinBad}(f, i)$.

**Proof:** Let $f_1$ be such that $\text{MinBad}(f_1, 1)$. $f_1$ exists by (4.3). Likewise, define $f_2, f_3, f_4, \ldots$. For any $k$, $\text{MinBad}(f_k, k)$. So define the diagonal function $f_\omega \equiv \lambda i. f_i(i)$. $f_\omega$ is well-defined since each of the functions which are used to construct it are, and we use the axiom of choice to collect all the values together. It is easy to show

$$\forall n. f_n \equiv_n f_\omega, \quad (4.4)$$

and

$$\text{Bad}(f_\omega). \quad (4.5)$$

(4.5) follows from the fact that if $f_\omega$ were good, there would be $i < j$ witnessing the good-ness of $f_\omega$. But then $f_j$ would also be good, which is a contradiction. It follows easily that

$$\forall n. \text{MinBad}(f_\omega, n) \quad (4.6)$$
and we have proven that $f_\omega$ is our sought-after minimal bad sequence.

The only work left to do in this theorem is to prove that, for any function $f \in \text{Fun}(m)$, there exists a subsequence selection function $\sigma$ such that the function $\lambda n. \text{fst}(f(\sigma(n)))$ is constant-valued. This proof is routine, and much like the informal proof of Lemma 2.2.1 in Chapter 2. Given this proof, we can reconstruct the other steps in the classical proof and proceed to prove a contradiction from the assumption $\exists f \in \text{Fun}(m).\text{Bad}(f)$. This allows us to conclude that all $f \in \text{Fun}(m)$ are good, as desired. ■

4.3 What's Wrong with the Axiom of Choice?

The proof we presented is simple and succinct; unfortunately, it is not translatable. The difficulty comes up in trying to translate the axiom of choice. To understand what is wrong with the axiom of choice requires a passing knowledge of double-negation translation results. We will describe these in only the briefest detail, delaying a thorough exposition of the results until Chapter 6. The basic idea in double-negation translations is that if $\vdash_K \Phi$, that is, if we can prove $\Phi$ classically, then $\vdash_J \Phi^\circ$ (we can prove $\Phi^\circ$ intuitionistically), where $\Phi \mapsto \Phi^\circ$ is a translation defined by induction on the syntax of terms. We will work with only one double-negation translation here - the Gödel [God65] translation. It is defined as follows:

\[
\begin{align*}
(A \lor B)^\circ & \rightarrow \neg \neg(A^\circ \lor B^\circ) \\
(A \land B)^\circ & \rightarrow A^\circ \land B^\circ \\
(\exists x \in A.B)^\circ & \rightarrow \neg \neg(\exists x \in A.B^\circ) \\
(\forall x \in A.B)^\circ & \rightarrow \forall x \in A.B^\circ \\
(A \Rightarrow B)^\circ & \rightarrow A^\circ \Rightarrow B^\circ \\
P^\circ & \rightarrow \neg \neg(P) \quad (P \text{ prime}^2)
\end{align*}
\]

Given a classical theory $K$, and its intuitionistic counterpart $J$, the basic result that one establishes is as follows: let $\vdash_J \Phi$ mean that we have an intuitionistic proof of $\Phi$, and $\vdash_K \Phi$ mean that we have a classical proof. If $\vdash_K \Phi$ then $\vdash_J \Phi^\circ$. We can embed classical logic into intuitionistic logic with an appropriate translation. In later chapters we will use this embedding to establish a conservative extension result for proofs of $\Pi^0_3$ formulas.

It is not possible to double-negate translate the preceding proof. In particular, the axiom of choice (AC), as formalized below, is not translatable:

\[
(\forall x \in \text{Dom} \exists y \in \text{Rng}. \Phi(x,y)) \Rightarrow \exists f. \forall x \in \text{Dom}. \Phi(x, f(x)).
\]

The double-negation of this formula simply isn't provable constructively, and in fact is false in the standard Nuprl semantics.

Let us investigate further why the axiom of choice fails. What we would really like is the following: if AC holds in some classical theory K, then we would like $AC^\circ$
to hold in the intuitionistic variant, J. So let us explore the logical consequences of the AC°,
\[
(\forall x \in \text{Dom.} \neg \neg (\exists y \in \text{Rng.} \Phi(x, y))) \\
\Rightarrow \neg \neg (\exists f \in (\text{Dom} \rightarrow \text{Rng}.) \forall x \in \text{Dom.} \Phi(x, f(x)))
\]
where \(\Phi\) is a stable formula \(^3\). First, we can see that
\[
(\forall x \in \text{Dom.} \neg \neg (\exists y \in \text{Rng.} \Phi(x, y))) \Rightarrow \neg \neg (\forall x \in \text{Dom.} \exists y \in \text{Rng.} \Phi(x, y)). \quad (4.7)
\]
Reasoning in the semantics of Nuprl, this sentence says there is a function from inhabitants of \(\forall x. \neg \neg (\exists y. \Phi(x, y))\) to inhabitants of \(\neg \neg (\forall x. \exists y. \Phi(x, y))\), for any \(\Phi\) which is stable. But this isn’t true. Consider \(\text{Dom} \equiv \text{Int}, \text{Rng} \equiv \text{Bool}\) and
\[
\Phi(x, y) \equiv \neg \neg ((y = \text{true} \in \text{Bool}) \Rightarrow \neg \neg (\phi_x(x) \downarrow))
\]
where \(\phi_x(x) \downarrow\) is true exactly when the \(x\)-th algorithm terminates on input \(x\). Looking at the antecedent of the implication, obviously \(\forall x. \neg \neg (\exists y. \Phi(x, y))\) is inhabited (though we may not be able to prove it in the proof system) since the function \(\lambda x, y.\text{axiom}\) inhabits this type. Let us show this, in the informal semantics for Nuprl discussed previously. Given an arbitrary \(x\), we must show that \(\lambda y.\text{axiom}\) inhabits \(\neg \neg (\exists y. \Phi(x, y))\). Since our metatheory is classical, we can say that either \(\phi_x(x) \downarrow\) is inhabited, or it isn’t. If it is, then \(\Phi(x, \text{true})\) is inhabited, and if not, then \(\Phi(x, \text{false})\) is. And so the type \(\exists y. \Phi(x, y)\) is inhabited; thus, \(\neg \neg (\exists y. \Phi(x, y))\) is not inhabited, and so \(\neg \neg (\exists y. \Phi(x, y))\) is inhabited by \(\lambda y.\text{axiom}\).

Let us look now at the consequent of the implication (in formula 4.7 above). The double-negated existential is inhabited exactly when the type \(\forall x. \exists y. \Phi(x, y)\) is inhabited. But clearly this couldn’t be inhabited, for then the term which inhabited it would be a program for deciding the halting problem, for which there is no program.

Thus, we can see that AC° is inconsistent with Nuprl, since it renders our semantics unsound.

### 4.4 The Final Classical Proof

The upshot of all this is that we cannot represent the space of sequences over \(\{1, \ldots, m\}^*\) by the type \(N^+ \rightarrow \{1, \ldots, m\}^*\). We have to find a different representation. In standard set theory, a function is simply a binary relation which is total and functional. We chose to use this representation, and represent these functions by the type \(\text{Rel}(m) \equiv N^+ \rightarrow (\{1, \ldots, m\}^* \rightarrow \text{U7})\) (the choice of U7 was arbitrary). We can think of inhabitants of this type as being relations over \(N^+\) and \(\{1, \ldots, m\}^*\). But functions from \(N^+\) to \(\{1, \ldots, m\}^*\) aren’t the same as relations - there are additional

---

\(^3\)A stable formula \(\Phi\) is one for which we can show \(\vdash J \neg \neg (\Phi) \Rightarrow \Phi\)
facts we know about them, namely they are relations that are total and functional. So we must "carry along" these requirements, by conjointing them to sentences existentially quantified over the space Rel(m), and adding them as extra assumptions in sentences universally quantified over Rel(m). That is, wherever (in the preceding proof) we quantified over \( \mathbb{N}^+ \rightarrow \{1, \ldots, m\}^* \), i.e. \( \forall f \in \mathbb{N}^+ \rightarrow \{1, \ldots, m\}^* \). \( P \), we must instead write

\[
\forall f \in \text{Rel}(m). \text{total}(f, \mathbb{N}^+, \{1, \ldots, m\}^*) \land \text{functional}(f, \mathbb{N}^+, \{1, \ldots, m\}^*) \Rightarrow P,
\]

where total and functional are defined below:

**Definition 4.4.1.** for \( A, B \) types, and \( f \in A \rightarrow (B \rightarrow \text{U7}) \),

\[
\text{total}(f, A, B) \equiv \forall x \in A. \exists y \in B. f(x,y),
\]

\[
\text{functional}(f, A, B) \equiv \forall x \in A. \forall y, z \in B. (f(x,y) \land f(x,z)) \Rightarrow (y = z \in B).
\]

We reconstructed the argument from section 4.2, though with some changes. We outline the final argument, as implemented on the machine, below:

**Definition 4.4.2 (Universal Quantification Over Functional Relations)**
Define \( \forall_{\text{func}} f. \Phi(f) \) to be an abbreviation for

\[
\forall f \in \text{Rel}(m). \text{total}(f, \mathbb{N}^+, \{1, \ldots, m\}^*) \land \text{functional}(f, \mathbb{N}^+, \{1, \ldots, m\}^*) \Rightarrow \Phi(f).
\]

**Definition 4.4.3 (Existential Quantification Over Functional Relations)**
Define \( \exists_{\text{func}} f. \Phi(f) \) to be an abbreviation for \( \neg(\forall_{\text{func}} f. \neg(\Phi(f))) \). We can show that, classically, this is equivalent to the Nuprl existential (the \( \Sigma \)-type),

\[
\exists f \in \text{Rel}(m). \text{functional}(f, \mathbb{N}^+, \{1, \ldots, m\}^*) \land \text{total}(f, \mathbb{N}^+, \{1, \ldots, m\}^*) \land \Phi(f),
\]

but we use the universal quantifier here because it allows us later to justify an impredicative definition that would not have been justifiable otherwise.

**Definition 4.4.4 (Well-Quasi-Order)** Given a relation \( f \in \text{Rel}(m) \), \( \text{WQO}(f) \) iff

\[
\exists i < j \in \mathbb{N}^+. \exists \mathbf{fi}, \mathbf{fj} \in \{1, \ldots, m\}^*. f(i, \mathbf{fi}) \land f(j, \mathbf{fj}) \land (\mathbf{fi} \ll \mathbf{fj}).
\]

**Definition 4.4.5 (Bad Sequence)**
Given a relation \( f \in \text{Rel}(m) \), \( \text{Bad}(f) \) iff \( \neg(\text{WQO}(f)) \).

**Definition 4.4.6 (f Matches g over 1..n)** Given \( f, g \in \text{Rel}(m) \), \( n \in \mathbb{N} \), we write \( f^\equiv g \) exactly when

\[
\forall i \in \{1, \ldots, n\}. \exists x \in \{1, \ldots, m\}^*. f(i, x) \land g(i, x).
\]

Right away we can see that \( f^\equiv g \) is transitive, reflexive, and symmetric in its first two arguments.
Definition 4.4.7 (x is minimal over f at i) Given \( f \in \text{Rel}(m), i \in \mathbb{N}^+ \), and \( x \in \{1, \ldots, m\}^* \), we write \( \text{MinOf}(x, f, i) \) exactly when

\[
\forall_{\text{func} g.} (f_i = g \land \text{Bad}(g)) \Rightarrow \exists \text{gn}. \ (g(n, \text{gn}) \land \mu(x) \leq \mu(\text{gn}) \land \exists_{\text{func} g.} f_i = g \land \text{Bad}(g) \land g(i, x)).
\]

Intuitively, \( x \) is the minimal value of \( g(i) \) for any \( g \) which match \( f \) on the first \( i - 1 \) values, and there is some bad \( g \) which attains the value of \( x \) at \( i \).

We can prove several useful properties about \( \text{MinOf}(x, f, i) \). For instance, if \( f, g \in \text{Rel}(m), n \in \mathbb{N}^+, x \in \{1, \ldots, m\}^* \), and \( f, g \in \text{Rel}(m) \) are functional relations, then \( (f^n = g \land \text{MinOf}(x, f, n)) \Rightarrow \text{MinOf}(x, g, n) \). That is, if \( f \) and \( g \) match on their first \( n - 1 \) values, and if \( x \) is the minimal \( h(n) \) over all functions \( h \) which match \( f \) on their first \( n - 1 \) values, then \( x \) must be the minimal \( h(n) \) over all functions \( h \) which match \( g \) on their first \( n - 1 \) values. We can also show that if \( \text{MinOf}(x, f, n) \) and \( \text{MinOf}(y, g, n) \), then \( x = y \).

At this point we encountered the first major inductive proof. We proved

\[
\forall_{\text{func} f \in \text{Rel}(m)}. \text{Bad}(f) \Rightarrow \forall n \in \mathbb{N}^+. \exists_{\text{func} g \in \text{Rel}(m)}. \text{Bad}(g) \land f^n = g \land \exists \text{gn} \in \{1, \ldots, m\}^*. (g(n, \text{gn}) \land \text{MinOf}(\text{gn}, f, n))
\]

That is, for every \( n \), for every bad \( f \), we can find a function that equals \( f \) for the first \( n - 1 \) elements, and on the \( n \)-th value, is \( \mu \)-minimal over all other functions that equal \( f \) on the first \( n - 1 \) elements. The proof is via induction, carried over to \( \{1, \ldots, m\}^* \) by the property of \( \mu \) (formula 4.1) which we proved earlier.

We used the theorem about \( \mu \) mentioned earlier (that for any property of \( \{1, \ldots, m\}^* \) which holds for some \( x \in \{1, \ldots, m\}^* \), there exists a unique least \( x \in \{1, \ldots, m\}^* \) such that the property holds of that \( x \) to find the \( g \) in question via induction. The proof method is to assert that either the provided bad sequence, which is minimal out to \( n - 1 \), is minimal at \( n \), or it is not. If it is minimal, then we are done. If it is not minimal, then there is some sequence which is minimal out to \( n - 1 \), and moreover is below our provided sequence at \( n \). So then we inductively assume that we can find a minimal bad sequence, starting with this new function, and we are done.

So, given a bad sequence, we can find a "minimal bad extension" of that bad sequence. In fact, since \( \mu \) is injective, we can find a "minimum bad extension". We need to define a minimum bad extension because otherwise we would need to use the axiom of choice to define our minimal bad sequence. But here, we need only define a minimum bad sequence, which is the next step.

As in the classical proof, we define it by first defining its approximations, e.g. of length 1, 2, etc, and then diagonalizing.
Definition 4.4.8. Let \( \text{diag}_0(i, x) \equiv \text{False} \), and, for \( n > 0 \), \( x \in \{1, \ldots, m\}^* \), let

\[
\text{diag}_n(i, x) \equiv i = n \land \exists_{\text{func} \, f} \exists_{g} g \in \text{Rel}(m) . \\
\text{Bad}(g) \land g(i, x) \land \text{MinOf}(x, g, i) \\
\land \forall j \in \{1, \ldots, i - 1\}, \exists y \in \{1, \ldots, m\}^* . g(j, y) \land \text{diag}_{n-1}(j, y) \\
\lor ((i \neq n) \land (\text{diag}_{n-1}(i, x)))
\]

We can prove that if \( i < n \), then \( \text{diag}_n(i, x) \iff \text{diag}_{n-1}(i, x) \). We can think of \( \text{diag} \) as a chain of approximations, each of which is more precise than the previous.

Each approximation \( \text{diag}_n \) is indeed total and functional over its domain:

\[
(\exists_{\text{func} \, f} . \text{Bad}(f)) \\
\Rightarrow \forall n \in \mathbb{N}^+. \text{total}(\lambda i, x. \text{diag}_n, \{1, \ldots, n\}, \{1, \ldots, m\}^*) \\
\land \text{functional}(\lambda i, x. \text{diag}_n, \{1, \ldots, n\}, \{1, \ldots, m\}^*)
\]

and since each approximation \( \text{diag}_n \) is an extension of \( \text{diag}_{n-1} \), we define \( \text{diag}_\omega \equiv \lambda i, x. \text{diag}_i(i, x) \), and show

\[
\exists_{\text{func} \, f} . \text{Bad}(F) \Rightarrow \text{functional}(\text{diag}_\omega, \mathbb{N}^+, \{1, \ldots, m\}^*) \\
\land \text{total}(\text{diag}_\omega, \mathbb{N}^+, \{1, \ldots, m\}^*) \\
\land \text{Bad}(\text{diag}_\omega) \\
\land \forall n. \exists_{\text{dn}} . \text{diag}_\omega(n, \text{dn}) \land \text{MinOf}(\text{dn}, \text{diag}_\omega, n)
\]

We show that (4.8) leads to a contradiction. The method, once again, is to pass to a subsequence such that each string in the subsequence has the same first element, and then strip off that first element. This step is much like Lemma 2.2.1, but modified pervasively to work on functional relations instead of functions. We obtain a subsequence selection function \( \sigma \in \mathbb{N}^+ \rightarrow (\mathbb{N}^+ \rightarrow \text{U7}) \) such that the composition \( \text{hd} \circ \text{diag}_\omega \circ \sigma \) is a constant function. That is, we want \( \sigma \) to pick out elements of \( \text{diag}_\omega \) all of which have the same first element.

We have not yet discussed the issue of impredicativity. The sequence \( \text{diag}_\omega \) is defined by induction on \( \text{diag}_n \), and each \( \text{diag}_n \) is defined by, in the end, universal quantification over \( \text{Rel}(m) \). Thus, in a predicative type system, \( \text{diag}_n \) sits in \( \mathbb{N}^+ \rightarrow (\{1, \ldots, m\} \text{ list} \rightarrow \text{U7}) \), and not \( \mathbb{N}^+ \rightarrow (\{1, \ldots, m\} \text{ list} \rightarrow \text{U7}) \). Thus, in a purely predicative type system, we cannot complete the argument, because \( \text{diag}_\omega \) must be a member of the class \( \mathbb{N}^+ \rightarrow (\{1, \ldots, m\} \text{ list} \rightarrow \text{U7}) \), so that we may derive from it a sequence which is below it in the lexicographic ordering on sequences. Thus, we will simply assume for now that \( \text{diag}_\omega \) is in \( \mathbb{N}^+ \rightarrow (\{1, \ldots, m\} \text{ list} \rightarrow \text{U7}) \). In Chapter 5 we will semantically justify this decision, by showing that the quantification in Definition 4.4.2 can be justified via a candidats de reductibilite argument.

Once we have an infinite subsequence selection function \( \sigma \), the rest of the informal classical proof can be formalized mutatis mutandis. All of these steps are constructive. We achieve a contradiction, as desired. The one nontrivial proof that we have skipped is the proof that there exists an appropriate \( \sigma \). The requirement
on \( \sigma \) is that the composition \( \lambda n.\text{hd}(\text{diag}_\omega(\sigma(n))) \) is the constant function. Since \( \text{hd} \) is clearly functional and total (on non-nil arguments), and so is \( \text{diag}_\omega \), and we can show that a bad sequence has no nil values we can rephrase the problem as follows: For a functional relation \( f \in \mathbb{N}^+ \rightarrow (\{1, \ldots, m\} \rightarrow \mathcal{U}) \), exhibit a subsequence selection function \( \sigma \) such that when composed with \( f \), the ensemble is a constant function.

Recall from the informal proof that there were two distinct steps. First, we showed that, for every function \( \mathbb{N}^+ \rightarrow \{1, \ldots, m\} \) there must exists \( k \in \{1, \ldots, m\} \) such that

\[
\forall i \in \mathbb{N}^+. \exists j > i \in \mathbb{N}^+. f(j) = k.
\]  

(4.9)

That is, there must exist a constant-valued subsequence. The classical proof for functions extends to a proof for functional relations. Second, we showed that, given \( m \) and a witness for (4.9) above, there in fact existed a function \( \sigma \) such that \( f \circ \sigma \) was constant-valued (with value \( k \)). \( \sigma \) iterates the witness to produce ascending values \( j \) for which \( f(j) = k \).

There is only one snag in this proof method. (4.9) does not tell us that the \( j \) which is asserted to exist is unique. Thus we can use (4.9) to define relation, but not a function. We can use a version of the pigeonhole principle to convert the existence of a \( j \) to the existence of a unique (least) \( j \), and then proceed to define a function \( \sigma \). All that remains is to exhibit a bad sequence which is below \( \text{diag}_\omega \), and we have the contradiction we desire. These steps are just as in the informal proof, and are omitted. That completes the discussion of machine formalization.

### 4.5 A Retrospective

It should be clear at this point that certain methods of formalizing proofs for later Gödel-translation will work, and others will not. The central failure of the double-negation translation is the axiom of choice. The double-negation translation of the axiom of choice is false in Nuprl. We avoid this difficulty by using functional relations, so we can define \( \text{diag}_\omega \) as simply being equal at each argument \( i \) to \( \text{diag}_i \). Then instead of using the axiom of choice, we simply demonstrate that our relations are total and functional.

Another difficult point (which we will come to again) is the issue of impredicative definition. As we discussed, the diagonal sequence \( \text{diag}_\omega \) does not sit (in a predicative type system) in the same universe level as the bad sequences from which it is defined. Thus, we cannot use it to conclude a contradiction due to the fact that \( \text{diag}_\omega \) is not a bad sequence at the same universe level as the bad sequences from which it was defined. This is because the "smaller" bad sequence we build from \( \text{diag}_\omega \) is not in the type of bad sequences either, and hence there is no contradiction. To remedy this problem, we simply assumed that \( \text{diag}_\omega \) was a member of the proper type of bad sequences, and in Chapter 5 we will fully semantically justify this decision.
Chapter 5

A Proof of Consistency of Impredicative \( \Pi \)-Types in Nuprl

In this chapter, we will prove that impredicative \( \Pi \)-types in Nuprl are consistent. We will do this by first defining a semantics for Nuprl and then extending that semantics to deal with the impredicative type constructor. This work is taken almost wholly from the work of Mendler [Men87].

5.1 Motivation

The motivation for the work in this chapter is a justification of impredicative quantification. Recall from Chapter 4 that at one point we make the definition

**Definition 4.4.8.** Let \( \text{diag}_0(i, x) \equiv False \), and, for \( n > 0, x \in \{1, \ldots, m\}^* \), let

\[
\text{diag}_n(i, x) \equiv \\
\begin{align*}
i &= n \land \exists_{\text{func}} g \in \text{Rel}(m). \\
& \quad \text{Bad}(g) \land g(i, x) \land \text{MinOf}(x, g, i) \\
& \quad \land \forall j \in \{1, \ldots, i - 1\}. \exists y \in \{1, \ldots, m\}^*. g(j, y) \land \text{diag}_{n-1}(j, y) \\
& \land \left( (i \neq n) \land (\text{diag}_{n-1}(i, x)) \right)
\end{align*}
\]

In this definition, the notation \( \exists_{\text{func}} g. B \) is defined as

**Definition 4.4.3 (Existential Quantification Over Functional Relations)** Define \( \exists_{\text{func}} f. \Phi(f) \) to be an abbreviation for \( \neg(\forall_{\text{func}} f. \neg(\Phi(f))) \).

and we defined \( \forall_{\text{func}} f. \Phi(f) \) as

**Definition 4.4.2 (Universal Quantification Over Functional Relations)** Define \( \forall_{\text{func}} f. \Phi(f) \) to be an abbreviation for

\[
\forall f \in \text{Rel}(m). \text{total}(f, N^+, \{1, \ldots, m\}^*) \land \text{functional}(f, N^+, \{1, \ldots, m\}^*) \Rightarrow \Phi(f).
\]
Thus $\text{diag}_n(i, x)$ is an abstract notation in which we universally quantify over the type $\text{Rel}(m)$, (which is $N^+ \rightarrow (\{1, \ldots, m\}^* \rightarrow U7)$). Of course, by the rules of construction of $\Pi$-types, this means that the type $\forall_{\text{func}} B$ is not in $U_7$, but rather in a higher universe. It follows that $\text{diag}_n(i, x)$ is not in $U_7$ either, and that means that $\text{diag}_\omega \equiv \lambda i, x. \text{diag}(i, x)$ isn’t in $N^+ \rightarrow (\{1, \ldots, m\}^* \rightarrow U7)$, either.

Recall that in the formalization, the next step was to construct a function that was lexicographically below $\text{diag}_\omega$, and then use the fact that $\text{diag}_\omega$ was a minimum bad sequence to reach a contradiction. Since $\text{diag}_\omega$ is not in $N^+ \rightarrow (\{1, \ldots, m\}^* \rightarrow U7)$, neither is the function we construct that is below $\text{diag}_\omega$. Since $\text{diag}_\omega$ is the minimum of all functions in $N^+ \rightarrow (\{1, \ldots, m\}^* \rightarrow U7)$, we cannot obtain a contradiction from the fact that there is a function in a higher universe which is below $\text{diag}_\omega$.

If we could prove, though, that $\text{diag}_n(i, x) \in U_7$, then the proof would work, since we would be able to show $\text{diag}_\omega \in \text{Rel}(m)$. We introduce impredicative $\Pi$-types to do this.

To be exact, suppose we had a rule for the $\Pi$-type as follows:

\[
\Gamma \triangleright (x : (C \rightarrow U_j) \rightarrow \Phi(x)) = (x : (C' \rightarrow U_j) \rightarrow \Phi'(x)) \in U_j
\]

\[
x : C \rightarrow U_j \triangleright \Phi(x) = \Phi'(x) \in U_j
\]

\[
\triangleright C = C' \in U_{j-1}.
\]

Then, under the assumptions $x : C \rightarrow U_j \triangleright \Phi(x) \in U_j$ and $\triangleright C \in U_{j-1}$, we could prove $(x : (C \rightarrow U_j) \rightarrow \Phi(x)) \in U_j$. $N^+ \rightarrow (\{1, \ldots, m\}^* \rightarrow U7)$ is a currying of $(N^+ \times \{1, \ldots, m\}^*) \rightarrow U7$.

### 5.2 Extending Mendler’s Work

Mendler [Men88] justifies inductive and impredicative type constructors via a Girard *candidats de reductibilite* argument [Gir71,Gir72]. In the rest of this chapter, we will prove that the impredicative $\Pi$-type is sound by reconstructing Mendler’s semantics for Nuprl, and extending it with a semantic model of the impredicative $\Pi$-type. We will skip over most of the results, as they are found in Mendler’s thesis, and instead concentrate on building enough of a structure for us to express our extensions. In the interests of brevity and of giving credit where credit is due, we refer readers interested in a more thorough understanding of the proof of soundness to Mendler’s thesis. Significant parts of this presentation are taken from Mendler’s thesis, to maintain a close connection with the presentation used there. Most of this chapter is “plug-compatible” with Mendler’s thesis. Mendler’s model for Nuprl is much like Allen’s [All87]. Types are modeled as partial equivalence relations (PERs) on closed, normalizing terms. We model a type system as a partial equivalence relation which tells us when two terms denote the same type, and a mapping from terms to partial equivalence relations which assigns to each term the PER which it represents (if any).

The plan of this chapter is as follows:
Section 5.3 will outline a predicative semantics for Nuprl

Section 5.4 will extend this semantics to account for a simplified version of the impredicative II-type, and

Section 5.5 will address how to extend this result to the general II-type that we discussed above.

5.3 Semantics of Predicative Nuprl

In this section, we develop Mendler's semantics for predicative Nuprl. This work is wholly taken from Mendler's thesis.

5.3.1 Ground Relations

We begin with the definition of the semantic objects which model types. These are partial equivalence relations over the closed, normalizing terms of Nuprl. Let $T$ be the set of terms of Nuprl.

Definition 5.3.1.

- A binary relation $\xi$ over a set $T$ is a partial equivalence relation (PER) iff it is symmetric and transitive. $T$ may be dropped if it is clear from context.

- If $\xi$ is a PER over $T$, let $T/\xi$ be the set of nonempty equivalence classes of $\xi$. For $a \in T$, define:

$$[a]_{\xi} \equiv \{b : T | a \xi b\}.$$ 

If $\xi$ is clear from context, we may write $[a]$. Let $Fi(\xi)$, the field of a relation $\xi$, be $\{\{b : T | b \xi b\}\}$. Note that PER $\xi$ is an equivalence relation over $Fi(\xi)$.

- If $R$ is a relation over $T$, then PER $\xi$ respects $R$ iff

$$(a \in Fi(\xi) \lor b \in Fi(\xi)) \land aRb \Rightarrow a\xi b.$$ 

- For a collection $S$ of PERs over $T$, and a given PER $\xi$, define

$$FAM(\xi, S) \equiv (T/\xi) \rightarrow S$$

$$FAM(S) \equiv \bigcup_{\xi \in S} FAM(\xi, S)$$

$$IFAM(S) \equiv \sum_{\xi \in S} FAM(\xi, S).$$

Let $\succ$ be the evaluation relation on $T$, and $\mathcal{V}$ the set of closed, normalizing terms. Define $a \downarrow \equiv a \in \mathcal{V}$, and $a \simeq b \equiv \exists c. a \succ c \land b \succ c$. Let $\Xi$ be the collection of PERs over $\mathcal{V}$ that respect $\succ$. Members of $\Xi$ are called ground relations. The metavariables we will use are as follows:
• $i, j$ and $k$ range over positive numbers.

• $v, w, x, y$ and $z$ range over variables.

• $a, b, c$ and $d$ and $A, B, C$ and $D$ range over open terms.

• $\langle \Xi, \psi \rangle$ ranges over $IFAM(\Xi)$.

• $\sigma$ and $\langle \tau, [\cdot] \rangle$ also range over $IFAM(\Xi)$.

The choice of an upper or lower case Roman letter for a term is meant to denote whether a term is being used as a type or as an element of a type, but, of course, there is no semantic distinction. We define the following notation on PERs:

\[
A \in \xi \equiv A \xi A \\
A = A' \in \xi \equiv A \xi A'
\]

\[
\forall A \in \xi P \equiv \forall A. A \xi A \Rightarrow P
\]

\[
\forall A = A' \in \xi P \equiv \forall A, A'. A \xi A' \Rightarrow P
\]

\[
[A] \equiv [[A]]
\]

Operations on Ground Relations

Mendler defines operations (his Definition 4.1) on ground relations and families of ground relations ($IFAM(\Xi)$) corresponding to the predicative II-type and $\Sigma$-type, I-type, set-type, and disjoint-union type. We will show only the II-type and I-type.

**Definition 5.3.2 (Semantic II Operator)** $\Pi \in IFAM(\Xi) \rightarrow \Xi$ is:

\[
a = a' \in \Pi(\xi, \psi) \equiv (a \in \mathcal{V}) \land (a' \in \mathcal{V}) \land (\forall b = b' \in \xi. a(b) = a'(b') \in \psi[b])
\]

**Definition 5.3.3 (Semantic I-type Operator)** $I \in \mathcal{T} \times \mathcal{T} \times \Xi \rightarrow \Xi$ is:

\[
a = a' \in I(b, b', \xi) \equiv a \supset \text{axiom} \land a' \supset \text{axiom} \land b \xi b'
\]

Note that in the definition of the semantic I-type operator, we use the token axiom instead of true as Mendler does, for compatibility with the rest of the presentation of Nuprl.

### 5.3.2 Type Systems

Define the collection of type systems by $\mathcal{T}S \equiv IFAM(\Xi)$. The notation for a type system is $\langle \tau, [\cdot] \rangle$, or simply $\sigma$. $IFAM(\Xi)$ is a CPO under the ordering

\[
\langle \xi, \psi \rangle \sqsubseteq \langle \xi', \psi' \rangle \equiv \xi \subseteq \xi' \land (\forall A \in \xi. \psi([A]) = \psi'([A]))
\]
with least element \( \langle \emptyset, \emptyset \rangle \) and the least upper bound of a chain being:

\[
\bigcup_{\alpha \in I} \langle \xi_\alpha, \psi_\alpha \rangle \equiv \langle \bigcup_{\alpha \in I} \xi_\alpha, \bigcup_{\alpha \in I} \psi_\alpha \rangle.
\]

\(\Xi\) captures the meaning of a type in that members of \(\Xi\) are binary, symmetric, transitive relations on normalizing terms which respect evaluation. These are all properties that any model of a type must satisfy. In the end, as Mendler observes, the true test of a model is whether it allows us to prove the things we wish to prove, and in this respect this model is quite acceptable.

The meaning of an intensional type system comes in two parts. First, we have a relation of type equality, and for each class of equal types, a membership relation. Thus we see that \(IFAM(\Xi)\) captures this exactly. Finally, the ordering \(\subseteq\) is the appropriate one on type systems, as \(\langle \xi, \psi \rangle \subseteq \langle \xi', \psi' \rangle\) means that types equated by \(\langle \xi, \psi \rangle\) are equated by \(\langle \xi', \psi' \rangle\) and they retain the same membership relation.

\(EqFam\) is a predicate which asserts that, for a given type system, the arguments are associated with equal families of types.

**Definition 5.3.4** (\(EqFam\)) For \(\langle \tau, [\_], \rangle \equiv \sigma \in TS, B, B', C, C' \in T\), define

\[
EqFam(\sigma, B, B', C, C') \equiv B \tau B' \land \forall b \in [B].(C(b))\tau(C'(b')),
\]

and let \(Fam(\sigma, B, C) \equiv EqFam(\sigma, B, B, C, C)\).

Mendler then asserts

**Lemma 5.3.1.** 1. \(EqFam\) and \(Fam\) are monotone in \(TS\).

2. If \(EqFam(\langle \tau, [\_], \rangle, B, B', C, C')\) then \(\langle [B], \lambda [b].[C(b)] \rangle\) and \(\langle [B'], \lambda [b].[C'(b')] \rangle\) are equal and an element of \(IFAM(\Xi)\).

The content of these two assertions is that \(EqFam\) picks out truly equal families of types, and that they are well-behaved under type-system extension, that is, if \(\sigma \subseteq \sigma'\), then \(EqFam(\sigma, B, B', C, C')\) implies \(EqFam(\sigma', B, B', C, C')\).

**Constructing Type Systems**

At this point, we construct type systems \(\sigma_\alpha\), for each \(\alpha \in \omega\). Our model for Nuprl will be \(\sigma_\omega\). Mendler defines a type construction function \(F_\alpha\) at each \(0 < \alpha \leq \omega\) by cases across all the constructors of Nuprl. Here we only do so for the II-type, I-type, and universe-type.

**Definition 5.3.5** (\(F_\alpha \) — The Type System Constructor) Fix \(0 \leq \alpha \leq \omega\), and by induction assume \(\sigma_j \equiv \langle \tau_j, [\_], \rangle \in TS\) for \(0 \leq j < \alpha\) are defined. Define \(F_\alpha \in TS \rightarrow TS\) by:

\[
F_\alpha(\langle \tau, [\_], \rangle) \equiv \langle \tau', [\_]' \rangle,
\]

where for a given \(\sigma \equiv \langle \tau, [\_], \rangle\), we define \(A = A' \in \tau'\) iff one of the following holds; and if so, define \([A]'\) by the corresponding \(\bullet\) clause:
1. (a) \( A \supset (x : B \rightarrow C) \land A' \supset (x : B' \rightarrow C') \)
   
   (b) \( EqFam(\sigma, B, B', \lambda x.C, \lambda x.C') \)
   
   • \([x : B \rightarrow C]' \equiv \Pi([B], \lambda [b].[C[b/x]])\)

2. (a) \( A \supset I(b, c, B) \land A' \supset I(b', c', B') \)
   
   (b) \( B \supset B' \land b = b' \equiv \frac{B}{B} \land c = c' \equiv \frac{B}{B} \)
   
   • \([I(b, c, B)]' \equiv I(b, c, [B])\)

Let \( \sigma_\alpha \equiv (\tau_\alpha, [\_]_\alpha) \), the least fixed point of \( F_\alpha \).

One must, of course, verify the well-formedness of the definition (i.e. that \( F_\alpha \) is indeed a monotonic function), but this is simply a routine case analysis.

**Properties of \( \sigma_\alpha \)**

We can prove several nice properties of our definition. First is cumulativity, which tells us that \( \alpha \leq \beta \) implies \( \sigma_\alpha \sqsubseteq \sigma_\beta \). This follows from \( F_\alpha \sqsubseteq F_\beta \). We then define the familiar relations of typehood and type-membership:

**Definition 5.3.6.**

\[
A = B \equiv A_{\tau_A} B \\
A Type \equiv A = A \\
a = b \in C \equiv C \in \tau_\alpha \land a = b \in [C]_\omega \\
a \in C \equiv a = a \in C
\]

Note that \( A = B \in U_i \Leftrightarrow A = B \in \tau_i \).

Mendler goes on to prove many lemmas which are used in the soundness proof. We list a few of them below:

\[
(x : B \rightarrow C) = (x : B' \rightarrow C') \Leftrightarrow B = B' \land \forall b = b' \in B.C[b/x] = C'[b'/x] \\
(x : B \rightarrow C) = (x : B \rightarrow C) \in U_i \Leftrightarrow B = B' \\
\land \forall b = b' \in B.C[b/x] = C'[b'/x] \in U_i \\
a = a' \in x : B \rightarrow C \Leftrightarrow x : B \rightarrow CT_{type} \land a \downarrow \land a' \downarrow \\
\land \forall b = b' \in B.a(b) = a'(b') \in C[b/x] \\
I(b, c, B) = I(b', c', B') \Leftrightarrow B = B' \land b = b' \in B \land c = c' \in B \\
I(b, c, B) = I(b', c', B') \in U_i \Leftrightarrow B = B' \in U_i \land b = b' \in B \\
\land c = c' \in B \\
a = a' \in I(b, c, B) \Leftrightarrow a \succ axiom \land a' \succ axiom \\
\land b = c \in B \\
\neg(a = a' \in void)
\]
5.3.3 Truth and Soundness

Having built $\sigma_\omega$, we define truth for sequents and show that the proof rules are sound.

Let's introduce some abbreviations. For $k \in \{1, 2\}$, write $a_k$ for $a_{k,1}, \ldots, a_{k,n-1}$, and for a given open term $c$, write $[c]_k$ for $c[a_k/x_1, \ldots, x_{n-1}]$. Write $\Gamma$ for the context $x_1 : A_1, \ldots, x_{n-1} : A_{n-1}$. Two vectors of terms are equal in a context when:

Definition 5.3.7. 1. $a_1 = a_2 \in \Gamma$ is defined by induction on $\Gamma$. The empty case is true; otherwise define $a_1, a_{1,n} = a_2, a_{2,n} \in \Gamma, x_n : A_n$ by:

$$a_1 = a_2 \in \Gamma \land [A_n]_1 = [A_n]_2 \land a_{1,n} = a_{2,n} \in [A_n]_1.$$  

2. $\Gamma \models b = b' \in B$ is defined by:

$$\forall a_1 = a_2 \in \Gamma, [B]_1 = [B]_2 \land [b]_1 = [b]_2 \in [B]_1.$$  

Lemma 5.3.2. $a_1, b = a_2, b' \in \Gamma$ is implied by

1. $a_1 = a_2 \in \Gamma$
2. $\Gamma \models B = B' \in U_j$
3. $b = b' \in [B]_1$.

To prove the soundness of the Nuprl type rules, we show by an induction on derivations that if $\Gamma \gg a = b \in A$ then $\Gamma \models a = b \in A$. (void) for any $a$ tells us that the semantics is consistent, and these two facts together allow us to conclude consistency.

5.4 Simple Impredicative $\Pi$-Types

A simple impredicative $\Pi$-type is a type like $x : U_j \rightarrow \Phi(x)$ which is a member of $U_j$. In this section we will modify the work of the previous section to account for simple impredicative $\Pi$-types. The development here was taken from chapter 5 of Mendler's thesis. The only original work here is the impredicative $\Pi$-type. The rules we will justify are the following:

$$\Gamma \gg (x : U_j \rightarrow \Phi(x)) = (x : U_j \rightarrow \Phi'(x)) \in U_j$$

$$x : U_j \gg \Phi(x) = \Phi'(x) \in U_j$$

and

$$\Gamma \gg \lambda x. c = \lambda x. c' \in (x : U_j \rightarrow B)$$

BY intro

$$x : U_j \gg c = c' \in B$$

$$\gg (x : U_j \rightarrow B) \in U_j$$
5.4.1 Ground Relations and Ground Types

In this section we will need a specialized notion of partial equivalence relations over terms, parameterized by a set of constants and with a distinguished constant $\Omega$. Assume we have fixed the term constructors and rules of reduction for the type theory, and suppose we are given a set $S$ of fresh constants, intended to represent atomic types. Terms are as before, with the possibility of occurrences of $\Omega$ or members of $S$ in them. $\Omega$ and each new constant evaluate only to themselves. Define $\text{PER}(S)$ to be the partial equivalence relations $\xi$ over normalizing terms that respect $>$ and ignore $\Omega$, that is, $a[\Omega/x] \xi a[b/x]$ for any $a[\Omega/x] \in Fi(\xi)$ and $b \in \mathcal{V}$. The idea here is that $\Omega$ represents any arbitrary, unspecified normalizing term, so of course any instance of $\Omega$ can be replaced by a normalizing term and and the result will be an equal term.

We continue with a series of definitions which culminate in the definition of the ultimate sets of terms and ground relations:

- Let $C_0 \equiv \emptyset$ be the set of closed terms with possible occurrences of $\Omega$, and let $\mathcal{V}_0$ be the normalizing terms of $T_0$.

- For $i > 0$, define $C_i, T_i, \mathcal{V}_i$ by induction on $i$. Let $C_{<i} \equiv \bigcup_{j<i} C_j$, and let $C_i$, the ground types of level $i$, be a set of new constant symbols, two for each element of $\text{PER}(C_{<i})$. Let $X_i, Y_i$ and $Z_i$ range over $C_i$. Let $T_i$ be the set of closed terms constructed with possible occurrence of $\Omega$ and the constants in $C_{<i} \cup C_i$. Finally, let $\mathcal{V}_i$ be the normalizing terms of $T_i$.

- Let $\mathcal{C} \equiv \bigcup_j C_j$, $T \equiv \bigcup_j T_j$, $\mathcal{V} \equiv \bigcup_j \mathcal{V}_j$. $T$ is the set of terms constructed with possible occurrences of constants in $\mathcal{C}$ and $\mathcal{V}$ are the normalizing terms in $T$.

- For $i \geq 0$, define $|a|_i \in T \rightarrow T_i$ (a kind of stripping operation) by letting $|a|_i$ be the term constructed from $A$ by replacing every instance of any ground type of level $i+1$ or higher by $\Omega$.

- Let $\Xi \equiv \text{PER}(\mathcal{C})$. For $i > 0$ define the ground relations of level $i$ by:

$$\Xi_i \equiv \{ \xi : \exists a \in Fi(\xi).a \xi |a|_{i-1} \}.$$  

- For $i > 0$, define $.-|i \in \Xi_i \rightarrow \text{PER}(C_{<i})$ to be the restriction to $\mathcal{V}_{i-1} \times \mathcal{V}_{i-1}$.

The motivation behind these definitions is to allow quantification over all types, without introducing circularity. By introducing a “name” for each PER, we can quantify over the names when defining a PER and not get caught in a circularity. We introduce two names for each PER at each level in order to achieve an intensional...
type theory. That is, the two types \( \{ x : T | \text{True} \} \) and \( T \) are not provably equal, and we wish for the theory to take account of this.\(^1\)

We can show that \( \text{PER}(C_{<i}) \) approximates \( \Xi_i \) as follows:

**Lemma 5.4.1.** The functions \( \cdot | i \in \Xi_i \to \text{PER}(C_{<i}) \) are bijections.

\( C_i \) was defined so that two constants in it were associated with each member of \( \text{PER}(C_{<i}) \), so by the preceding lemma we can transfer this association to \( \Xi_i \): let \( \text{val}_i \in C_i \to \Xi_i \) be this association, and let \( \text{rep}_i \in \Xi_i \to C_i \) be a right inverse of \( \text{val}_i \). Mendler then shows

**Lemma 5.4.2.** Each \( \Xi_i \) is a complete lattice under \( \subseteq \).

So we finally have a universe of ground relations upon which to build our impredicative construction. Mendler then shows

**Lemma 5.4.3.** When \( \Pi, \Sigma \) (the analog of the semantic \( \Pi \)-type operator), or \( \{ \cdot | \cdot \} \) are restricted to \( \text{IFAM}(\Xi_i) \), its range is \( \Xi_i \). \( I \)'s range is within \( \Xi_1 \).

### 5.4.2 Type Systems

As before \( TS \equiv \text{IFAM}(\Xi) \). The definitions of \( \text{EqFam} \) and \( \text{Fam} \) are well-formed here, and lemma 5.3.1 still hold.

We now repeat the construction of type systems, with the addition of more cases to handle to ground types we introduced, and also the impredicative \( \Pi \)-type. We will display the cases for the new constants and the impredicative \( \Pi \)-type. First, though, we need a semantic impredicative \( \Pi \)-type operator:

**Definition 5.4.1 (Semantic Impredicative \( \Pi \) Operator)** For \( 0 < \alpha < \omega \), define \( \Pi_\alpha \in (\Xi_\alpha \to \Xi_\alpha) \to \Xi_\alpha \) by:

\[
a = a' \in \Pi_\alpha(\phi) \equiv a \downarrow \land a' \downarrow \land \forall \xi \in \Xi_\alpha . a(\text{rep}_\alpha(\xi)) = a'(\text{rep}_\alpha(\xi)) \in \phi(\xi).
\]

It is easy to show that \( \Pi_\alpha \) is well-formed, that is, maps \( \Xi_\alpha \to \Xi_\alpha \) into \( \Xi_\alpha \).

**Constructing Type Systems**

Finally, we construct the Nuprl type system.

---

\(^1\)One could imagine a semantics in which this was not the case, though, and in which type equality were purely extensional. Some of this presentation would be simplified in that case, but we would diverge from Mendler's presentation, as well as from the intended semantics of Nuprl, so we will proceed with an intensional theory. It turns out that the only place where this difference show up is in a substitution lemma, Lemma 5.4.5.
Definition 5.4.2 \((G_\alpha \text{ Defined})\) Fix \(0 < \alpha \leq \omega\) and by induction assume \(\sigma_j \equiv \langle \tau_j, [\_], j \rangle \in TS\) for all \(0 \leq j < \alpha\) are defined. Define \(G_\alpha \in TS \xrightarrow{\text{mon}} TS\) by:

\[G_\alpha(\langle \tau, [\_] \rangle) \equiv \langle \tau', [\_]' \rangle,\]

where for a given \(\sigma \equiv \langle \tau, [\_]\rangle\), we define \(A = A' \in \tau'\) iff one of the following hold; and if so, define \([A]'\) by the corresponding \(\bullet\) clause:

1. \(\exists j \leq \alpha. A \supset X_j \land A' \supset X_j\)
   
   \(\bullet\) \([A]' \equiv val_j(X_j)\)

2. \((a)\) \(\alpha < \omega \land A \supset x : U_\alpha \rightarrow B \land A' \supset x : U_\alpha \rightarrow B'\)
   
   \((b)\) \(\forall X_\alpha \in C_\alpha. B[X_\alpha/x] = B'[X_\alpha/x] \in \tau\)
   
   \(\bullet\) \([x : U_\alpha \rightarrow B]' \equiv \Pi_\alpha(\lambda \xi. [B[\text{rep}_\alpha(\xi)/x]])\)

Well-formedness Proof: We must now verify the well-formedness of the definitions presented. The only new case here is the impredicative \(II\)-type.

Fix \(\sigma_p \equiv \langle \tau_p, [\_]_p \rangle \subseteq G_\sigma \equiv \langle \tau_q, [\_]_q \rangle\), and write \(\sigma'_p \equiv G_\alpha(\sigma_p)\) and \(\sigma'_q \equiv G_\alpha(\sigma_q)\). Assume \((x : U_\alpha \rightarrow B) = (x : U_\alpha \rightarrow B') \in \tau'_p\). First, one would check the well-definedness of \([x : U_\alpha \rightarrow B]'_p\), but this is clear; condition 2 guarantees that for any \(X_\alpha\), \(B[X_\alpha/x] = B'[X_\alpha/x] \in \tau_p\), and this tells us immediately that

\[\Pi_\alpha(\lambda \xi. [B[\text{rep}_\alpha(\xi)/x]]) = \Pi_\alpha(\lambda \xi. [B'[\text{rep}_\alpha(\xi)/x]]).\]

As for the monotonicity of \(G_\alpha\), \(\sigma_p \subseteq \sigma_q\), so we must show \((x : U_\alpha \rightarrow B)_p(x : U_\alpha \rightarrow B')\) implies \((x : U_\alpha \rightarrow B)_q(x : U_\alpha \rightarrow B')\). But this is easy, because the antecedent shows \(\forall X_\alpha \in C_\alpha. B[X_\alpha/x] = B'[X_\alpha/x] \in \tau_p\), so \(\forall X_\alpha \in C_\alpha. B[X_\alpha/x] = B'[X_\alpha/x] \in \tau_q\), thence \((x : U_\alpha \rightarrow B)_q(x : U_\alpha \rightarrow B')\).

Finally, we must show that \(\sigma'_q\) associates the same membership relation with equal types that \(\sigma'_p\) does. This follows easily from the definition of \(\Pi_\alpha\).

Properties of \(\sigma_\alpha\)

We can now define iterates of \(G_\alpha\):

**Definition 5.4.3.** Define the iterates of \(G_\alpha\) as follows:

\[
\sigma_\alpha^0 \equiv \langle \tau_\alpha^0, [\_]^0_\alpha \rangle \equiv \perp TS = \langle \emptyset, \emptyset \rangle
\]

\[
\sigma_\alpha^{\gamma + 1} \equiv \langle \tau_\alpha^{\gamma + 1}, [\_]^{\gamma + 1}_\alpha \rangle \equiv G_\alpha(\sigma_\alpha^\gamma)
\]

\[
\sigma_\alpha^\lambda \equiv \langle \tau_\alpha^\lambda, [\_]^{\gamma + 1}_\alpha \rangle \equiv \sqcup_{\gamma < \lambda} \sigma_\alpha^\gamma \text{ for limit } \lambda
\]

Thus, \(\sigma_\alpha \equiv \sqcup \sigma_\alpha^\gamma\).
Mendler goes on to show that, where $\sigma_\alpha \equiv \langle \tau_\alpha, \llbracket \cdot \rrbracket_\alpha \rangle$ is the least fixed point of $G_\alpha$, it is the case that $\alpha < \beta \leq \omega$, $\sigma_\alpha \subseteq \sigma_\beta$. That is, the type system construction indeed forms a chain. Thus, we can use the type system $\sigma_\omega \equiv \langle \tau_\omega, \llbracket \cdot \rrbracket_\omega \rangle$ as a model of NuPrl, and define the relations of typehood and membership in the same fashion as in definition 5.3.6. The same facts carry over about this definition as for the previous one, and we omit any listing here, as a representative sample are listed on page 44. However, we would like to prove some of these same facts about impredicative II-types. To do this, though, we will need a slightly complicated lemma, whose use will only be apparent later on. It is a substitution lemma, and it is essential to proving the correct properties about impredicative II. The proof of this lemma will depend on the following fact:

Let $\succ_\omega$ be the normal evaluation relation, augmented with the rule that a free variable in a term evaluates to itself. Then

**Lemma 5.4.4.** For $D \in T$, if $A \succ_\omega B$, and $B$ is not $y$ (a variable), then $A[D/y] \succ_\omega B[D/y]$; and if $A \succ_\omega y$ and $D \succ_\omega E$ then $A[D/y] \succ_\omega E$.

Define the notion of extensional equality on types.

**Definition 5.4.4.** For $A$ Type and $B$ Type,

$$ A =_\varepsilon B = \llbracket A \rrbracket_\omega = \llbracket B \rrbracket_\omega $$

We can now state and prove the following:

**Lemma 5.4.5 (Substitution Lemma)** $\forall i. \forall \gamma. \forall Y_i, Z_i \in C_i$, where $Y_i$ and $Z_i$ are distinct, yet extensionally equal, $\forall A[Y_i/y], A'[Y_i/y], D_1, D_2 \in T$ such that $D_1 = D_2 \in U_i$ and $D_1 =_\varepsilon Y_i$,

$$ A[Y_i/y] = A'[Y_i/y] \in \tau_i^\gamma \text{ and } A[Z_i/y] = A'[Z_i/y] \in \tau_i^\gamma \text{ implies } $$


**Proof:** Fix $i$ and induct on $\gamma$. Fix $Y_i, Z_i, A, A', D_1, D_2$. For a given open term $E$, let $[E]_Y \equiv E[Y_i/y], [E]_Z \equiv E[Z_i/y], [E]_1 \equiv E[D_1/y], [E]_2 \equiv E[D_2/y]$.

The proof is by cases on the clauses of the definition of $G_i$, and we do only a few cases, notably the cases for impredicative II.

1. Suppose $A \succ_\omega void$ and $A' \succ_\omega void$, or $A \succ_\omega U_j$ and $A' \succ_\omega U_j$. These clauses are trivial.

2. Suppose $A \succ_\omega X_j$ and $A' \succ_\omega X_j$, or $A \succ_\omega y$ and $A' \succ_\omega y$. Both cases are immediate, but note how cases like $A \succ_\omega X_i, A' \succ_\omega y$ are ruled out by $Y_i$ and $Z_i$ being nonequal types. In fact, this is the sole purpose of having extensionally equal but intensionally distinct ground types.
3. Suppose $A \succ x : B \# C$, $A' \succ x : B' \# C'$. By definition:

(a) $[B]_Y = [B']_Y \in U_i$ (let $\xi \equiv [[[B]_Y]]_i$)
(b) $\forall b = b' \in \xi.[C]_Y[b/x] = [C']_Y[b'/x] \in U_i$
(c) $[B]_Z = [B']_Z \in U_i$ (let $\xi' \equiv [[[B]_Z]]_i$
(d) $\forall b = b' \in \xi.[C]_Z[b/x] = [C']_Z[b'/x] \in U_i$

By induction:

(e) $[B]_1 = [B]_2 \in U_i$
(f) $[B]_1 = \varepsilon [B]_Y = \varepsilon [B]_Z$
(g) $\forall b = b' \in \xi.[C]_1[b/x] = [C']_2[b'/x] \in U_i$
(h) $\forall b \in \xi.[C]_1[b/x] = \varepsilon [C]_Y[b/x] = \varepsilon [C]_Z[b/x]$

From (e) and (g), we conclude $[A]_1 = [A']_2 \in U_i$, and from (f) and (h), $[A]_1 = \varepsilon [A]_Y = \varepsilon [A]_Z$.

4. Suppose $A \succ x : U_i \rightarrow B$ and $A' \succ x : U_i \rightarrow B'$.

Assumptions:

(a) $[A]_Y = [A']_Y \in U_i$
(b) $[A]_Z = [A']_Z \in U_i$

Show:

(c) $[A]_1 = [A']_2 \in U_i$
(d) $[A]_1 = \varepsilon [A]_Y = \varepsilon [A]_Z$.

We can conclude:

(e) $\forall X_i \in C_i.[B]_Y[X_i/x] = [B']_Y[X_i/x] \in U_i$ from $[A]_Y = [A']_Y \in \tau_i^Y$ by definition of (a).
(f) $\forall X_i \in C_i.[B]_Z[X_i/x] = [B']_Z[X_i/x] \in U_i$ by definition of (b).

By induction we can show

$\forall X_i \in C_i.[B]_1[X_i/x] = [B']_2[X_i/x] \in U_i$

by choosing $A \equiv [B]_1$, $A' \equiv [B]_2$, $D_1 \equiv D_2 \equiv X_i$, $Y_1 \equiv Y_i$.

From this we can prove (c) easily.

By induction again we can show

$[B]_1[X_i/x] = \varepsilon [B]_Y[X_i/x] = \varepsilon [B]_Z[X_i/x]$

by choosing $A \equiv B[X_i/x]$, $A' \equiv B'[X_i/x]$, $D_i \equiv D_i$, $Y_i \equiv Y_i$, $Z_i \equiv Z_i$.

And again, from this we can prove (d) easily.
5.4 Simple Impredicative \( \Pi \)-Types

The other cases can be found in Mendler's thesis, Lemma 5.8. With this lemma in hand, we can prove the required properties of the impredicative \( \pi \)-type.

**Lemma 5.4.6.**

\[
x : U_\alpha \rightarrow B = x : U_\alpha \rightarrow B' \in U_\alpha \\
\iff \forall A = A' \in U_\alpha. B[A/x] = B'[A'/x] \in U_\alpha
\]

**Proof:** Suppose \( x : U_\alpha \rightarrow B = x : U_\alpha \rightarrow B' \in U_\alpha \). Then by definition:

(a) \( \forall X_\alpha \in C_\alpha. B[X_\alpha/x] = B'[X_\alpha/x] \in U_\alpha \)

(b) \( \forall \xi. \lambda \xi.[B[\text{rep}_\alpha(\xi)/x]] \in \Xi_\alpha \rightarrow \Xi_\alpha \)

Fix \( A = A' \in U_\alpha \). Let \( X_j = \text{rep}_j([A]_j) \). By lemma 5.4.5, \( B[A/x] = B'[A'/x] \in U_\alpha \). For the converse, suppose:

\( \forall X_\alpha \in C_\alpha. B[X_\alpha/x] = B'[X_\alpha/x] \in \tau \).

Since \( X_\alpha = X_\alpha \in U_\alpha \), (a) implies

\( \forall X_\alpha \in C_\alpha. B[X_\alpha/x] = B'[X_\alpha/x] \in U_\alpha \).

Then there must be some type system \( \tau_\alpha \) such that the equality holds in that type system (e.g. \( \forall X_\alpha \in C_\alpha. B[X_\alpha/x] = B'[X_\alpha/x] \in \tau_\alpha \)), and in \( \tau_\alpha \), \( x : U_\alpha \rightarrow B = x : U_\alpha \rightarrow B' \in \tau_\alpha \) will hold.

Likewise, we can prove:

**Lemma 5.4.7.**

\[
a = a' \in x : U_\alpha \rightarrow B \iff (x : U_\alpha \rightarrow B)\text{Type} \land a \downarrow \land a' \downarrow \\
\land \forall A = A' \in U_\alpha. a(A) = a'(A') \in U_\alpha
\]

**Proof:** Suppose \( a = a' \in x : U_\alpha \rightarrow B \). First of all, \( (x : U_\alpha \rightarrow B)\text{Type} \) is clear from the definition of \( \mathcal{G}_\alpha \). Likewise, \( a \downarrow \) and \( a' \downarrow \). By definition,

\( \forall X_\alpha \in C_\alpha. a(X_\alpha) = a'(X_\alpha) \in U_\alpha \).

Fix \( A = A' \in U_\alpha \). Let \( X_\alpha = \text{rep}_j([A]_j) \). By Lemma 5.4.5, we can show \( a(A) = a(X_\alpha) \in U_\alpha \), and likewise \( a'(X_\alpha) = a'(A') \in U_\alpha \). These together justify \( a(A) = a'(A') \in U_\alpha \).

For the converse, suppose \( (x : U_\alpha \rightarrow B)\text{Type} \) and for all \( A = A' \in U_\alpha \), \( a(A) = a'(A') \in U_\alpha \). First of all, the preconditions on the appropriate clause in \( \mathcal{G}_\alpha \) are met easily for that is the meaning of \( (x : U_\alpha \rightarrow B)\text{Type} \). Likewise, \( a \downarrow \) and \( a' \downarrow \), and we need show:

\( \forall \xi \in \Xi_\alpha. a(\text{rep}_\alpha(\xi)) = a'(\text{rep}_\alpha(\xi)) \in B[\text{rep}_\alpha(\xi)/x] \).

Fix \( \xi \). Let \( X_\alpha = \text{rep}_\alpha(\xi) \). Then by assumption we have \( a(X_\alpha) = a'(X_\alpha) \in U_\alpha \), and we are done. \( \blacksquare \)
Truth and Soundness

Again, we must define the notions of truth and soundness, but these follow almost identically to the previous presentation, now that we have isolated the properties presented in the last section. As an example, we will verify the soundness of the rule of impredicative \( \Pi \)-formation:

**Lemma 5.4.8.** \( \Gamma \models x : U_i \rightarrow B = x : U_i \rightarrow B' \in U_i \) is implied by

\[ \Gamma, x : U_i \models B = B' \in U_i \]

**Proof:** Trivial from Lemma 5.4.6.

Likewise, we can prove

**Lemma 5.4.9.** \( \Gamma \models \lambda x.c = \lambda x.c' \in x : U_i \rightarrow B \) is implied by

1. \( \Gamma, x : U_i \models c = c' \in B \)
2. \( \Gamma \models (x : U_i \rightarrow B) \in U_i. \)

**Proof:** Again, trivial from the properties previously established.

### 5.5 Parameterized Impredicative \( \Pi \)-Types

The proof of Higman's Lemma uses parameterized impredicative \( \Pi \)-types in \( U_j \) of the form \( f : C \rightarrow U_j \rightarrow \Phi(f) \). It is a relatively simple matter to extend the previous results to this type constructor.

The extension again is due to Mendler, and, roughly speaking, goes as follows:

1. Instead of \( C_i \) being a set of fresh type constants, two for every member of \( \text{PER}(C_{<i}) \), we let \( C_i \) be a set of fresh type family constants, two for each member of \( \text{FAM}(\text{PER}(C_{<i})) \). These constants are more appropriately called ground type families.

2. Evaluation will treat \( X_i \in C_i \) as a function constant, i.e.

\[
\frac{a \triangleright X_i}{a(b) \triangleright X_i(b)}
\]

3. the mappings \( \text{val}_i \) and \( \text{rep}_i \) can be moved up to families also. Let \( \text{dom}(X_i) \equiv \xi \), where \( \xi \) is the relation in \( \Xi_i \) such that \( \text{val}_i(X_i) \in \text{FAM}(\xi, \Xi_i) \).

4. Define \( \Pi_\alpha \), the parameterized impredicative \( \Pi \)-type operator, as follows:

**Definition 5.5.1.** For \( 0 < \alpha < \omega \), \( \xi \in \Xi_\alpha \) define \( \Pi_\alpha(\xi) \in (\text{FAM}(\xi, \Xi_\alpha) \rightarrow \Xi_\alpha) \rightarrow \Xi_\alpha \) by:

\[
a = a' \in \Pi_\alpha(\xi)(\phi) \equiv a \downarrow \land a' \downarrow \land \forall \xi' \in \text{FAM}(\xi, \Xi_\alpha). a(\text{rep}_\alpha(\xi')) = a'(\text{rep}_\alpha(\xi')) \in \phi(\xi').
\]
5. Alter clause 1 of definition 5.4.2 (the definition of $\mathcal{G}_\alpha$ upon ground type constants) as follows:

(a) $\exists j < \alpha. \, A \succ X_j(c) \land A' \succ X_j(c')$
(b) $c = c' \in dom(X_j)$
• $[X_j(c)]' \equiv \text{val}_j(X_j)([c])$

6. Alter definition 5.4.2 by replacing clause 2 (for simple impredicative $\Pi$-type formation) by:

(a) $\alpha < \omega \land A \succ x : (C \rightarrow U_\alpha) \rightarrow B \land A' \succ x : (C' \rightarrow U_\alpha) \rightarrow B'$
(b) $\tau C' \tau (\text{let } \xi \equiv [C' \tau] \text{ and } \Phi \equiv FAM(\xi, \Xi_\alpha))$
(c) $\forall X_\alpha \in C_\alpha. \text{dom}(X_\alpha) = \xi \Rightarrow B[X_\alpha/x] = B'[X_\alpha/x] \in \tau$
• $[x : U_\alpha \rightarrow B]' \equiv \Pi_\alpha(\xi)(\lambda \xi. [B[\text{rep}_\alpha(\xi)/x]])$

In the style of lemmas 5.4.6 and 5.4.7, via a substitution lemma similar to 5.4.5, we can prove:

Lemma 5.5.1.

$$x : (C \rightarrow U_\alpha) \rightarrow B = x : (C \rightarrow U_\alpha) \rightarrow B' \in U_\alpha$$
$$\Leftrightarrow \forall c = c' \in (C \rightarrow U_\alpha). B[c/x] = B'[c'/x] \in U_\alpha$$

Lemma 5.5.2.

$$a = a' \in x : (C \rightarrow U_\alpha) \rightarrow B \Leftrightarrow (x : (C \rightarrow U_\alpha) \rightarrow B) Type \land a \downarrow \land a' \downarrow$$
$$\land \forall c = c' \in (C \rightarrow U_\alpha). a(c) = a'(c') \in U_\alpha$$

With the facts just listed, we can prove the soundness of the rule of parameterized impredicative quantification as it was formulated in the beginning of this chapter, and thus prove that, indeed, the proof of Higman's Lemma that we described in the formalization chapter 4 is sound.
Chapter 6

The Translations on Proofs and Types

This chapter will explain the Gödel double-negation translation and the Friedman A-translation. We begin with a survey of double-negation translations, and prove an embedding result for one of them (the Kolmogorov translation). We also discuss the Friedman A-translation, with some examples from the Nuprl ruleset. Finally, we show how the combination of the two translations allows one to translate classical proofs of $\Pi^0_2$ formulae into constructive proofs, and show that any function provably total in an appropriate classical logic is already provably total in the analogous constructive logic.

There are two different problems being addressed here. First, there is the problem of exactly which classical logics can be embedded into their intuitionistic counterparts. The Gödel-translation solves this problem for predicate logic, and in fact for number theory. Other extensions suffice for intuitionistic set theory [Fri78]. It turns out, though, that a classical variant of Nuprl cannot be embedded into Nuprl itself, due to the axiom of choice. In Chapter 7, we discuss this problem, along with its solution.

Once we have a constructive proof of the double-negation-translated sentence, we need to recover a proof of the original sentence. This problem was solved by Friedman [Fri78]. He showed that many logics were closed under Markov’s Principle:

$$
\frac{x : \text{Dom} \vdash I \; \neg(\exists y \in \text{Int}. \Phi(x, y)), \; \; x : \text{Dom}, y : \text{Int} \vdash I \; \Phi(x, y) \lor \neg \Phi(x, y)}{x : \text{Dom} \vdash I \; \exists y \in \text{Int}. \Phi(x, y)}.
$$

Markov’s Principle tells us that if we have a constructively decidable predicate over the integers, and a proof that it is contradictory for the predicate to be false for all integers, then the predicate is true for some integer. The “computational content” of the principle is an unbounded search operation, and as a result, many constructivists take exception with this rule. Friedman sets forth a simple, syntactic method of translating a proof of the form $\Gamma \vdash I \; \neg(\exists x. \; \Phi(x))$ into a proof of the form $\Gamma \vdash I \; \exists x. \; \Phi(x)$. He goes on to expound simple syntactic conditions on a theory for
### 6.1 The Double-Negation Translation

The double-negation translation was invented by Gödel [God65] as a way of embedding classical number theory into constructive number theory (to prove the equiconsistency of the two theories). Denote the translation by \((-)^\circ\). He proved that if \(\vdash_C \Phi\) then \(\vdash_I \Phi^\circ\). The Gödel-translation is shown in Figure 6.1. For clarity, we will prove the required embedding result for the Kolmogorov translation, shown in Figure 6.2; the result for the Gödel translation is simple, but cumbersome.

**Theorem 6.1.1 (Kolmogorov's Double-Negation Embedding)**

Given a proof, \(\vdash_C \Phi\) in classical predicate logic, we can effectively generate a proof \(\vdash_I \Phi^\circ\) in constructive predicate logic.

**Proof:** By induction on the height of the proof, show that if \(\Gamma \vdash_C \Phi\), then \(\Gamma \vdash_I \Phi^\circ\). The base cases are trivial, since the only height-one proofs in predicate logic are of

---

**Figure 6.1: The Gödel Translation**

\[
\begin{align*}
(A \lor B)^\circ & \quad \rightarrow \quad \neg (A^\circ \lor B^\circ) \\
(A \land B)^\circ & \quad \rightarrow \quad A^\circ \land B^\circ \\
(\exists x \in A.B)^\circ & \quad \rightarrow \quad \neg (\exists x \in A.B)^\circ \\
(\forall x \in A.B)^\circ & \quad \rightarrow \quad \forall x \in A.B^\circ \\
(A \Rightarrow B)^\circ & \quad \rightarrow \quad A^\circ \Rightarrow B^\circ \\
P^\circ & \quad \rightarrow \quad \neg (P) \quad (P \text{ prime}^1)
\end{align*}
\]

**Figure 6.2: The Kolmogorov Translation**

\[
\begin{align*}
\overline{(A \lor B)} & \quad \rightarrow \quad \neg \overline{(A \lor B)} \\
\overline{(A \land B)} & \quad \rightarrow \quad \neg \overline{(A \land B)} \\
\overline{(\exists x \in A.B)} & \quad \rightarrow \quad \neg \overline{(\exists x \in A.B)} \\
\overline{(\forall x \in A.B)} & \quad \rightarrow \quad \neg \overline{(\forall x \in A.B)} \\
\overline{(A \Rightarrow B)} & \quad \rightarrow \quad \neg \overline{(A \Rightarrow B)} \\
\overline{P} & \quad \rightarrow \quad \neg (P) \quad (P \text{ prime})
\end{align*}
\]

it to admit this translation. He also shows that intuitionistic Zermelo-Fraenkel set theory (IZF) with collection admits this translation, and thus is closed under Markov's Principle. Leivant [Lei85] later extended these results to show that many other theories were closed under Markov's Principle. However, for the purposes of this work, all we need is the original Friedman result.
the form: \( \Gamma, A, \Gamma' \vdash_C A \). We construct \( \overline{\Gamma}, \overline{A}, \overline{\Gamma'} \vdash_I \overline{A} \).

If we have a proof with an instance of modus ponens, then it must look like:

\[
\begin{align*}
&\Gamma \vdash_C B \\
&\quad \Gamma \vdash_C A \Rightarrow B \\
&\quad \Gamma \vdash_C A
\end{align*}
\]

and the translated rule would be:

\[
\begin{align*}
&\overline{\Gamma} \vdash_I \overline{B} \\
&\quad \overline{\Gamma} \vdash_I \neg \neg (\overline{A} \Rightarrow \overline{B}) \\
&\quad \overline{\Gamma} \vdash_I \overline{A}
\end{align*}
\]

The justification for this translated rule makes use of the following lemma, stated without proof:

**Lemma 6.1.1 (Double-Negation Uncovering)** If \( B = \neg(\overline{\Phi}) \), that is, if \( B \) is negative, then \( \Gamma, \neg \neg (H) \vdash_I B \) iff \( \Gamma, H \vdash_I B \).

We must prove: \( \overline{\Gamma}, \neg \neg (\overline{A} \Rightarrow \overline{B}), \overline{A} \vdash_I \overline{B} \). But this is easy now, since the preceding lemma tells us that we need only show: \( \overline{\Gamma}, \overline{A} \Rightarrow \overline{B}, \overline{A} \vdash_I \overline{B} \).

We can likewise justify the translations of all the other rules of predicate logic.

### 6.1.1 Other Properties of Double-Negation Translations

The Kolmogorov-translation is easy to work with because every logical subformula of a translated formula is double-negated, which means that the conclusion of a sequent is double-negated, and we can then use the uncovering lemma to "remove" double-negations from any hypotheses which have them. For the Gödel-translation, though, it's not quite so simple. Imagine that in a classical proof, we prove an implication from a disjunction thus:

\[
\begin{align*}
&\Gamma, A \vee B \vdash_C U \Rightarrow V \\
&\quad \Gamma, A \vdash_C U \Rightarrow V \\
&\quad \Gamma, B \vdash_C U \Rightarrow V
\end{align*}
\]

and the translated proof would be:

\[
\begin{align*}
&\Gamma^\circ, \neg \neg (A^\circ \vee B^\circ) \vdash_I U^\circ \Rightarrow V^\circ \\
&\quad \Gamma^\circ, A^\circ \vdash_I U^\circ \Rightarrow V^\circ \\
&\quad \Gamma^\circ, B^\circ \vdash_I U^\circ \Rightarrow V^\circ
\end{align*}
\]

But to justify this inference, we must show that:

\[
\begin{align*}
&\Gamma^\circ, \neg \neg (A^\circ \vee B^\circ) \vdash_I U^\circ \Rightarrow V^\circ \\
&\quad \Gamma^\circ, A^\circ \vee B^\circ \vdash_I U^\circ \Rightarrow V^\circ
\end{align*}
\]
is a valid inference. And since the conclusion, \( U^\circ \Rightarrow V^\circ \) isn't negative, we cannot use the previous lemma to justify this inference. But we can prove the following lemma, which will indeed allow us to justify the previous inference.

**Lemma 6.1.2 (Double-Negation Covering)*** For all \( \Phi \), for all \( \Gamma \) in which the free variables of \( \Phi \) are declared, \( \Gamma \vdash \Phi^\circ \iff \neg(\neg \Phi^\circ) \).

The proof is by structural induction on the formula \( \Phi \), and we leave it as an exercise for the reader. This lemma allows us to "cover" the conclusion \( U^\circ \Rightarrow V^\circ \) with a double-negation, and then we can use the Double-Negation Uncovering Lemma to justify the rule by uncovering the disjunction. Of course, the size of the proof that is inserted to justify the inference above is dependent on the structure of the goal, which complicates the translation procedure.

There are other translations from classical into constructive logics. The most peculiar is the Kuroda negative translation, which translates a formula \( \Phi \) by double-negating after universal quantification, and at the outside of the formula. Again, for this translation, we are forced to prove a lemma like the Double-Negation Covering Lemma, again by structural induction on the term being "covered", which means that the size of the justification of certain steps will be related to the size of various formulae in the proof step being justified.

As an aside, the reader might have noted that, for a term \( \bar{\Phi} \), \( \bar{\Phi} \) can be, and in practice is, much bigger than \( \Phi^\circ \). However, as we saw, to justify certain proof steps for Gödel-translated terms may take a proof (constructed by structural induction) which is linear in the size of one of the terms in the original proof, whereas for the Kolmogorov translation, this is never the case. Thus, we see that there is a tradeoff here between the size of terms in a proof, and the depth of the proof itself.

A further useful property of all double-negation translations is their redundancy for decidable formulae. That is,

\[
\Gamma \vdash \Phi \lor \neg \Phi \Rightarrow \Gamma \vdash \Phi^\circ.
\]

Again, we can prove this property via structural induction on terms.

### 6.2 The Friedman A-Translation

The Friedman A-translation naturally arises out of the observation that there is a simple mapping from intuitionistic theories to their minimal counterparts. Recall that a minimal logic is one in which there is no rule of the form

\[
\bot \vdash A
\]

where \( A \) is arbitrary. That is, there is no rule allowing one to conclude anything from \( \bot \) except \( \bot \) itself. Another characterization is that \( \bot \) is treated as an uninterpreted
\[
\begin{align*}
(A \Rightarrow B)^A & \mapsto A^A \Rightarrow B^A \\
(A \lor B)^A & \mapsto A^A \lor B^A \\
(A \land B)^A & \mapsto A^A \land B^A \\
(\exists x \in A.B)^A & \mapsto \exists x \in A.(B^A) \\
(\forall x \in A.B)^A & \mapsto \forall x \in A.(B^A) \\
P^A & \mapsto P \lor A \quad (P \text{ prime})
\end{align*}
\]

Figure 6.3: The Friedman A-Translation

propositional symbol. We will write provability in minimal logic as \( \vdash_M \), and we will write \((-)^\perp\) for the following translation on terms:

\[
\begin{align*}
(A \Rightarrow B)^\perp & \mapsto A^\perp \Rightarrow B^\perp \\
(A \lor B)^\perp & \mapsto A^\perp \lor B^\perp \\
(A \land B)^\perp & \mapsto A^\perp \land B^\perp \\
(\exists x \in A.B)^\perp & \mapsto \exists x \in A.(B^\perp) \\
(\forall x \in A.B)^\perp & \mapsto \forall x \in A.(B^\perp) \\
P^\perp & \mapsto P \lor \perp \quad (P \text{ prime})
\end{align*}
\]

It is a relatively easy result that for predicate logic, if \( \Gamma \vdash_I \Phi \) then \( \Gamma^\perp \vdash_M \Phi^\perp \). Since \( \perp \) is being treated as an uninterpreted symbol, let us replace it with a new one, \( A \), to get the A-translation, shown in Figure 6.3. We get the same provability result, that if \( \Gamma \vdash_I \Phi \) then \( \Gamma^A \vdash_M \Phi^A \), and since minimal logic is a subset of intuitionistic logic, \( \Gamma^A \vdash_I \Phi^A \).

6.3 Putting It All Together

As we saw in previous sections, if we have a proof \( \vdash_C \Psi \), then we may conclude \( \vdash_I \Psi^o \), and \( \vdash_I (\Psi^o)^A \). In the case where \( \Psi = \forall x \in \text{Dom.} \exists y \in \text{Rng.} \Phi(x, y) \), and \( \Phi \) is decidable, that is, \( x : \text{Dom}, y : \text{Rng} \vdash_I \Phi(x, y) \lor \neg \Phi(x, y) \), we may conclude \( \vdash_I \forall x \in \text{Dom.} \neg(\exists y \in \text{Rng.} \Phi(x, y)) \). Moreover, the A-translation allows us to infer, for any \( A \),

\[
x : \text{Dom} \vdash_I ((\exists y \in \text{Rng.} \Phi(x, y)^A) \Rightarrow A) \Rightarrow A.
\]

The decidability of \( \Phi \) is used in showing that \( \Phi(x, y)^A \Leftrightarrow (\Phi(x, y) \lor A) \), yielding

\[
x : \text{Dom} \vdash_I ((\exists y \in \text{Rng.} \Phi(x, y) \lor A) \Rightarrow A) \Rightarrow A.
\]

If we let \( A = \exists y \in \text{Rng.} \Phi(x, y) \), then a simple argument allows us to infer

\[
x : \text{Dom} \vdash_I \exists y \in \text{Rng.} \Phi(x, y).
\]
This argument hinges on showing

\[ x : \text{Dom} \vdash I ((\exists y \in \text{Rng}. \Phi(x, y) \vee A) \Rightarrow A). \]  

(6.1)

For the case where \( \Phi \) is decidable, we can always prove (6.1). In Chapters 9 and 10, we will discuss why we are restricted to \( \Pi^0_2 \) sentences. Once we have proven (6.1), we have a constructive proof where previously we had only a classical proof. All of this is documented extensively in the literature [Fri78, Lei85, DT89]. The work remaining to effectively use these results falls into two categories:

- Encode logic into the Nuprl type theory, and show that this encoding preserves the translation properties observed above.

- Demonstrate the "extralogical" axioms of Nuprl can be effectively translated.

The next chapter will cover these tasks.
Chapter 7

Justifying the Translations in Nuprl

The task of applying the double-negation and A-translations to Nuprl is two-fold. First, we must decide exactly how to translate each term of the Nuprl type theory. Second, we must decide how to translate each proof step, such that when the translations are pieced together, they result in valid proofs. That is, if we have a classical proof $\langle C \rangle \Phi$ by $M$, we must construct a constructive proof $\langle \rangle \Phi$ by $M'$ where $M'$ is computed from $M$. This chapter addresses these issues. We will show that a particular fragment of Nuprl is amenable to double-negation translation, and we will characterize this fragment in ways which specifically exclude certain pathological cases which are known to be untranslatable. The main problem which arises in translating Nuprl proofs is that Nuprl is not really a higher-order logic. We will find that our restrictions are all aimed at forcing Nuprl into the mold of a higher-order logic. Hence this chapter is mostly technical, and consists of a development of a set of restrictions which are intuitively correct, but must still be verified. This chapter can be skipped by all but those who wish a very technical understanding of our work. The plan of the chapter is as follows:

1. Embed logic into Nuprl.

2. Define Classical Nuprl.

3. Define the double-negation translation on Nuprl terms. This translation puts certain restrictions on the classes of proof that we can translate. We will describe a logic Nuprl$^0$ which is much like Nuprl, except that proofs in Classical Nuprl$^0$ are always double-negation-translatable. The translation on proofs will then be trivial for proofs in Classical Nuprl$^0$, and we will demonstrate that, for the most part, a translation from Nuprl to Nuprl$^0$ can be effected by simple heuristics.

4. Define the double-negation translation on Classical Nuprl$^0$ proofs. We will find that the double-negation translation on classical Nuprl$^0$ proofs always
translates a classical Nuprl\textsuperscript{0} proof to a Nuprl\textsuperscript{0} proof. Moreover, we will see that there is a faithful embedding of Nuprl\textsuperscript{0} into Nuprl via erasure. It will turn out that there are intelligent heuristics for annotating Nuprl proofs to render them into Nuprl\textsuperscript{0} proofs. Thus, instead of proving theorems in Nuprl\textsuperscript{0}, we could just prove them in Nuprl, and afterwards check that they are in Nuprl\textsuperscript{0} via these heuristics.

5. Define the A-translation on Nuprl\textsuperscript{0} terms and proofs.

6. Put the work of the chapter together, and demonstrate how to translate a simple argument from classical Nuprl\textsuperscript{0} into constructive Nuprl\textsuperscript{0}, and, by the embedding, into Nuprl.

Unfortunately, because of the size, complexity, and expressiveness of the Nuprl type theory, we can only provide sufficient conditions for the success of the translation. In this whole discussion, we ignore the set-type, the notational abstraction facility, and the lemma facility. This is only for the purposes of formalization, however, and in practice, we can use all of these facilities to a limited extent. In the last section we will address extending the techniques herein presented to encompass those also.

### 7.1 An Overview of the Translation Results

Nuprl is a constructive type theory, not a constructive logic. That is, it is not a system which fits the standard definition of a logic. In a "standard" logic, we distinguish between syntactic domains of quantification (such as $\text{Int}$), and domains of propositions (such as $\text{Int} \rightarrow U_1$). A universal proposition $\forall x \in T.P(x)$ is not the same thing, ontologically, as a dependent data-type, $D : U_1 \rightarrow D \rightarrow D$ (the type of the parametrically polymorphic identity function). It turns out that upon this difference hinges the entire success of our translation effort. The most difficult problem we will have to solve is to find a fragment of the Nuprl type theory which we can call a "standard" logic. The work of Friedman [Fri78] and Leivant [Lei85] on schematic theories (a general approach to the problem of showing that a logic is translatable) is largely unneeded (in this context), because Nuprl's rules are so easily translatable when we restrict ourselves to the proper fragment.

Let us state this another way. One way of deciding whether a theory is translatable is to verify that, for each rule of the theory, for any instance of that rule, we can translate that instance. The problem here is that we cannot always translate a rule. An example of a rule which we can never translate is the axiom of choice. An example of a rule which we can sometimes translate, but not always, is a rule which decomposes membership goals; a proof of $t \in T$ may involve classically proven assumptions which, under the double-negation translation, are modified to the point where the membership goal is unprovable. In the cases where we cannot translate, there might be a subset of the instances of that rule that we can translate (in the
previous membership example, the classically proven assumptions that we can still use in the membership goal turn out to be the decidable ones). We might then factor that rule into many rules, each for a certain subset of the instances of the original rule. Then, we could show that some of the new rules were translatable, and some not. But this is laborious and unilluminating, because it does not address the central question, which is "what features of Nuprl are, and are not, amenable to translation?"

We will answer this question by describing some conditions on complete Nuprl proof trees, and showing that these conditions allow translation to succeed. In the remainder of this chapter, we develop these conditions one-by-one. Here, though, for the reader who wishes only an overview, we summarize the conditions and the translations.

### 7.1.1 Preconditions for Translation

First, we require that there be a clear division between the notion of "data" and the notion of "proposition". Consider $HA^\omega$: constructive arithmetic, with the addition of the axiom of stratified comprehension, which allows one to define a set at level $n$ via arithmetic formulas, and formulas which refer to sets at level $n - 1$ and lower. There is a clear difference in $HA^\omega$ between a predicate $\Phi$ of two arguments, and the set which collects the graph of the relation expressed by $\Phi$. One is a predicate, and the other is a data-object. Contrast this with Nuprl, where both may be data-objects (and even data-types). The difference is that when we double-negation translate, or A-translate, we wish to only translate propositions (and predicates), and not data-types or data-objects. For instance, we would want to translate the proposition $\forall A \in U_1. A \Rightarrow A$, but we would not want to translate the $\Pi$-type $A : U_1 \rightarrow A \rightarrow A$. These two terms are identical (the former is a syntactic sugaring of the latter), but we would allow ourselves to prove the proposition by recourse to excluded middle, where we would not allow ourselves to do the same for the data-type.

We can make this a bit more precise by first introducing the notion of static typing:

**Informal Definition 7.1.1 (Static Typing)** A term $t$ is statically typed with type $T$ when we can prove, from only type assumptions $H$ which are themselves statically typed that

$$H \gg t \in T$$

Our first condition on Nuprl proofs is

**Informal Condition 7.1.1 (Static Typing)** Every subterm of every term of a Nuprl proof has a type annotation upon it such that, if a term $t$ has type $T$, then, from a hypothesis list $H$ of the type-annotations of the free variables of $t$ (and no
other hypotheses), we can prove $H \gg t \in T$. Of course, $T$ can have free variables from this hypothesis list. Moreover,

- whenever the goal of a sequent is a membership goal $t \in T$, it must be the case that $t$ is actually annotated with $T$, or some type which is "just like" $T$ except that universe numbers may be changed.

- the static typing cannot change from sequent to sequent except via substitution, e.g. the typing of $x$ in $A : U_1 \rightarrow x : A \rightarrow A$ can change from $A$ to $Int$ if we eliminate the type on $Int$.

The first stipulation allows us to prove $Int \in U_2$ even when we annotate Int with $U_1$, for instance. The second stipulation allows us to modify type annotations as we change our hypothesis lists.

This condition should sound eminently reasonable - in our normal discourse, we can automatically identify the type of any expression. In type theory, the type of an expression is often used to carry along more information, such as its adherence to specifications. In the current context, though, such properties, which are undecidable, must be expressed propositionally, and not in the type of an expression, since we want type checking to be decidable.

In addition, one could think of requiring that type-inference be decidable also. This reduces the burden on the user of the system, and we found that in practice we could devise automatic methods for inferring the types of almost every expression in a proof tree.

**Informal Condition 7.1.2 (Kind-ing of Terms)** Every subterm of every term in a Nuprl proof be marked with (a "kind") either a $P$ or a $D$, to distinguish whether it is a proposition or a data-type/data-object, such that every term which is marked with a $P$ is type-annotated with a universe-type.

Given a kindable proposition, we can perform a translation in which each subterm occurrence which is marked with a $P$ is replaced with its double-negation, and the resulting term is well-formed and kindable.

Another condition that is clear in $HA^\omega$ is that whenever we can prove $A \equiv B$, where $A, B$ are propositions, we can exchange them freely in any proposition. Likewise, we can exchange the two predicates

$$\lambda x.x = 1 \in Int \text{ and } \lambda x.x - 1 = 0 \in Int$$

in any proposition. If we extend this idea up the type hierarchy, we get a notion of predicate equivalence which tells us that, when a predicate $P$ is presented with two predicate-equivalent arguments, $x$ and $y$, $P(x)$ is predicate-equivalent to $P(y)$. We require that

**Informal Condition 7.1.3 (Predicate Substitutivity)** All propositions and predicates are predicative-equivalent under substitution of predicate-equivalent subterms. That is, if $A \leftrightarrow B$ then $\Phi[x/A] \leftrightarrow \Phi[x/B]$ (where $x$ has the same type and "kind" as $A$ and $B$.}
Again, this condition is eminently reasonable, and we would expect this to be true of any logic.

What these conditions really do is limit the expressiveness of the logic in such a way that we end up with a logic resembling arithmetic, or higher-order logic (e.g. \( HA^\omega \)). But these conditions are more general than \( HA^\omega \), since we could imagine a restricted Nuprl which was also impredicative (which \( HA^\omega \) is not).

### 7.1.2 Translation and Conservative Extension

With these conditions in hand, we can now define double-negation translation. We identify which hypotheses are propositions, and for each proposition, we can identify which subterm positions are propositions. We simply double-negate each propositional subterm position of each propositional hypothesis, and also of each conclusion (which is guaranteed to be a proposition). It is easy to show

**Desired Theorem 7.1.1 (Double-Negation Translation)** Given a classical proof of a proposition \( \Phi \) which meets informal conditions 7.1.1, 7.1.2, and 7.1.3, we can effectively generate a constructive proof of the double-negation translation of \( \Phi \).

**Proof Sketch:** We show that, for every rule of Nuprl, for every instance of that rule in a classical Nuprl proof tree which meets the above conditions:

\[
H \gg T^P
\]

by \( r \)

\[
H_1 \gg T_1^P
\]

\[
H_2 \gg T_2^P
\]

\[
\vdots
\]

\[
H_n \gg T_n^P
\]

we can effectively construct a combination of rules of constructive Nuprl which reduces the double-negation translation of the root sequent above to the double-negation translations of a subset of the subsidiary sequents above. 

Of course, this proof sketch is tremendously unsatisfactory, because we have not demonstrated that the extralogical axioms of Nuprl are translatable. To do this, we must develop the machinery to formalize conditions 7.1.1, 7.1.2, and 7.1.3. The rest of this chapter is devoted to this task.

A theorem much like 7.1.1 holds for A-translation also, since, for any classical Nuprl proof tree which satisfies the above conditions, the double-negation translated proof tree also satisfies the above conditions. So we can show

**Desired Theorem 7.1.2 (A-Translation)** Given a Nuprl proof of \( \Phi \) that satisfies conditions 7.1.1, 7.1.2, and 7.1.3, we can effectively generate a Nuprl proof tree of the A-translation of \( \Phi \).
don’t want the inhabitants of this type to be assertions about the non-existence of conterexamples to the inhabitation of some subset of the integers; we want functions from positive integers to positive integers to a subset of the positive integers. So what we would really like is two different kinds of $\Pi$-type, one for universally-quantified propositions, and one for dependent function types. Then, we would want to guarantee that we never inhabit a dependent function type via classical argument. Likewise, we would want to guarantee that we never use the “inhabitant” of a universally-quantified proposition except to prove other propositions.

The distinction we are making here is the distinction between propositions and types (to avoid confusion, we will use the term data-types instead). In the history of constructivism, the Curry-Howard isomorphism stands out as a unifying concept in the understanding of proofs and computations, since it tells us that propositions may be identified with data-types. However, this is only for a truly constructive logic. For a classical logic, it should be clear that indeed data-types are different objects from propositions, since when we interpret propositions constructively, we must double-negate them, but we need not do so for data-types. Thus, we should have a way to make and maintain that distinction. One solution is to use a different logic, one that supports the distinction between propositions and data-types. However, it suffices to annotate terms in our proofs with information telling us whether or not the term is a proposition, a data-type, or a non-type. That is, if we had an annotation of every term in a proof, telling us whether that term was a type, and what sort of type it was, then we could verify that when we said something like “$m : N^+ \rightarrow N^+ \rightarrow \{1, \ldots, m\}$ is inhabited”, we meant that it was inhabited by a function. Let us look again at the axiom of choice, and see where this idea can help us. The axiom of choice is proven in Nuprl thus:

\[
A : U_1, B : U_1, \Phi : (A \times B) \rightarrow U_1 >> \\
\forall x \in A. \exists y \in B. \Phi(x, y) \Rightarrow \exists f \in A \rightarrow B. \forall x \in A. \Phi(x, f(x)) \\
BY \ (\text{ITerm '}\lambda \ h. \lambda \ x. (h(x)).1') \ldots
\]

We have a proof of the proposition $\forall x \in A. \exists y \in B. \Phi(x, y)$, and we would like a proof of the proposition $\exists f \in A \rightarrow B. \forall x \in A. \Phi(x, f(x))$. We get that proof by providing an inhabitant of the data-type $A \rightarrow B$, such that certain properties hold. But what inhabitant do we provide? Looking at the proof, we see that we give a name to the proposition $\forall x \in A. \exists y \in B. \Phi(x, y)$, $h$, and then use that name as a function, to construct the data-type inhabitant. This involves a conversion of a proposition into a data-type, and just does not make sense in a classical theory. If we had two different $\Pi$-types, one for universal quantification and one for dependent function space, we could not prove the axiom of choice in its above form. We solved this problem by annotating each $\Pi$-type so we could statically recognize which $\Pi$-types were supposed to denote data, and which propositions. Then, we could guarantee that no $\Pi$-types that represented data were proven via classical reasoning, and that no $\Pi$-types that represented propositions were ever used as functions.
Proof Sketch: Again, by structural induction over the proof tree. □

Finally, we can put these results together and, using Friedman’s argument from chapter 6, show

Desired Theorem 7.1.3 (Conservative Extension for Restricted Nuprl)
Given a classical Nuprl proof of \( \Phi \) which meets the conditions above, we can effectively generate a constructive Nuprl proof of \( \Phi \).

7.2 Classical Nuprl

Let us denote by \( \gg \) provability in the standard Nuprl type theory, and by \( \triangleright \text{C} \triangleright \) provability in the Nuprl type theory with the addition of the following rule:

\[
\Gamma \triangleright \text{C} \triangleright \Phi \lor \neg \Phi
\]

BY excluded middle at \( i \)

\( \triangleright \text{C} \triangleright \Phi \in U_i \)

The idea is that we may apply excluded middle to any well-formed formula of Nuprl. We will use the classical turnstile \( \triangleright \text{C} \triangleright \) to denote a classically proven sequent. Then we would like to prove the following:

Non-Theorem 7.2.1 (Nuprl Double-Negation Translation) Given a proof, \( \triangleright \text{C} \triangleright \Phi \), we can effectively (and automatically) construct a proof \( \gg \neg \neg \Phi \).

This isn’t true (the axiom of choice, among other axioms, isn’t translatable). We would like a characterization of what rules of Nuprl, of what subtheory of Nuprl, is translatable automatically. We will provide sufficient conditions for the success of the translation. These conditions will at first glance appear tremendously costly, but in reality, they correspond to a very rigid, well-behaved, encoding of logic in Nuprl. First, though, we must discuss ontology.

7.3 Propositions vs. Types

In constructive type theory, a proof of \( x : A \rightarrow B \) is a function which, given a value \( a \in A \), returns a value in \( B[a/x] \). In a classical type theory, though, the real meaning of a \( \Pi \)-type is shrouded in uncertainty, since a proof of the \( \Pi \)-type is really a witness for a lack of counterexamples for the truth of the \( \Pi \)-type. Let us say that again. If I prove \( f \in \forall x \in A. \Phi(x) \) constructively, I’m saying that I can, for any \( a \in A \), produce a value \( f(a) \in B[a/x] \). Given a classical proof, though, all I can really say is that, for any \( a \in A \), there can be no counter-examples to the inhabitation of \( B[a/x] \). And that is a much weaker statement. However, for certain types, there just isn’t such a thing as a “classical” proof. Consider the type \( m : \mathbb{N}^+ \rightarrow \mathbb{N}^+ \rightarrow \{1, \ldots, m\} \). If we are trying to use type theory as a logic, we
H >> T^P
by seq T_1^P,\ldots,T_n^P
>> T_1^P
T_1^P >> T_2^P
\vdots
T_1^P,\ldots,T_n^P >> T^P

Let us give some examples:

- (\forall \Phi^P \in Prop_1.\neg(\neg\Phi^P) \Rightarrow \Phi^P)^P is a proposition, since the first rule lets us universally quantify over the universe Prop_1 whose inhabitants are propositions. However, since we cannot universally quantify over a proposition, we are prevented from quantifying over a classically proven universal type.

- (\forall x^D \in Int^D.(x^D = 1 \in Int)^P)^P is also universally quantified statement, but this time, we are quantifying over a data-type, with data-inhabitants.

The seq rule is another example of this distinction. It does not make sense to inhabit a data-type via proof. That is, it does not make sense to prove \texttt{\texttt{\#}}\ Int. Thus, all the types "cut in" by the seq-rule should be propositions. Likewise, in the rule for universal-elimination, we see that the "kind" of the codomain of the type being eliminated must match the "kind" of the term supplied. This prevents us from converting a proposition into a data-type.

### 7.4.1 Membership and Nuprl° Proofs

We turn now to the problem of membership goals, which are the most common sort of goals which come up in Nuprl proofs. They are all of the form >> t \in T. By far the most common sort of membership goal is the well-formedness goal, where T is a universe. In some cases, this goal is proven in a fashion that cannot be double-negation translated. For instance, if in the proof of t \in T, we used undecidable propositional assumptions, we would be unable to push thru the double-negated goal. The basic error here is that a propositional assumption is being used to prove a membership goal. Membership goals are the "baggage" that we must carry with us to encode logic in Nuprl, and as such should not be amenable to proof via double-negation. That is, membership goals are part of the substrate which makes logical discourse possible, and not part of the discourse itself. Thus, we require that

**Condition 7.4.1 (Static Membership)** No membership goal in a Nuprl° proof may be proven with recourse to propositional assumptions.

This will later allow us to show that whenever we have a classical Nuprl° proof H \texttt{\texttt{\#}} t \in T, we can effectively translate it into a constructive Nuprl° proof of the same goal.
7.4 Nuprl\(^o\) Defined

The system Nuprl\(^o\) is just like Nuprl, except that we make a distinction between *data types* and *propositional types*. This distinction is accomplished by annotating every variable, every term, and every subterm of every term, with a marking denoting whether it is a data type or data object or a proposition. We list below some of the rules for Nuprl\(^o\), specifically those for universal quantification formation, and the seq rule. The remainder are in appendix B.

**Universal proposition formation. Domain must be data.**

\[ H \gg (x^D : A^D \to B^P)^P \text{ in Prop}_i \text{ by intro [new } y \text{]} \]
\[ \gg A^D \text{ in Data}; \]
\[ y^D : A^D \gg B[y^D/x]^P \text{ in Prop}_i \]

**Universal proposition introduction - domain Data**

\[ H \gg (x^D : A^D \to B^P)^P \text{ by intro at } U_i \text{ [new } y \text{]} \]
\[ \gg A^D \in \text{Data}; \]
\[ y^D : A^D \gg B[y^D/x]^P \]

**Universal elim - domain Data**

\[ H, (x^D : A^D \to B^P)^P, H' \gg T^P \text{ by elim } f \text{ on } a \]
\[ \gg a^D \text{ in } A^D \]
\[ B^P[a^D/x] \gg T^P \]

**Universal proposition formation. Domain must be propositions.**

\[ H \gg (x^P : \text{Prop}_j \to B^P)^P \text{ in Prop}_i \text{ by intro [new } y \text{]} \]
\[ \gg \text{Prop}_j \text{ in } U_i \]
\[ y^P : \text{Prop}_j \gg B[y^P/x]^P \text{ in Prop}_i \]

**Universal proposition introduction - domain Prop**

\[ H \gg (x^P : \text{Prop}_i \to B^P)^P \text{ by intro at } U_j \text{ [new } y \text{]} \]
\[ \gg \text{Prop}_i \in U_j \]
\[ y^P : \text{Prop}_i \gg B[y^P/x]^P \]

**Universal elim - codomain Prop.**

\[ H, (x^P : \text{Prop}_i \to B^P)^P, H' \gg T^P \text{ by elim } f \text{ on } a \]
\[ \gg a^P \text{ in } \text{Prop}_i \]
\[ B^P[a^P/x] \gg T^P \]

**Seq rule**
7.4 Nuprl Defined

Often in double-negated proofs we will end up needing to prove the well-formedness of terms over and over again. For instance, the proof step for and-introduction,

\[ \Gamma \gg A \land B^P \]

\[ \text{BY and-intro at } i \]

\[ \gg A \]

\[ \gg B \]

becomes, under double-negation translation,

\[ \Gamma \gg \neg(\neg(A \land B^P)) \]

\[ \text{BY function intro at } i \]

\[ \neg(\neg(A \land B^P)) \gg \text{void} \]

\[ \text{BY function elim on last hyp} \]

\[ \gg \neg(A \land B^P) \]

\[ \text{BY and-intro at } i \]

\[ * \gg \neg(A) \]

\[ * \gg \neg(B) \]

\[ \gg \neg(\neg(A \land B^P)) \in Prop_i. \]

The well-formedness goal \( \gg \neg(\neg(A \land B^P)) \in Prop_i \) should be automatically provable, and it is, if we have a well-formedness proof for the original term \( A \land B^P \). So what we will require is:

**Condition 7.4.2 (Well-formedness of Propositions)** Every proposition \( \Phi \) in every sequent (both hypotheses and conclusions) comes with a proof of well-formedness, such that:

- for every sequent in the well-formedness proof of the form \( H \gg t^P \in T, T \) is \( Prop_i \) for some \( i \).
- every subterm occurrence of \( \Phi \) has a type which is defined in terms of the free variables at that occurrence.
- the well-formedness proof is done without recourse to any propositional hypotheses.

### 7.4.2 Proposition-hood

One final problem we encounter is with the basic notion of proposition-hood in Nuprl. To be more exact, we will need some more definitions:

**Definition 7.4.1 (Term Context)** A term context is a term with a “hole” in it. For example, the term \( \neg\neg(\bullet) \) is a context, and we can fill this context with \( Q \) to get the term \( \neg\neg(Q) \). With every hole is associated a kind, e.g. \( (D \text{ or } P) \), and only terms of the proper kind may be put into the hole.
Definition 7.4.2 (Subterm Occurrence) Given a term, $t$, a subterm occurrence of $s$ in $t$ is written as $t = C[s]$, meaning that the subterm $s$ occurs in $t$ in the context $C[\bullet]$.

Consider the formula $\forall P \in Prop_1 \rightarrow Prop_1.P(\Psi)$ where $\Psi$ is some undecidable proposition.

We have left out the annotations, but it's clear what they are. Let $C[\bullet] = \forall P \in Prop_1 \rightarrow Prop_1.P(\bullet)$. Clearly, we should be able to prove that if $\Phi \Leftrightarrow \Psi$, then $C[\Phi] \Leftrightarrow C[\Psi]$. That is, the term $\Psi$ is used as a proposition, and not as a type. However, this isn't provable for the context $C[\bullet]$ above ($P$ need not respect bi-implication in its argument). At its core, the problem is one of making type constructors which are being used as proposition constructors respect double-implication. We will do this by placing restrictions on the method of construction of propositions and predicates.

The basic problem the previous example points out is that the type constructors don't always respect the substitution of equivalent (but not equal) propositions. In particular, the true culprit is universal (and existential) quantification. Universal quantification over, for instance, $Prop_1 \rightarrow Prop_1$ makes it impossible to guarantee that, in the example above, $P$ respects $\Leftrightarrow$ relations on its argument. One way to fix this problem is to disallow such "higher-order predicates". For the formalization of Higman's Lemma, this was sufficient. However, for a more general development of mathematics, this is clearly insufficient, as higher-order predicates are necessary to express, for instance, classically defined functions that take other functions as arguments (a good example might be infinite sequences of infinite graphs, and operations on these). We will show that the familiar constructs which we use to express ourselves in discourse are all well-behaved (in a sense which we will make precise) under the substitution of equivalent propositions. We will also show that for higher-order predicates, a certain simple, easy-to-express translation renders them also well-behaved. Let us begin by defining what we mean by the "familiar constructs" used to express propositions and predicates.

Condition 7.4.3 (Well-Constructedness) A proposition is well-constructed when it is consistently kinded, and no variable is a higher-order predicate.

This condition rules out universal quantification over any predicates which take other predicates as arguments. An example of an ill-constructed proposition is $\forall P \in Prop_1 \rightarrow Prop_1.P(\Psi)$. Soon we will prove that this definition is sufficient to guarantee our propositions have the properties we wish them to have. First, though, some more definitions:

Definition 7.4.3 (Predicate Equivalence) Two expressions $a, b$, members of a type $T$, are predicate-equivalent (written $a \Leftrightarrow b \in T$) when:

Case 1 — $T$ is not a type of predicates: $T$ is a concrete data type, or a type of functions between concrete data types or tuples of concrete data types. As such, $a \Leftrightarrow b \in T$ is just $a = b \in T$. 

Case 2 — $T = \text{Prop}_i$ ($\text{Prop}_i$ is a type of 0-ary predicates):

\[ a \iff b \wedge a \in \text{Prop}_i \wedge b \in \text{Prop}_i \]

Case 3 — $T$ is a simple (i.e. not dependent) predicate type, $T = T_1 \rightarrow T_2$:
If $T_1$ is a simple type, that is, not a predicate or propositional type itself, then

\[ \forall x \in T_1. (a(x) \iff b(x) \in T_2) \]

else ($T_1$ is a predicate or propositional type)

\[ \forall P, Q \in T_1. P \iff Q \in T_1 \Rightarrow a(P) \iff a(Q) \in T_2 \]
\[ \wedge \forall P, Q \in T_1. P \iff Q \in T_1 \Rightarrow b(P) \iff b(Q) \in T_2 \]
\[ \wedge \forall P, Q \in T_1. P \iff Q \in T_1 \Rightarrow a(P) \iff b(Q) \in T_2. \]

Case 4 — $T$ is a dependent type of predicates: Say $T = z : A \rightarrow B$, $x$ not free in $B$. Then $A$ must be a simple type, since no data-type can have a proposition as a subterm (this will be proven later) and $B$ is a data-type. Then $A \iff B \in T$ iff

\[ \forall x \in T_1. (a(x) \iff b(x) \in T_2[x/z]). \]

We will sometimes need to talk about the reflexive equivalence relation:

**Definition 7.4.4 (Reflexive Predicate Equivalence)**

$A$ is a predicate in a type $T$ (written $A \in_{\equiv} T$) when $A \iff A \in T$.

Finally, we will need to codify the notion of a well-behaved predicate:

**Definition 7.4.5 (Well-Behaved Predicate)**

A well-behaved predicate $P$ is one for which, under the assumption that, for every one of its free variables $x : X$, $x \in_{\equiv} X$, we can show that for every subterm occurrence $P = C[r]$, where $r \in R$ if $P$ is safe for substitution (that is, none of the free variables of $P$ are bound in $C[\bullet]$), that the following is provable:

\[ \forall u, v \in R. [(u \iff v \in R) \Rightarrow (C[u] \iff C[v] \in T)]. \]

That is, any free subterm of $P$ can be replaced with an $\iff$-equivalent term, and the resulting term is $\iff$-equivalent to $P$.

Let us show:

**Lemma 7.4.1.** For every well-constructed predicate $P \in T$, if for every free variable $x \in X$, $x \in_{\equiv} X$, then

- $P \in_{\equiv} T$.
- $P$ is a well-behaved predicate.
Proof: By induction on the construction of predicates: Show that if for all free variables $x : X$, $x \in \Theta X$, and $P \in T$, and for subterm occurrences $P = C[s]$, where $s \in S$, $s \in \Theta S$ and $s$ is well-behaved, then $P \in \Theta T$ and $P$ is well-behaved.

Case 1 — $P$ a variable: Trivial. There are no subterms safe for substitution except $P$ itself, and variables are automatically well-behaved.

Case 2 — $P$ not a predicate: If $P$ is not a predicate, then $P \in \Theta T$ is just $P \in T$, which is given by hypothesis. Moreover, since $P$ is not a predicate, it can contain no propositional subterms, which means that for any subterm of $P = C[r]$, $r \in R$ which is safe for substitution, $u \leftrightarrow v \in R$ is just $u = v \in R$, so clearly $P$ well-behaved is just $\forall u, v \in R. u = v \in R \Rightarrow C[u] = C[v] \in T$, which follows from type-functionality and the fact that all terms are statically typable.

Case 3 — $P = u(v)$ is an application, $u \in (x : A \to B)$, $x$ free in $B$:
Show: $u(v) \in \Theta T$: Since $P \in T$, $v \in A$ and $T = B[v/x]$. By assumption, $u \in \Theta (x : A \to B)$, and $v \in \Theta A$. Immediately, we get $u(v) \in \Theta T$, since
\[ \forall z \in A. u(x) \in \Theta B[z/x]. \]

Show: $u(v)$ is well-behaved: Consider a subterm occurrence $P = C[r]$, $r \in R$. If $r = P$, $(C[\bullet])$ is the null context, $r \in T$), then clearly the second proof obligation is met. Otherwise, $r$ is a subterm of either $u$ or $v$. In either case, the substitution there respects $\leftrightarrow$, and from this, and the fact that application respects $\leftrightarrow$, we can show that the result of the substitution respects $\leftrightarrow$.

Case 4 — $P = \lambda x. b$, $T = y : A \to B$:
Show: $\lambda x. b \in \Theta T$: $\forall u, v \in A. u \leftrightarrow v \in A \Rightarrow (\lambda x. b)(u) \leftrightarrow (\lambda x. b)(v) \in T$. This is trivial, since $(\lambda x. b)(u) = b[u/x]$, and $(\lambda x. b)(v) = b[v/x]$, and we can show the required result by using the inductive assumption that for all subterms of $b$ which are free for substitution, the substitution respects $\leftrightarrow$.

Show: $\lambda x. b$ well-behaved: Pick a subterm of $\lambda x. b$ which is free for substitution. If it’s the whole term, then we are done. If it’s not the whole term, then it’s a subterm of $b$, say, $b = C[r]$, $r \in R$. Then, under the assumption that $x \in \Theta A$, we have $\forall u \in v.Ru \leftrightarrow v \in R \Rightarrow C[u] \leftrightarrow C[v] \in B[x/y]$. From this, the goal $\forall u \in v.Ru \leftrightarrow v \in R \Rightarrow (\lambda x. C[u]) \leftrightarrow (\lambda x. C[u]) \in y : A \to B$ follows easily, since $r$ does contain $x$ free.

Case 5 — $P$ is an inductively defined predicate: In this case, we need to use the fact that we can decide whether the principal argument of the inductive definition is equal to, less than, or greater than, zero. This allows us to handle each case separately. We proceed by unrolling the definition one step, and then substituting the proper predicate in. We have to use induction to do all of this, but it does indeed work, and we get the required result.

Case 6 — $P = \forall x \in A.B$: In this case, $T = \text{Prop}_i$, and the first obligation is easy, since $(\forall x \in A.B) \in \Theta \text{Prop}_i$ is just $(\forall x \in A.B) \in \text{Prop}_i$. To fulfill the second obligation, we must assume that $A$ is not a type of higher-order predicates, since under the assumption that $x \in \Theta A$, we have that $B$ is well-behaved. This makes $x \in \Theta A$ just $x \in A$, and we can then prove that $P$ is well-behaved.
The other cases are much the same. Note that we have purposely left out the case of universal quantification over a type of higher-order predicates. We will handle this presently.

Finally, the theorem we have been waiting for:

**Theorem 7.4.1 (Well-Constructed ⇒ Well-Behaved)** Any well-constructed proposition $\Phi$ in a Nuprl proof is well-behaved.

**Proof:** This theorem follows trivially from the previous lemma. We assumed that there was no quantification over types of higher-order predicates, and this means that there are no variables $v \in V$ in the sequent for which we cannot show $v \in_{\leftrightarrow} V$. Thus the conclusion to the lemma follows, and $\Phi$ is well-behaved.

This theorem tells us that we can do rewriting on predicates (substituting formulas related by bi-implication) with impunity, and always construct a proof justifying the rewrite.

Before we get to the definition of Nuprl° proofs, let us discuss ways of extending this idea to universal quantification over higher-order predicates. In the example of $\forall P \in Prop_{1} \rightarrow Prop_{1}.P(\Phi)$, the problem was that we couldn’t know $P \in_{\leftrightarrow} Prop_{1} \rightarrow Prop_{1}$. But if we added this assumption to the universal quantification thus:

$$\forall P \in Prop_{1} \rightarrow Prop_{1}.P \in_{\leftrightarrow} Prop_{1} \rightarrow Prop_{1} \Rightarrow P(\Phi),$$

we could show $P \in_{\leftrightarrow} Prop_{1} \rightarrow Prop_{1}$. So we must translate all universal quantifications over higher-order predicates. Unfortunately, we must make this translation on terms in parallel with double-negation translation, so we will not address this in our discussion of the translations, for the sake of clarity.

### 7.4.3 Nuprl° At Last

We can now define what is meant by a *valid* proof in Nuprl°.

**Definition 7.4.6.** A *valid Nuprl° proof* is a proof tree composed of the rules of Nuprl° which meets the three conditions stated above; that is, all membership goals are static, every proposition comes with a static well-formedness proof, and every proposition is well-constructed (which means that it is well-behaved).

The system Nuprl° contains the type constructors Int, Less, Void, List, $\Pi$, $\Sigma$, Union, and Equality. When augmented by the rule of excluded middle as modified below,

$$\Gamma \triangleright\triangleright \Phi^{P} \lor \neg\Phi^{P}$$

**BY magic at $i$**

$$\triangleright\triangleright \Phi^{P} \in U_{1}^{P}$$
we get a system, Classical Nuprl\(^o\), such that for any proof of a statement \(\Phi\) in Classical Nuprl\(^o\) there exists a proof of the analogous statement, \(\overline{\Phi}\) in constructive Nuprl\(^o\), and via erasing the annotations, in Nuprl. We will arrive at a proof of this statement shortly. First, though, some technical results, and the translations on terms. In the rest of this chapter, whenever annotations are left off, they are to be inferred from context.

### 7.4.4 Technical Results Concerning Nuprl\(^o\)

Here we set forth some technical results that we will need to show that Nuprl\(^o\) is really subtheory of Nuprl, and that it is indeed double-negation translatable. We set them forth without proof, as they should be obvious. All these facts are about complete, consistent (perhaps classical) Nuprl\(^o\) proof trees.

**Fact 7.4.1.** The first technical result, and most important, is that we can map Nuprl\(^o\) back to Nuprl by simply erasing annotations.

**Fact 7.4.2.** No hypothesis whose witness is available by name is ever a proposition.

**Fact 7.4.3.** Every hypothesis which is a data-type is available by name.

**Fact 7.4.4.** No data type is ever inhabited by direct proof. That is, no sequent is of the form \(H \gg T^D\).

**Fact 7.4.5.** No proposition is ever the equality-type of an I-type (i.e. in \(A = B \in T\), \(T\) is a data-type, and not a proposition). This means that we can never reason explicitly about the inhabitants of a proposition.

Looking at the rules for function types, we see right away that if we have a universal quantification, we can eliminate it only on the "right" value. That is, we cannot eliminate \(\forall A^D \in B^D.\Phi(A^D)^{P^P}\) on the term \(P^P\). We can put this another way by saying

**Fact 7.4.6.** Universal elimination preserves the notion of annotation.

**Fact 7.4.7.** Reduction in Nuprl\(^o\) preserves the notion of annotation. That is, no variable \(x^D\) is ever replaced by \(\Phi^P\), and no variable \(x^P\) by a data value \(T^D\) during reduction.

**Fact 7.4.8.** No subterm of a data-type is a proposition.
7.5 Double-Negation Translation on Terms, Sequents, and Proofs

The double-negation translation for terms of Nuprl\(^\circ\) is simple. Given a term \(t\), and a type \(T\) such that \(t \in T\), we double-negate those subterms of \(t\) which are annotated \((-)^P\). So, for instance, the term \(\forall x^D \in Int^D.x^D = 1 \in Int^{PP}\) is translated as (with annotations omitted):

\[
\neg\neg(\forall x \in \text{Int}.\neg\neg(x = 1 \in \text{Int}))
\]

and the term \(\forall Q^P \in \text{Prop},(Q \lor \neg(Q))^P\) is translated as

\[
\neg\neg(\forall Q \in \text{Prop},\neg\neg(\neg\neg(Q) \lor \neg\neg(\neg\neg(Q) \Rightarrow \neg\neg(\text{void})))).
\]

Let us denote the double-negation translation by \(\hat{\bullet}\), since it's really just an extension of the Kolmogorov translation already discussed. The translation on sequents is very simple. Let us write a sequent as:

\[
x_1 : T_1, \ldots, x_n : T_n \gg \Phi.
\]

Some of the hypotheses are propositions, in which case the symbol \(x_i\) for the \(i\)-th hypothesis would be null. We translate this sequent as follows:

- The goal is double-negation translated as described above.
- Any hypothesis which is data, that is, a hypothesis of the form \(x^D : T^D\), is translated as-is.
- Any hypothesis which is a proposition is translated via the translation on terms.

We will need some notation for describing a double-negated sequent. If the original sequent is \(H \gg T\), then we will denote the double-negated sequent by \(\overline{H} \gg \overline{T}\).

7.5.1 Some Results about Double-Negation Translation

We will need to prove some technical results about double-negation translation, and define some rule combinations which perform specific tasks.

We begin with some facts about hereditarily decidable propositions.

**Fact 7.5.1.** If a proposition \(\Phi\) and all of its subterms are decidable, then \(\Phi \leftrightarrow \overline{\Phi}\).

That is, we can "erase" the double-negations from \(\Phi\) if \(\Phi\) is hereditarily decidable.

**Lemma 7.5.1 (Double-Negated Membership Goals)**

Given a goal \(H \gg t \in T\) in a Classical Nuprl\(^\circ\) proof, the goal \(\overline{H} \gg \overline{t} \in T\) is provable in Nuprl\(^\circ\).
Proof: The proof is by cases: Case 1 — $T = \text{Data}_i$: $\overline{H} \gg \overline{\mathfrak{t}} \in T$ is provable in Nuprl°. This is obvious; the hypotheses used in the original proof don’t change, since they’re all data-types, and data-types don’t contain propositions as subterms. Moreover, $\overline{\mathfrak{t}} = t$ since $t$ is a data-type, so it does not contain propositions either.

Case 2 — $T = \text{Prop}_j$: $\overline{H} \gg \overline{\mathfrak{t}} \in T$ is provable, since the relevant hypotheses don’t change, and every time a term $s^P$ which is a subterm of $t$ appeared in the original proof as a membership goal $H \gg s^P \in \text{Prop}_j$, it appears in the new proof as $\overline{H} \gg \overline{s^P} \in \overline{\text{Prop}_j}$, and we can strip the outermost double-negation before applying the same rule that was applied in the old proof to justify the sequent $H \gg s^P \in \overline{\text{Prop}_j}$.

Case 3 — $T \neq \text{Prop}_j, T \neq \text{Data}_i$: $T$ is a data-type, and so it isn’t changed by translation. Neither are any relevant hypotheses. We can apply the same method as in the previous case, and just “pop off” the double-negations when they appear.

Combining this fact with the requirement that, for every proposition in every sequent we have a proof of well-formedness, we can say:

Lemma 7.5.2 (Double-Negated Well-Formedness) For every sequent in a classical Nuprl° proof, for every proposition $\Phi$ in the double-negated version of that sequent, we can automatically prove the well-formedness of $\overline{\Phi}$.

Proof: Trivial, since we have available well-formedness proofs for all propositions of the classical proof.

Lemma 7.5.3 (Double-Negation Compaction) Given a proposition $\Phi^P$ in a classical proof, and its double-negated version, $\overline{\Phi^P}$, for any propositional subterm occurrence $\overline{\Phi^P} = C[\neg\neg(s^P)]$, we can prove $C[\neg\neg(\neg\neg(s^P))] \iff C[\neg\neg(s^P)]$.

Proof: Trivial consequence of the fact that all propositions in classical proofs are well-behaved, and double-negation preserves well-behavedness.

This lemma allows us to rewrite any term $t = C[\neg\neg(\neg\neg(s^P))]$ to a term $C[\neg\neg(s^P)]$. Using this lemma, we can put any double-negated term in its most compact form, that is, a form in which there are no quadruple negations. Note that without the stipulation on the well-behavedness of propositions, we wouldn’t be able to prove this lemma.

Another concept we will need to use again and again is the idea of double-negation translating a term under substitution.

Lemma 7.5.4 (Double-Negating Substitution) Suppose that $x$ is free in $B^P$. Then the term $B[t/x]$ is the result of substituting the term $t$ in for all free occurrences of $x$. Either

- $x$ is not a proposition, and $\overline{B[t/x]} = \overline{B[t/x]}$.
- $x$ is a proposition, $\overline{\mathfrak{t}} = \neg\neg(s)$, and $\overline{B[t/x]} = \overline{B[s/x]}$. 

Proof: A proof of this fact is simple. Case 1 — $x$ is not a proposition: Consider any occurrence of $x$, e.g. $C[x] = B$. Since $x$ is not a proposition, we will not double-negate $x$, nor would we double-negate any term that we substituted for $x$. We might double-negate subterms of the term $t$, but we wouldn’t actually double-negate $t$. So applying the double-negation translation to $t$ before substituting into $B$ is equivalent to applying it after substituting.

Case 2 — $x$ is a proposition: Consider an occurrence of $x$, e.g. $C[x] = B$. Since $x$ is a proposition, in $B$, that occurrence will be double-negated, and so would any term $t$ we replaced that occurrence of $x$ with. Let $\overline{B} = C'[x]$ be the analogous occurrence of $x$ in $\overline{B}$. Then the immediately surrounding context of that occurrence is a double-negation, and we would expect that any proposition we substituted for $x$ was not outermost double-negated, since the context will supply that double-negation. Thus, the term to substitute is not $\overline{t}$, but $s$, where $\overline{t} = \neg\neg(s)$. Let us look at an example: if $B = Q^p \land x^p$, and $t = R^p$, then $B[t/x] = Q \land R$, $\overline{B} = \neg\neg(\neg\neg(Q) \land \neg\neg(x))$, $\overline{t} = \neg\neg(R)$, and $\overline{B}[\overline{t}/x] = \neg\neg(\neg\neg(Q) \land \neg\neg(\neg\neg(R)))$. Clearly, this is not right. We would like $\overline{B}[t/x] = \neg\neg(\neg\neg(Q) \land \neg\neg(R))$. To get this, we let $\overline{B}[t/x] = \overline{B}[s/x]$, where $s$ is $\overline{t}$ after stripping off its outer double-negation.

We will now introduce proof procedures which perform certain transformations upon sequents. The general scheme is to exhibit combinations of rules which, given a sequent, will reduce that sequent to another sequent which is suitably modified. These proof procedures are simply descriptions of the transformation tactics which we implemented in order to effect the actual proof translations in Nuprl. We describe our implementation experiences in Chapter 8.

There are many such transformations that are useful in the translation result to be proven in the next section, and we will give each translation a mnemonic name.

Tactic 7.5.1 (UncoverDNHyp) This procedure reduces a goal $\overline{H}, \neg\neg(Q), \overline{H'} \triangleright \triangleright \neg\neg(T)$ to the goal $\overline{H}, Q, \overline{H'} \triangleright \triangleright \neg\neg(T)$. That is, it “uncovers” the hypothesis $P$. It is defined as follows:
$\overline{H}, \neg \neg (Q), \overline{H'} \gg \neg \neg (T^P)$
by intro at $U_i$
$\neg (T^P) \gg \text{void}$
by elim $\neg \neg (Q)$
$\text{void} \gg \text{void by hypothesis}$
$\gg \neg (Q)$
by intro at $U_j$
$Q \gg \text{void}$
by Assert $\neg \neg (T^P)$
$\star$
$\gg \neg \neg (T^P)$
$\neg \neg (T^P) \gg \text{void}$
by elim $\neg \neg (T^P)$ THEN Hypothesis
$\gg \neg (T^P) \in \text{Prop}_i$
$\gg \neg (T^P) \in \text{Prop}_i$.

The well-formedness goals $\gg \neg (T^P) \in \text{Prop}_i$ and $\gg \neg (T^P) \in \text{Prop}_i$ are both provable automatically, as shown before. So, what's left over is the $\star$-ed goal $Q \gg \neg \neg (T^P)$. So this sequence of steps "uncovers" the $Q$.

In the same way, we can define other sequences of steps:

**Tactic 7.5.2 (UncoverDNConcl)** Uncovers a double-negated conclusion. The proof steps are as follows:

$H \gg \neg \neg (\Phi)$
by intro at $U_i$
$\neg (\Phi) \gg \text{void}$
by elim on $\neg (\Phi)$ then thin
$\star \gg \Phi$
$\text{void} \gg \text{void by hypothesis}$
$\gg \neg (\Phi) \in \text{Prop}_i$

The only subgoal left over is $H \gg \Phi$, and this sequent is identical to the top sequent, except that the conclusion has had its outermost double-negation stripped away (it has been "uncovered"). The well-formedness goal $\gg \neg (\Phi) \in \text{Prop}_i$ is automatically proven from static well-formedness information.

**Tactic 7.5.3 (EraseDNHyp)** Erases the double-negations of a hereditarily decidable hypothesis.

**Tactic 7.5.4 (EraseDNConcl)** Erases the double-negations of a hereditarily decidable conclusion.

**Tactic 7.5.5 (CompactDNHyp)** Uses the Double-Negation Compaction lemma to compact the named hypothesis.
Tactic 7.5.6 (CompactDNConcl) Uses the Double-Negation Compaction lemma to compact the conclusion.

CompactDNHyp and CompactDNConcl might seem unnecessary. However, they are essential for double-negation-translating direct-computation proof steps, as we shall see in the next section.

7.5.2 The Translation on Proofs

The translation on proofs is given in terms of the translation on terms. We specify for each rule of Classical Nuprl how to translate the rule into a Nuprl. Of course, the hard part is that the goals and (propositional) hypotheses are all double-negated.

Lemma 7.5.5 (Double-Negation for Nuprl Proof Steps) For any rule r of Classical Nuprl in a complete Classical Nuprl proof tree, suppose the proof step using that rule looks like:

\[ H \gg T^P \]
\[ \text{by } r \]
\[ H_1 \gg T_1^P \]
\[ H_2 \gg T_2^P \]
\[ \vdots \]
\[ H_n \gg T_n^P \]

Then there is a combination of rules of Nuprl (without excluded middle) which, when applied to the translation of the top sequent \( H \gg T^P \), yield the translations of a (possibly empty) subset of the subsidiary sequents \( H_1 \gg T_1^P \) thru \( H_n \gg T_n^P \) as their unproven subgoals.

Proof: A proof of this lemma, of course, consists of a proof by cases over all the rules of the Nuprl type theory. We will do only a few, for space (and exhaustion) reasons. In the cases we present, we will mark the subgoals which are left over (the yielded subset) with asterisks (*).

Excluded Middle The classical rule of excluded middle is:

\[ \Gamma \triangleright C \triangleright \Phi \lor \neg \Phi \]
\[ \text{BY magic at i} \]
\[ \triangleright C \triangleright \Phi \in \text{Prop}_i \]

the double-negation translation of the goal is

\[ \neg\neg(\neg\neg(\Phi) \lor \neg\neg(\Phi)) \Rightarrow \neg\neg(\text{void}) \]

which is equivalent to \( \neg\neg(\Phi \lor \neg(\Phi)) \). This goal is provable constructively. The well-formedness goals that come up are all for variously double-negated
versions of $\Phi \lor \neg \Phi$, $\neg(\Phi)$, and $\Phi$, and all these are provable automatically from the given proof of $\triangleright \Rightarrow \Phi \in Prp_i$. So we can prove this goal completely, and there are no subgoals left over.

**Universal Elimination for Data Types** Universal elimination is simple. The classical proof step looks like

$$H, (x^D : A^D \to B^P)^P, H' \Rightarrow \Rightarrow T^P \text{ by } \text{elim } f \text{ on } a$$

$$\triangleright \Rightarrow a^D \text{ in } A^D$$

$$B^P[a^D/x] \Rightarrow \Rightarrow T^P$$

and, ignoring the extra hypotheses $H$ and $H'$, the double-negated goal is $\neg(\neg(x^D : A^D \to B^P)) \Rightarrow \Rightarrow \neg \neg T^P$. Since $\neg \neg T^P$ is outermost a double-negation, we can use the UncoverDNHyp procedure to remove the double-negation from the hypothesis, and then do the elimination, with $\overline{a}$ instead of $a$. Since the membership goal is automatically provable, all we are left with is $\neg \neg B^P[a^D/x] \Rightarrow \Rightarrow \neg \neg T^P$, which is the translation of the last goal of the classical proof step.

**Universal Introduction for Data Types** Universal introduction is much like universal elimination. The classical rule is:

$$H \Rightarrow \Rightarrow (x^D : A^D \to B^P)^P \text{ by intro at } U_i \text{ [new } y\text{]}$$

$$\triangleright \Rightarrow A^D \in Data_i$$

$$y^D : A^D \Rightarrow \Rightarrow B[y^D/x]^P$$

This time, we use UncoverDNConcl to "pop" the double-negations off the conclusion, and then do the same introduction as above:

$$\Rightarrow \Rightarrow (x^D : A^D \to \overline{B^P})$$

by UncoverDNConcl

$$\Rightarrow \Rightarrow x^D : A^D \to \overline{B^P}$$

by intro at $U_i$ [new $y$]

$$\Rightarrow \Rightarrow A^D \in Data_i$$

* $y^D : A^D \Rightarrow \Rightarrow \overline{B}[y/x]$  

The membership goal is automatically provable, so all we are left with is the single $*$-ed goal.

**Lambda Formation** The rule of lambda-formation justifies the construction of inhabitants of function spaces. It is a membership goal of the form $H \Rightarrow \Rightarrow \lambda \ x. b \in x : A \to B$, and as such, a classical proof of this sequent is automatically convertible to a constructive proof (since, of course, a classical proof of a membership goal is in fact a constructive proof). Thus, we have a proof of $H \Rightarrow \Rightarrow \lambda \ x. b \in x : A \to B$, and from this we get trivially $H \Rightarrow \Rightarrow \neg(\lambda \ x. b \in x : A \to B)$.  

7.5 Double-Negation Translation on Terms, Sequents, and Proofs

**Arith** The arith rule is difficult to describe, since it is an inference procedure which justifies a whole range of goals. Basically, it uses hypotheses of the forms \( x < y, \ x = z \in \text{Int}, \ \text{void}, \ \neg(x < y), \ \neg(x = z \in \text{Int}), \) and conjunctions of these terms as hypotheses, and proves a goal of the same form. As a special case, it can also prove an arbitrary goal from a contradictory set of hypotheses, but we will assume that this does not happen, as we can guarantee that the only non-arithmetic goal that arith is ever given to prove is void. The proof procedure that arith follows is as follows:

- perform elimination upon all conjunctions and existentials repeatedly until there are no more remaining
- accumulate all hypotheses of the previously mentioned forms
- form a graph internally of these hypotheses, and the conclusion, and verify that the conclusion actually holds.

We translate this procedure as follows: Starting with the double-negated sequent,

1. repeatedly apply the procedure UncoverDNHyp to every hypothesis which is outermost double-negated, followed by conjunction and/or existential elimination.
2. Apply DNerasureHyp to every arithmetical hypothesis
3. Apply DNerasureConcl to the conclusion
4. Apply arith to the resultant sequent.

The first three steps above basically produce a sequent which has all the same decidable arithmetic assumptions as the original classical sequent had. DNerasureHyp and DNerasureConcl can be applied, because we know that arithmetic propositions are decidable. Thus, we can assume that DNerasure can complete its job, and convert the sequent to a form in which all the assumptions that arith would have available in the classical sequent are available here also. Thus, arith will succeed, since it did so for the classical sequent. The only subgoals that arith produces are of the form \( t \in \text{Int}, \) and since we have proofs of these in the classical proof and they are well-formedness goals, we can automatically translate those proofs into proofs in our double-negated proof also. There are no remaining subgoals.

**Substitution** The substitution rule looks like:

\[
H \gg T^P[t^D/x^D]
\]

by subst at \( U_i t^D = t'^D \in T'^D \) over \( x.T \)

\[
\gg t^D = t'^D \in T'^D
\]

\[
\gg T^P[t^D/x^D]
\]

\( x^D : T'^D \gg T^P \in \text{Prop}_i \)
So we use the following proof strategy:

\[ \overline{H} \gg TP[t^D/x^D] \]
by Assert \( t^D = t'^D \in T^D \)
\[ \ast \gg t^D = t'^D \in T^D \]
\[ \neg(t^D = t'^D \in T^D) \gg TP[t^D/x^D] \]
by UncoverDNHyp on last hyp then thin the hyp
\[ t^D = t'^D \in T^D \gg TP[t^D/x^D] \]
by subst at \( U_i \) \( U_i t^D = t'^D \in T^D \) over \( x.T \)
\[ t^D = t'^D \in T^D \]
by hypothesis
\[ \ast \gg TP[x.D t^D] \]
\[ \gg TP[t^D/x^D] \]
\[ x^D : T^D \gg TP \in Prop_i \]

The goals marked with asterisks are the only ones left over, and they are indeed the translations of subgoals of the original sequent. The last well-formedness goal is automatically provable from the well-formedness proof in the original proof.

What we did was "cut in" the double-negation of the equality. We get two goals: one, to prove the double-negated equality. This becomes our first subgoal "left over" (it is the double-negation translation of the first subgoal of the classical proof step). The other goal we get is, from the double-negated equality, to prove the original goal. We apply UncoverDNHyp to the equality, and to the proper substitution. The first subgoal of the substitution is trivially proven from the equality hypothesis. The second is also "left over" (the double-negation translation of the second subgoal of the classical proof step). The membership goal, \( x^D : T^D \gg \overline{TP} \in Prop_i \), is automatically provable given the membership goal in the original proof.

Direct computation on hypotheses The direct computation rule on hypotheses allows us to convert a redex in a hypothesis into its contractum without any obligation to prove that the conversion valid. It is justified because the rules of Nuprl respect evaluation. It is also hard to codify, because it involves so much symbolic manipulation of the hypothesis. Arbitrary amounts of computation can occur at arbitrarily many redex positions in a hypothesis. The rule provides a copy of the hypothesis to be computed upon, with subterm occurrences marked for computation with the number of reduction steps to be carried out at each occurrence. For instance, if we had a hypothesis the term \((\lambda x.x)(True)\), we could reduce it by using the rule direct computation hyp \([1; (\lambda x.x)(True)]\). The hypothesis would then be
reduced to True. It turns out that we can compose arbitrary direct computations on a term from the simple direct computation rule that specifies a single redex to be evaluated a single step. We will justify the double-negation translation of this rule. In the case where the hypothesis being computed down is a data-type, the rule instance looks like:

\[ H, x^D : C[r]^D, H' \gg T^P \]
by simple direct computation hyp \( C[[1; r]] \)
\[ H, x^D : C[t]^D, H' \gg T^P \]

where \( r \) reduces to \( t \) in a single outermost reduction step. Since the hypothesis is a data-type, it does not contain any propositions, so it's double-negation translation is itself, and we need do nothing to translate this rule.

If the hypothesis is a proposition, the rule instance looks like:

\[ H, C[r^*]^P, H' \gg T^P \]
by simple direct computation hyp \( C[[1; r]] \)
\[ H, C[t^*]^P, H' \gg T^P \]

and there are two cases:

Case 1 \( r^* = r^D \): then \( r^D \) is not outermost double-negated, and an analysis of evaluation shows that \( r^D \) outermost-reduces to \( t^D \), so the translated rule is:

\[ \overline{H}, \overline{C[t^D]^P}, \overline{H'} \gg \overline{T^P} \]
by simple direct computation hyp \( C[[1; t^D]] \)
\[ \overline{H}, \overline{C[t^D]^P}, \overline{H'} \gg \overline{T^P} \]

Case 2 \( r^* = r^P \): \( r \) is a proposition. Suppose \( r^P \) outermost-reduces to \( t^P \), and that \( \overline{r^P} = \neg\neg(s) \) (since \( r \) is a proposition, \( \overline{r^P} \) is automatically outermost double-negated. We know that \( \overline{C[r^P]} = \overline{C[s]} \) since in \( \overline{C[\bullet]} \) the (propositional) "hole" is already double-negated. \( s \) must be a redex, by the rules for double-negation translation of terms. Assume that \( s \) outermost-reduces to \( u \). \( u \) may have outermost double-negations also. If \( u \) does not, then \( \overline{C[t^P]} = \overline{C[u^P]} \).

Suppose \( v \) is \( u \) after stripping all outermost double-negations. Then \( \overline{C[t^P]} = \overline{C[v]} \). This easy to see if we look at an example:

\[
(\lambda Q^P.Q^P)(A^P) = \neg\neg((\lambda Q.\neg\neg(Q))((\neg\neg(A)))
\]
and \( (\lambda Q^P.Q^P)(Q^P) \) outermost-reduces to \( A^P \). We would like to mimic this behaviour on \( \neg\neg((\lambda Q.\neg\neg(Q))((\neg\neg(A))) \). We do so by letting \( C[\bullet] = \neg\neg(\bullet) \),
and computing \((\lambda \ Q. \neg\neg(\neg(Q)))(\neg\neg(A))\) to \(\neg\neg(\neg\neg(A^P))\) in one step, and then showing that \(\neg\neg(\neg\neg(A^P)) \Leftrightarrow \neg\neg(A)\), which is a consequence of the double-negation compacting lemma.

In the general framework above, what we need to justify is: for a propositional context with a propositional "hole", \(C^P[\bullet^P]\), if \(u \Leftrightarrow v\), then \(C[u] \Leftrightarrow C[v]\). The double-negation compacting lemma does this for us.

So the translation rule is:

\[
\overline{H}, \overline{C^P[r^P]}, \overline{H'} \gg \overline{T^P}
\]

by simple direct computation hyp \(C[[1;r^P]]\)

\[
\overline{H}, \overline{C^P[t^P]}, \overline{H'} \gg \overline{T^P}
\]

by DNCompactHyp \(\overline{C^P[t^P]} \Leftrightarrow \overline{C^P[u^P]}\)

\(* \ C^P[u^P] \gg \overline{T^P}\)

where \(r\) outermost-reduces to \(t\) in one step, and \(u\) is \(t\) after stripping all outermost double-negations. Clearly, the subgoal left over is the double-negation translation of the original subgoal.

**Direct computation on conclusions** We can translate direct computation steps on conclusions in much the same way.

**Universal Elimination for Propositions** Universal elims on propositions can also be complicated. We will sketch out the solution for this problem: If we eliminate the hypothesis \(\forall Q \in Prop. \neg\neg(\neg\neg(Q) \Rightarrow \neg\neg(Q))\) on the double-negation of some proposition variable \(A\), for instance, we will supply a term \(\neg\neg(A)\); the new hypothesis will be \(\neg\neg(\neg\neg(A)) \Rightarrow \neg\neg(\neg\neg(A))\). We would like instead the hypothesis \(\neg\neg(\neg\neg(A) \Rightarrow \neg\neg(A))\), and the problem we have to solve is the same as was had to solve for the direct computation rule. In fact, this problem comes up whenever we allow substitution or computation at propositional positions in a proposition. Again, though, we can simply use CompactDNHyp to compact this hypothesis to the desired form.

**the seq (or "cut") rule** The seq rule is trivially translatable. In a classical proof, it looks like:

\[
H \gg T^P
\]

by seq \(T_1^P, \ldots, T_n^P\)

\(\gg T_1^P\)

\(T_1^P \gg T_2^P\)

\(\vdots\)

\(T_1^P, \ldots, T_n^P \gg T^P\)

and its translation is:
7.6 A-Translation in Nuprl°

\[
\overline{H} \gg \overline{T^P} \\
\text{by seq } \overline{T_1^P}, \ldots, \overline{T_n^P} \\
\gg \overline{T_1^P} \\
T_1^P \gg \overline{T_2^P} \\
\vdots \\
T_1^P, \ldots, T_n^P \gg \overline{T^P}
\]

There are other proof steps for which the justification is also complicated, but we will leave those out for the sake of brevity. It should be clear at this point, though, that indeed we have proven that every classical Nuprl° proof step can be translated in the appropriate fashion into a double-negated proof step.

**Theorem 7.5.1 (Double-Negation Translation of Nuprl° Proofs)** For any complete Classical Nuprl° proof tree, there is a complete Nuprl° proof tree.

**Proof:** By structural induction on the classical proof and appeal to the previous lemma.

### 7.6 A-Translation in Nuprl°

#### 7.6.1 A-Translating Terms and Sequents

To A-translate Nuprl° terms is trivial. Given a proposition \( \Phi \), simultaneously disjoin every atomic propositional subterm of \( \Phi \) with \( A \). Let us give some examples:

\[
(U \lor V)^A \quad \longrightarrow \quad (U^A \lor V^A) \lor A \\
(\forall x \in D.B)^A \quad \longrightarrow \quad \forall x \in D.B^A \lor A \\
(U \Rightarrow V)^A \quad \longrightarrow \quad U^A \Rightarrow V^A \lor A \\
(\lambda x. \Phi(x))(1)^A \quad \longrightarrow \quad (\lambda x. \Phi(x)^A)(1) \lor A
\]

Friedman [Fri78] and Leivant [Lei85] use an A-translation which does not disjoin universal quantifications, existential quantifications, implications, conjunctions, or disjunctions with \( A \). In particular, omitting the disjunction of existentials only works if we know \textit{a priori} that all data-types are inhabited. Since we admit data-types such as \( T : Data_1 \rightarrow T^D \), clearly all data-types need not be inhabited. Thus, we need to use this modified translation.

The A-translation on sequents is simple, also. If the sequent we wish to translate is: \( H \gg T \), we A-translate all propositional hypotheses and the conclusion, as outlined above. We denote this translated sequent by \( H^A \gg \Phi^A \).

#### 7.6.2 Some Results about A-Translation

We will need some results about A-translation to prove our translation correctness result. They are listed below, mostly without proof:
Fact 7.6.1. For any proposition $\Phi$, $A \gg \Phi^A$.

Fact 7.6.2. For any term $\Phi$, we can prove $\Phi^A \leftrightarrow \Phi^A \lor A$.

Fact 7.6.3 (A-Translation Erasing) For any proposition $\Phi$ which is hereditarily decidable, and none of whose subterms is a predicate (i.e., has a type $T_1 \rightarrow \cdots \rightarrow T_n \rightarrow \text{Prop}_1$) we can prove $\Phi \lor A \leftrightarrow \Phi^A$.

Proof: By induction over the structure of propositions. Let us give just the case for implication:

Suppose $\Phi \Rightarrow \Psi$ is hereditarily decidable, $\Phi^A \leftrightarrow \Phi^A \lor A$, $\Psi^A \leftrightarrow \Psi \lor A$.

Show: $(\Phi \Rightarrow \Psi)^A \leftrightarrow (\Phi \Rightarrow \Psi) \lor A$. Case 1 $\Rightarrow$: From $\Phi^A \Rightarrow \Psi^A \lor A$, we get $((\Phi \lor A) \Rightarrow (\Psi \lor A)) \lor A$. From $A$ we get the desired conclusion. From $(\Phi \lor A) \Rightarrow (\Psi \lor A)$, by first asserting $\Phi \lor \neg \Phi$ and $\Psi \lor \neg \Psi$, and arguing by cases, we can prove the desired conclusion.

Case 2 $\Leftarrow$: Trivial: $A \Rightarrow (\Phi^A \Rightarrow \Psi^A)$. $\Phi \Rightarrow \Phi^A$ and $\Psi \Rightarrow \Psi^A$ together do the rest.

The condition that none of the subterms of the term in question be predicates means that there can be no disjunctions with $A$ inside of terms which are not patently logical. For instance, in the term $P(\lambda \ x. x = 1 \in \text{Int})$, if $P \in \text{Int} \rightarrow \text{Prop}_1$, and $P$ is decidable on decidable arguments, we would like to be able to prove that $P(\lambda \ x. x = 1 \in \text{Int})^A \leftrightarrow P(\lambda \ x. x = 1 \in \text{Int}) \lor A$. But if $P$ is a higher-order predicate variable, this isn’t possible. this isn’t a big loss, since in most cases decidable formulas are simple, and are not composed of complex applications of higher-order predicate symbols.

Lemma 7.6.1 (A-Translation Compaction) Given a well-behaved proposition $\Phi$, if we look at a propositional subterm occurrence $\Phi = C[r \lor A]$, we can prove $C[r \lor A] \leftrightarrow C[(r \lor A) \lor A]$.

Proof From the facts that $r \lor A \leftrightarrow (r \lor A) \lor A$, and $\Phi$ is well-behaved.

This lemma allows us to “compact” a term which has double-A-disjunctions, that is, $T \lor A \lor A$ to a term $T \lor A$.

We can codify these lemmas and facts in tactics:

Tactic 7.6.1 (CompactAHyp) Given a well-behaved proposition $\Phi$, this tactic rewrites it to another term $\Phi'$ such that $\Phi \leftrightarrow \Phi'$, and $\Phi'$ contains no instances of double-A-disjunctions.

Tactic 7.6.2 (CompactAConcl) performs the same rewrite on a conclusion as CompactAHyp.

Tactic 7.6.3 (ProveFromA) Justifies the goal $A \gg T^A$.

Tactic 7.6.4 (EraseAHyp) Given a hypothesis $\Phi^A$ such that $\Phi$ is hereditarily decidable and contains no predicates as subterms, EraseAHyp rewrites this hypothesis to $\Phi \lor A$. 
Finally, we have the following facts about well-formedness:

**Lemma 7.6.2 (A-Translated Membership Goals)**
Given a goal $H \gg t \in T$ in a Nuprl$^o$ proof, the goal $H^A \gg t^A \in T$ is provable in Nuprl$^o$.

**Proof:** This theorem is proven much like the analogous theorem for double-negation translation.

**Lemma 7.6.3 (A-Translated Well-Foundedness)**
Given a sequent in a Nuprl$^o$ proof, for every proposition $\Phi^A$ in the A-translated version of that sequent, we can automatically prove the well-formedness of $\Phi^A$.

**Proof:** From the analogous theorem about double-negation, and a simple argument on the size of well-formedness proofs.

### 7.6.3 A-Translation on Proofs

We will now prove the central result about A-translation:

**Lemma 7.6.4 (A-Translation for Nuprl$^o$ Proof Steps)**
For any rule $r$ of Nuprl$^o$ in a complete proof tree, suppose the proof step using that rule looks like:

$$H \gg T^P$$

by $r$

$$H_1 \gg T_1^P$$

$$H_2 \gg T_2^P$$

$$\vdots$$

$$H_n \gg T_n^P$$

Then there is a combination of rules of Nuprl$^o$ which, when applied to the A-translation of the top sequent $H \gg T^P$, yield a (possibly empty) subset of the A-translations of the subsidiary sequents $H_1 \gg T_1^P$ thru $H_n \gg T_n^P$ as their unproven subgoals.

**Proof:** Again, by cases over all the rules of Nuprl$^o$. We will demonstrate for only a few cases:

**Universal Elimination for Data Types** The original proof step looks like:

$$H,(x^D : Dom^D \rightarrow B^P)^P,H' \gg T^P \text{ by } \text{elim } f \text{ on } a$$

$$\gg a^D \text{ in } Dom^D$$

$$B^P[a^D/x] \gg T^P$$

The translated proof step is:
$H^A, (x^D : \text{Dom}^D \rightarrow B^A) \lor \neg H^A \gg T^A$

by or-elimination

$x^D : \text{Dom}^D \rightarrow B^A \gg T^A$

by elim on $a^A$

$\gg a^A \in \text{Dom}$

* $B^A[a^A/x] \gg T^A$

The membership goal $\gg a^A \in \text{Dom}$ is provable automatically from the original proof of $\gg a \in \text{Dom}$, and we are left with the goal $B^A[a^A/x] \gg T^A$, which is the A-translation of the last original subgoal. Note that since $a$ is a data-object, we know, as in the double-negation case, that $B[a/x]^A = B^A[a^A/x]$.

Universal Introduction for Data Types This case is even easier. The original proof step is:

$H \gg (x^D : \text{Dom}^D \rightarrow B^P)^P$ by intro at $U_i$ [new $y$]

$\gg \text{Dom}^D \in \text{Data}_i$

$y^D : \text{Dom}^D \gg B[y^D/x]^P$

Here we use Left-Or-Introduction to remove the outermost disjunction $(\forall x \in D. B^A \equiv (\forall x \in D. B^A) \lor A)$:

$\gg (x^D : \text{Dom}^D \rightarrow B^P)^A \lor A$

by Or-Intro-Left at $U_i$

$\gg x^D : \text{Dom}^D \rightarrow B^P$

by intro at $U_i$ [new $y$]

$\gg \text{Dom}^D \in \text{Data}_i$

* $y^D : \text{Dom}^D \gg B[y/x]$

$\gg A \in \text{Prop}_i$ by assumption

and the only remaining subgoal is, of course, the A-translation of the last subgoal of the original proof step.

Arith The arith rule is trivial to A-translate, since we know that any usable hypothesis of arith, and any conclusion provable by arith, is hereditarily decidable. So what we do is apply EraseAHyp to all applicable hypotheses, apply EraseAConcl to the conclusion, and then apply arith to the resulting sequent. Arith is guaranteed to succeed, since this resulting sequent has all the same decidable hypotheses, and the same conclusion, as the original sequent. The only subgoals generated are well-formedness, and these can be automatically proven from the analogous goals in the original proof tree.
direct computation on hypotheses Direct computation on hypotheses proceeds in much the same manner as it did for double-negation. In fact, the method is identical, only instead of compacting double-negations, we compact A-disjunctions with CompactAHyp.

All of the rules of Nuprl⁰ can be translated thus. We use this to prove

Theorem 7.6.1 (A-Translation of Nuprl⁰ Proofs) For any complete Nuprl⁰ proof tree with goal G, there is a complete Nuprl⁰ proof tree with G^A as its goal.

Proof: Again, by trivial structural induction on the original Nuprl⁰ proof tree. Finally, we can prove

Theorem 7.6.2 (Conservative Extension for Nuprl⁰) Given a proof in Classical Nuprl⁰ of a \(\Pi^0_2\) statement, we can construct a proof in (constructive) Nuprl⁰ of the same statement.

Proof: By the same technique used in proving the conservative extension result in chapter 6.

7.7 Extensions and Implementation

Of course, the above theorems apply to the theory Nuprl⁰, which is pretty ugly and hard-to-use. However, it turns out that we can apply the same translations to Nuprl, if we allow the possibility that the translation can fail. That is, we can look at these translations as being applied to Nuprl⁰ proofs, in which case we must first translate a Nuprl proof into a Nuprl⁰ proof (which is a painful, laborious task), or we can write a heuristic translator from Nuprl to Nuprl⁰ which works most of the time, and then think of our translation apparatus as being the heuristic in addition to the translator detailed above. That is, for “well-behaved” proofs in Nuprl, the translation will succeed. But, if we try to translate a proof with a hidden use of the axiom of choice, for example, the translation procedure will fail, as the heuristic will fail to produce a Nuprl⁰ annotation. In the places where the translation procedure needs to know the annotations on terms, we simply substitute an automatic inference procedure that infers them based on whatever information it can find. It turns out that this is what we did in practice, and we found that for large, complicated proofs, this procedure worked very well.

We can extend these results somewhat to encompass the set-type. It turns out that we can allow use of the set-type \( \{ x : A \mid B(x) \} \), provided that \( B(x) \) is hereditarily decidable (\( B(x) \) is decidable, and so are all of its subterms). The basic idea is that, whenever we do a set-elimination, converting a hypothesis \( x : \{ y : A \mid B(y) \} \) into the pair \( x : A, [B(x)] \), where the brackets around \( B(x) \) tells us that this hypothesis is not available for computational use, we prove the goal \( B(x) \lor \neg B(x) \), and use
this to prove $B(x)$ (without the brackets). It is a standard technique to convert noncomputational information into computational information.

We can handle notational abstractions by treating them as prime formulas, like predicate symbols, and then using the same strategy that we used for direct computation to normalize propositions in which abstractions have been unfolded. That is, if we had an abstraction $Decidable(x) \equiv x \lor \neg(x)$, the double-negation of this definition would be $DNDecidable(x) \equiv \neg(\neg(\neg(x) \lor \neg(\neg(x))))$, and an instance $Decidable(x)$ would be double-negated as $\neg(\neg(DNDecidable(\neg(x))))$, and when expanded simply minded, we would get

$$\neg(\neg(\neg(\neg(x)) \lor \neg(\neg(\neg(x))))).$$

This is clearly unsatisfactory, but we can prove automatically that this term is equivalent to $\neg(\neg(x) \lor \neg(\neg(x)))$, and this is what we do.

We can handle the lemma facility by using a naming convention for classical and double-negated lemmas, and translating an instance of a classical lemma into an instance of the corresponding constructive lemma.

### 7.8 Summary

To summarize, in this chapter we specified a modification of a subtheory of Nuprl which admitted double-negation translation. We showed that this subtheory was one for which translation would always be successful, and we argued that the subtheory was close enough to Nuprl that a simple heuristic which decorated Nuprl proofs with additional information was sufficient to make translation possible. We then showed how to extend this work to lemmas, notational abstraction, and the Nuprl "set" type (used to hide noncomputational information).
Chapter 8

The Implementation of Proof Translations in Nuprl

In this chapter we will discuss our experiences in implementing the double-negation and A-translations in Nuprl. We will discuss the implementation decisions we made, their effect upon the performance and usability of the translation system, and some of the conclusions about theorem-proving system organization and implementation which came out of this experiment. We will also discuss the results of the translation experiments. We begin by discussing, in gross generalities, the translation project.

8.1 The Translation Project

We built a general-purpose translation system which (semi-) automatically translated a classical proof of Higman's Lemma into a constructive proof. The translation method, as discussed in Chapters 6 and 7, consisted of two stages, double-negation translation and A-translation. The entire classical proof measured approximately half a megabyte in its disk-resident form. The final translated form took nearly 100 Megabytes of disk storage. However, the classical proof was done using tactics, which raise the level of inference in Nuprl, so this comparison is a bit unfair. A better measure of the size of the task we undertook is the total in-memory size of the proofs. Since we were running in a LISP system, we do not have any real numbers for the in-memory size of the proofs. However, the entire classical proof, fully checked, fit into 40 megabytes of memory. On the other hand, the fully checked translated proof required well in excess of a gigabyte of space in order to hold in memory. In fact, the proof was so large that it had to be constructed on five separate machines, and the constructive extract was then dumped out to a file, and loaded back into a single machine for execution. Part of the reason for this blowup was the inefficiency of the translation tactics; another large part of the blowup resulted from the fact that before each translation phase we had to expand the tactic invocations in the original proof tree into primitive rules. Then each primitive rule was translated to a tactic invocation. As we discuss in Section 8.7, a large part of
this second blowup could have been avoided if we had available an abstract syntax
tree representation of tactic expressions.

As the reader can tell, our translation project was a large one. We learned quite
a bit about the problems that occur in large theorem-proving projects, and we feel
that much of this knowledge can be of use elsewhere. In the following sections, we
describe at a high level the task of proof translation, describe the results of the
experiment, and discuss the strategies which we believe would make the task of
proof translation more efficient in the future.

8.2 A High Level Overview of Proof Translation

Proof translation in the mathematical world is a process which, by structural in-
duction upon proof trees, constructs new proof trees from old ones. One can view
proof translation as nothing more than a syntax-directed translation upon particu-
larly complex abstract syntax trees. However, unlike more normal syntax-directed
translation, proof translation in our experiment, and in much of theorem-proving,
is used to translate truly huge proof trees. As we mentioned before, our final proofs
took well over a gigabyte of virtual memory to store online. We estimate that our
proofs contained at least a quarter of a million sequents, and probably more than
that.

As such, standard techniques in syntax-directed translation such as attribute
grammars were simply unsuitable for the tasks at hand. We had to resort to more
efficient techniques. We decided upon a system whereby we constructed the trans-
lated proof in a top-down pass on the original proof tree. A proof tree consists
of sequents and either primitive rule invocations or tactic invocations. The tactic
invocations are strings, and we required that we could dump out translated proof
trees to disk storage and reload them without corrupting the trees, as for normal
proof trees. To achieve this, we had to guarantee that tactic expressions did not
reference global assignable variables (that might have different values the next time
we invoked the system). We did this by making all global references from the trans-
lation tactics be indices into hash tables. We stored the original proof tree in a
hash table, which we constructed before we ran the translator. Any information
that a tactic needed about the original proof tree was obtained by lookups in the
hash table with an integer key.

Another important task was the inferring of properties of sequents and terms
in sequents. The annotation of Nuprl proofs to yield Nuprl⁰ proofs, developed in
Chapter 7, was achieved on-the-fly thru various heuristics which identified when a
given term was a proposition or a data-type.

Since our translation system is only heuristic, being implemented for Nuprl, and
not for Nuprl⁰, we had to write numerous tactics to “patch up” proof trees that were
only partially translated by the system. We required efficient routines for taking
apart proof trees, modifying them, and putting them back together again.
The proof translation system we built was quite simple (in principle). It consisted of a term-translation component which, given a term \( t \) found in a classical proof, would compute the double-negation translation of that term. This involved considerable amounts of structural induction over the term, and computing a typing of every subterm of the term also.

Using this component, the translation task boils down to a tactic which is fed two arguments: the root of the translated proof tree, and the complete classical proof tree. The tactic then inspects the top refinement step in the classical proof tree, constructs analogous refinement step(s) to apply to the translated proof tree, and, after applying them, collects the subgoals generated, and applies itself recursively to the subgoals and the corresponding children of the classical proof tree root. Since the translated subgoals may be in a different order from the original subgoals, or fewer in number, we must also compare each translated subgoal against each of the original subgoals in turn until a match is found, at which point we can recurse.

Of course, the difficult part is writing the tactics which mimic single refinement rules in the classical theory. As we pointed out in Chapter 7, rules such as direct computation are tremendously difficult to handle in their full generality. Nevertheless, we chose to implement them completely, in order to stay as faithful to the standard method of formalizing mathematics as possible. Most of the work of writing such tactics is simply to write programs which implement the steps outlined in Chapter 7, and we will not repeat them here. The difficult part of translating refinement steps almost always is performing syntax-directed translation of term arguments in the classical refinement step to compute the constructive refinement step.

The process of \( A \)-translation is much the same, except that we must \( A \)-translate terms instead of double-negation translating.

### 8.3 Comparisons with Other Theorem-Proving Systems

Nuprl presents the user with a rich array of tools which make proof translation and construction much easier. To begin with, the idea of a proof tree as a structured data type in the metalanguage, which is absent from, for instance, the Edinburgh LCF [Pau85] is obviously essential. This same critique can be made of Isabelle [PN90], though with less success, since there the witness-objects could ostensibly be constructed to contain enough of the derivation that they could be used for translation.

The Theory of Constructions theorem-provers [Hue] build constructive witnesses which contain essentially the same information as a Nuprl derivation (sequent) tree. Thus, the translation project we undertook would be easily doable in most Constructions theorem-provers. However, since most Constructions theorem-provers do not maintain derivation trees, but only the equivalent of a primitive refinement
tree (no tactic information is kept), the improvements we suggest in Section 8.7 would not be possible there. To be fair, though, the work involved in implementing some abstract syntax tree representation of tactic expressions is great enough that adding a derivation-tree editor to a Constructions system would be a minor part of the entire task.

At present, no system we know of has a representation of metalanguage expressions as abstract syntax trees.

### 8.4 The Actual Proof Translation Experiment

We implemented the entire translation system in Cambridge ML, the Nuprl metalanguage, and used it to (almost automatically) translate the classical proof of Higman's Lemma into a constructive proof. In addition, we also translated two other proofs:

1. a proof of another theorem, that every decreasing function from \( \mathbb{N}^+ \) to \( \mathbb{N}^+ \) has a "one" (that is, achieves the value 1). This proof was obtained via a minimal bad sequence argument much like that for Higman's Lemma.

2. a proof that every infinite sequence of bits has two zeroes or two ones. This proof was obtained via simple, predicative means.

The results of these translations were quite illuminating, especially in light of the work (in Chapters 9 and 10) on continuations and double-negation translation. We found that the program extracted from the predicative proof ran as expected, with no real problems and with correct outputs, as would be expected for programs proven correct in Nuprl. The two programs extracted from the translated minimal bad sequence proofs (Higman's Lemma and the "decreasing function" theorem), both terminated on the most trivial input. In the case of Higman's Lemma, which requires an infinite sequence of lists of integers from some finite set, the function was \( \lambda x. \text{nil} \). In the case of the decreasing theorem, the function was \( \lambda x.1 \).

However, neither program extracted from a translated minimal bad sequence proof terminated in vast amounts (a quarter gigabyte) of swap space on a SUN SparcStation. We ran the program extract of Higman's Lemma on several other machines, all to no avail. With the foreknowledge of the next few chapters, we can attempt an answer to the question of why the translated programs failed to run correctly. The answer lies in the fact that the Nuprl term evaluator, like many for theorem-provers, is a standard recursive evaluator in the style of the early LISP evaluators. It does not recognize tail recursion, and as such, given a tail-recursive term to evaluate, it will consume enormous amounts of stack space during its execution. In fact, the amount of stack consumed is proportional to the number of redices contracted. On the other hand, an evaluator which recognized and optimized away tail-recursion would not suffer from this problem, and would almost certainly run much more efficiently. With the knowledge that, in the next two
chapters, we will prove that double-negation/A-translation are the same as CPS-compilation, which produces tail-recursive code, we can conclude that the minimal bad sequence programs failed to terminate because they ran out of stack space, and that a tail-recursion-optimizing evaluation strategy is needed to run these programs.

We did not implement such an evaluator because the Nuprl term set would have made such an evaluator a large undertaking, and because Nuprl is scheduled for re-implementation soon, at which point the modified evaluator would become useless (since it would not be compatible with the re-implemented system).

8.5 Computational Metatheory and Efficient Translation

One of the greatest roadblocks to efficient translation is the large number of superfluous steps in a proof. Usually, there are many, many hypotheses which can easily be ignored in any particular sequent. Likewise, there are often many proof steps which it would be safe to skip in proving a particular goal. Thus, there are two “cleanup” translations which we identified as being important for generating the smallest proofs possible.

First, the elimination of gratuitous proof steps, and the elimination of unnecessary hypotheses, proceeding hand-in-hand, would reduce the size of proofs tremendously. Moreover, this would reduce the numbers of hypotheses which needed to be maintained (to satisfy inductive invariants) by the proof-translator.

The second cleanup operation which we believe is necessary is the contraction of large numbers of $\beta$-redices (and other redices) in the proof. This cleanup is quite expensive, and could result in a blowup in the size of the proof. However, there are several ways to control the blowup. Most notably, we can implement the colon-translation of Plotkin (described in Chapter 9) which contracts “administrative” redices in a translated term. This translation reduces significantly the size and complexity of translated proofs, and we feel it is essential to efficient double-negation translation.

8.6 Exploiting Gratuitous Parallelism

The amount of parallelism available in sequent proof translation is tremendous. After just a moment’s reflection, this should be apparent. To exploit this parallelism, we used multiple machines each running a copy of Nuprl and translating different lemmas. This method worked relatively well, but after some reflection, we realized there was much more parallelism than we could ever hope to exploit with such a heavy-handed approach.

Most mathematical proofs consist of arguments by induction, rewriting, cases, etc, connected by propositional reasoning. These higher-level reasoning schemas,
8.7 Efficient Translation of Tactic Expressions

A Nuprl proof tree node consists of a (possibly empty) list of terms, the hypotheses, a term, the conclusion, a (possibly empty) list of subgoals, which are proof trees, and a refinement, which is either null, a primitive refinement rule, or a string, which is a tactic expression. If the rule for a node $p$ is a tactic expression, then evaluating that tactic expression with argument $p$ will produce a proof tree which can be used under $p$, and has the property that it contains only primitive refinement rules. Thus, we can “expand” proof tree with instances of tactic expressions in it into an equivalent proof tree with only primitive rules.

We did this before translating the proofs, and translated at the level of individual refinement steps. There is a tremendous cost to working at this level. For instance, the single rule of direct computation is used as a workhorse to encode normalization (reducing every available redex, even those defined via notational abstraction), reduction (reducing all redices except those defined via notational abstraction) and unfolding (unfolding a notational abstraction, leaving everything else fixed). If we convert everything down to the level of primitive refinement rules, we lose all information about the actual use of a refinement rule, which could make figuring out how to translate a rule much easier. But Nuprl stores tactic expressions as strings, so it is impossible to infer such information without parsing tactic expressions. This is, of course, exactly what the compiler does, and if we could reach into the compiler and use its routines to construct tactic parse-trees, we could make a lot of things much more efficient.

The version of ML we use, Cambridge ML [GMW79], is over ten years old; hence, it really is not designed with applications as large as Nuprl in mind. In a more advanced implementation, e.g. Standard ML of New Jersey [AM87], we could use the system parser to construct our syntax trees of tactic expressions. The CAML system [For89] provides the user with ways of parsing ML expressions into abstract syntax trees. Combining this with an interpreter for these trees, we could evaluate complex tactic expressions down to their building-blocks (a carefully-chosen set of basic tactics that we have written translations for) and then translate the invocations of these basic tactics. As time went by, we would expect that people would hand-translated more and more complex tactics, with corresponding increases in the efficiency of the translation process.
With parsed tactic expressions, we could translate tactics at a very high level, thus achieving a higher-level translation of classical to constructive proofs. A side benefit of this would be that the translated proof would be much more readable than the translated proofs we generate currently.
Chapter 9

Double-Negation, Compilation, and Continuations

In the following two chapters, we will discuss the connections between double-negation translation of formulas and proofs and the notions of continuations and continuation-passing-style (CPS) compilation of programs. We will re-prove Friedman's conservative extension result, this time paying close attention to the exact nature of the program extracted from the constructive proof, and its relationship to the "putative program" extracted from the classical proof. We will demonstrate that the "constructive" content of the double-negation elimination rule is the non-local control operator $C$. In the next chapter, we will extend our work to encompass many different double-negation translations, and demonstrate that the process of double-negation is fundamentally a compilation process which fixes evaluation order for program expressions. In this first section we present an overview of both chapters, and in the following section, a discussion of the background of this work, and the connections with and implications for programming, programming language theory and design, logic, and constructivism.

9.1 Overview of The Next Two Chapters

This chapter will focus on demonstrating that the algorithmic content of double-negation elimination is exactly captured by the control operator, $C$, a relative of call-with-current-continuation. It will combine the two seemingly unrelated ideas of explicit access to the current continuation, and classical reasoning, and show that they are actually identical. To wit, this chapter will demonstrate that the content of the rule of double-negation elimination is exactly $C$, and that the effect of CPS-translation is exactly that of a combined double-negation/A-translation. Hence, we show that the following diagram commutes:
9.1 Overview of The Next Two Chapters

\[ \vdash_{PRL+EM} \forall \exists R \rightarrow A - Translation \rightarrow \vdash_{PRL} \forall \exists R \]

\[ \begin{array}{c}
\lambda x. M \\
\downarrow \text{CPS - Translation} \\
[ ]_J \\
\downarrow \\
b \\
\uparrow \\
[ ]_K
\end{array} \]

where \( \text{ext}_K \) is the classical program extraction procedure, \([ ]_K\) is the classical operational semantics (with \( C \)), the constructive versions (without \( C \)) are subscripted with a \( J \), and \( b \) is a primitive value.

1. We begin with a general discussion of the uses of continuations in programming language semantics, the implementation of programming language features (esp. non-local jumps), and program compilation. As a small example of the theorem we wish to prove, we will restrict ourselves to pure implicational logic, and show:

**Desired Theorem 9.1.1.**  
- The Kolmogorov/A-translation from classical implicational logic to intuitionistic implicational logic is a conservative extension for primitive sentences.
- The CPS-translation upon classical programs is exactly the Kolmogorov/A-translation upon classical proofs.
- For classical witnesses for proofs of prime propositions, the CPS-translation preserves values.

This theorem allows us to conclude that the Kolmogorov translation is nothing more than a CPS-compilation from a nonfunctional language (the lambda-calculus + \( C \)) to a functional language. The intuitive reasoning which would lead us to this conclusion is based on the fact that the Kolmogorov-translation is so simple-minded. It would be (we feel) utterly preposterous for the Kolmogorov-translation to be doing something so complex as "injecting" a construction into a classical proof; the construction must have been there already. It is then a small step to notice that CPS-compilation modifies types in a manner similar to double-negation translation, and then we have all the pieces; and we must simply put them together.

2. We will define the logic Heyting Arithmetic (HA) in complete detail, starting with the programming language, a subset of Nuprl, its evaluation rules, the
forms of propositions in the logic, and finally the rules of the logic. This section will basically be a recasting of Gödel's language $T$ [Ste72].

3. We will extend our logic and programming language to classical reasoning (Peano Arithmetic, or PA) with the rule of double-negation elimination, and its counterpart, the programming language construct $C$. We will define the evaluation semantics of PA, and point out the places where we must modify, rather than simply extend, the evaluation semantics of HA in order to maintain total correctness.

4. We will re-do the theorem just described for our new pair of logics, PA and HA, and show that a particular Kolmogorov translation on formulas is equivalent to a particular CPS-translation on proofs, and that the particular CPS-translation is value-preserving for witnesses of $\Sigma^0_1$ sentences; that is, classical witnesses for such sentences are translated to constructive witnesses, and both evaluate to the same value. In the process, we will demonstrate that for $\Pi^0_2$ sentences, classical proofs are in fact evidence, just as constructive proofs are.

In the next chapter, we extend this presentation as follows:

1. We define other double-negation translation strategies, and show that each double-negation translation strategy is equivalent to specifying a well-defined, easily-understood evaluation order to the original program. Hence, we show that the process of double-negation/A-translation is simply a CPS-translation, and that the choice of different double-negation translations is exactly a choice of different evaluation orders for our source programming language.

2. We discuss some negative results and their meaning in the light of this new interpretation of double-negation translation, and also the extension of these results to higher-order type theories.

3. We conclude with an overview of the entire scope of this research into the connections between classical reasoning, $C$, double-negation elimination, CPS-compilation, and operational semantics.

### 9.2 Continuations, Evaluation, and CPS Compilation

Before we begin with the technical material, we need to understand the context in which our work is best understood. The idea of continuations and continuation-passing-style translation originated with work in the denotational semantics of programming languages. It was further developed by Fischer [Fis72] and Plotkin [Plo75], and bore fruit in the form of efficient compilation techniques in the work of Steele [Ste78] and others [KKR+86, AJ89]. We see our own work as simply another chapter in this long history. Let us begin with a discussion of the history of continuations.
9.2 Continuations, Evaluation, and CPS Compilation

9.2.1 Continuations Throughout History

In Programming Languages

One of the major directions in the study of programming language semantics has been the denotational semantics approach [MS76] pioneered by Dana Scott. After a relatively short time, the standard method of specifying a program's semantics became the method of continuation semantics (called "standard semantics", in the literature). A continuation in denotational semantics is a value that needs to be applied to (usually) a value and/or store, and returns a final answer, the result of the entire computation. Program fragments are viewed as continuation transformers, which take continuations as arguments and return other continuations. Within this framework, one can define the mathematical semantics of nonfunctional languages by specifying that a nonlocal goto \( l \), for example, does not use the continuation supplied to it, but instead uses a continuation stored under the label \( l \).

It is sometimes the case that theory inspires, rather than explains practice. Out of the work in continuation semantics came the novel idea that perhaps allowing the programmer to explicitly access the current continuation would be a useful programming feature. To achieve this effect, the nonfunctional control operator \( C \) was invented. Its evaluation semantics can be expressed via one other nonfunctional control operator, \( A \), and the two evaluation rules:

\[
E[ZM] \rightarrow_1 M(\lambda x. A[E[x]]) \\
E[AM] \rightarrow_1 M.
\]

That is, \( C \), when applied to a term \( M \), packages up its current evaluation context in a function, and resumes evaluating \( M \) applied to that function. The effect is to allow \( M \) to do what it wants when it is finished and wants to exit - it can resume execution at the original evaluation context, or resume at another evaluation context if it has others available, etc. \( A \) simply discards away its current evaluation context and proceeds with evaluating its argument in the empty (i.e. top-level) context.

It turns out that many nonfunctional control operations can be expressed with these simple additions to a functional language. Felleisen argues that, in a certain sense, \( C \) and \( A \) are the basic units from which we can build almost all new control structures [Fel87].

In Compilation

Another area in which the idea of continuations has had an impact is compiler technology. The essential ideas in continuation semantics were separated out by [Fis72] into a continuation-passing-style (CPS) translation which, when applied to a program, converted that program to one in which the evaluation order was specified not externally, but explicitly in the program. Plotkin [Plo75] extended this work to show that one could, in a certain sense, simulate a call-by-value programming language in a call-by-name language, and vice-versa. Felleisen [FFED86] extended
the CPS-translation idea to encompass the $\mathcal{C}$ operator, showing that one could translate a functional language with instances of $\mathcal{C}$ into a language without such instances, such that, for programs which computed to ground values, e.g. integers, the original program and the translated program gave equal values. This program translation technology has been integrated into several compiler systems [AJ89, KKR+86, Ste78].

9.3 CPS-Compilation, Semantic Equivalence, and Definitional Interpreters

In this section we will discuss the foundational work of Plotkin and Griffin, who set the stage for our present investigation. Let us begin with a discussion of Plotkin's seminal work on definitional interpreters for the lambda-calculus.

The setting for Plotkin's work was the work of Landin [Lan63], who showed that one could construct a machine which embodied exactly the operational semantics of the call-by-value lambda-calculus (the SECD machine). Plotkin defined a similar SECD machine which captured exactly the operational semantics of the call-by-name lambda-calculus.

The difference between these two machine models, SECD for CBV lambda-calculus and SECD for CBN lambda calculus, was exactly the difference between the two lambda-calculi themselves. Plotkin sought to make this difference apparent by defining a translation from one lambda-calculus to the other, such that the translated term mimicked in a certain sense the operational semantics of the original term.

More precisely, given a language $\mathcal{L}$, the object-language, and a language $\mathcal{M}$, the meta-language, each of which is equipped with a deterministic evaluator, Plotkin sought a translation, $\bullet : \mathcal{L} \to \mathcal{M}$, such that for any term $P$ from $\mathcal{L}$, if $P \to b$, then $P \to b'$, where $b'$ is a value analogous to $b$ (in a very precise sense).

Plotkin showed that one could could simulate (to a point) a CBN lambda-calculus with $\delta$-reductions in a CBV lambda-calculus with analogous $\delta$-reductions, and that the converse is also true. He showed that for programs which evaluated to primitive values, e.g. integers, the simulation was exact. His simulation was by the method of CPS-compilation. He compiled a lambda-term in the CBV lambda-calculus to a term in the CBN lambda-calculus such that regardless of what evaluation order was chosen for the term, it evaluated to a value analogous to that which the original term evaluated, by the clever device of making his translated term have exactly one binding-free redex at any particular state during evaluation. Hence, any evaluation strategy would pick that redex. Hence, every evaluation strategy would follow the same course.

Plotkin refers to such a translation from $\mathcal{L}$ to $\mathcal{M}$ as providing a definitional interpreter for a programming language $\mathcal{L}$ in a programming language $\mathcal{M}$. That is, such a translation provides a way of understanding programs in $\mathcal{L}$ without understanding
the evaluation order on programs in $\mathcal{M}$.

Felleisen [Fel87] showed that, given a program $M$ in $\lambda^v + C$, if $M$ evaluated to an integer $b$, then so did its CPS-translation $\overline{M}$ in $\lambda^u$. Since the evaluation sequence travelled by $M$ is independent of order of evaluation, it follows that in any lambda calculus, including the call-by-name lambda-calculus, the same is true.

Griffin observed that, in the typed setting, one could assign a type to the operator control ($C$), described earlier, of $\neg\neg(P) \Rightarrow P$, for any type $P$. He then observed that call-by-value CPS-translation translated a program in typed $\lambda^v + C$ to the typed lambda-calculus, in such a way that the original program computed to an integer $b$ if and only if the translated program also computed to the same integer. Moreover, the translation upon types that is induced by the translation upon programs is very much like a double-negation translation.

Thus, he was able to conclude that there was a sensible operational semantics for the original program in which the original program would terminate in a value of an appropriate type. His result trivially implied that, when the type of the program was primitive, e.g. $T = \text{Int}$, one could apply the CPS-translated program to the empty continuation, $\lambda x.x$, and recover the same value that the original program evaluated to. That is, the two programs were semantically equivalent. This result tells us that CPS-compilation of programs with $C$ preserves their semantics. In fact one can look at a Griffin’s result as the specialization of Friedman’s conservative extension result to implicational logic, with a modified Kuroda translation (more on that later) as his double-negation translation.

Unfortunately, Griffin’s work was done in the absence of the rich history of A-translation and its connections with constructivity, as discovered by Friedman. Hence, he was not able to extend his discovery to total-correctness type theories in a way which would allow him to give a semantics of evidence to classical proofs.

We greeted Griffin’s work (in February of 1990) with tremendous excitement. At that time, we had been working on the problem of understanding the method by which the Gödel/Friedman translation injected a classical (putatively noncomputational) proof with computational content. We immediately saw that Griffin’s fundamental observation, that the $C$ operator could be given a consistent typing in classical logic, explained completely the Gödel/Friedman translation. As we shall see, the Gödel/Friedman translation consists of exactly a CPS-translation upon the “classical witness” for a classical proof.

## 9.4 Connections With Our Work

Much of the history of computer science has been an attempt, on the one hand, to rationalize mathematically the ad hoc principles that programmers use every day, and, on the other, to discover what meaning various techniques, tricks, and systems from the world of mathematics have when interpreted from the viewpoint of computing.
The entire enterprise of constructivism can be interpreted as an attempt to formalize and inject rigor into the informal art of programming. The formalization of induction, and its relation with bounded iteration, is a good example of this phenomenon, as is the relationship between infinite streams and Brouwerian choice sequences. Likewise with the connection between existential propositions and program specifications, and between universally quantified propositions and polymorphism.

Our work can be viewed in this tradition also, as the beginnings of an attempt to connect classical proof methods with programming. Our demonstration in these two chapters that classical reasoning is exactly the use of the call-with-current-continuation operator is simply the beginning of a course of investigation in how to merge the practice of programming with continuations with what is known about classical logic. The task of programming with continuations is troublesome and error-prone, because continuations are as error-prone as unrestricted goto’s. Nevertheless, we can prove that programs with continuations meet their specifications exactly via the machinery of classical logic.

Thus, our work provides for programming language theorists and designers the beginnings of a rational and correct basis for typing and reasoning about explicit continuations. As we shall see, when we add explicit continuations to our functional language, we are forced to specify that certain programs which were previously allowed to be evaluated in either call-by-name or call-by-value evaluation order must now be evaluated in call-by-value order. We discover this only when we attempt to translate these programs in such a way as to preserve a call-by-name evaluation order, and fail. Hence, our work allows programming language designers who care about total correctness to reason about the manner in which features of their languages must be presented in order to maintain total correctness.

For constructivists and logicians, our work provides a long-missing high-level understanding of the “sometimes-constructive” content of classical proofs, and a very clear and understandable “semantic” (i.e. program/evaluation-oriented) rather than syntactic (proof-translation oriented) account of the process of double-negation/A-translation. Through our work, we feel that those interested in the constructive content of classical proofs, and even those interested simply in classical proofs and what sorts of idealized computations they represent, will find new insight into the ideas behind tricky, seemingly opaque proofs.

Our work explains the foundational results of Gödel and Friedman in a new way, shedding more light on the fundamental connections between logic and computer science. It also explains the programming language features derivable from the C operator in a new way, pointing the way to a new understanding of these control structures.
9.5 A Tractable Example Theory

We begin with a simple example theory, and its CPS-translation. This example is a re-working of Griffin’s paper, from the standpoint of the Gödel/Friedman translation. The result we will prove is that, given a classical proof, a modified Kolmogorov-translation on formulas, accompanied by a compatible Kolmogorov-translation upon that proof, and followed by a compatible A-translation, is identical in effect to the process of call-by-name CPS-compilation \(^1\) upon the “classical witness” for that proof. Moreover, the translated program (the witness extracted from the translated proof) mimicks the evaluation semantics of the original program, and they are semantically equivalent. We make all of this quite precise in this section.

9.5.1 The Programming Language

Programs in our language will consist of closed terms drawn from the set \(\text{Term}\), defined inductively as

\[
\text{Term} \equiv \text{Var}_1 \\
| \quad \text{Term}_1(\text{Term}_2) \\
| \quad \lambda \text{Var}_1.\text{Term}_1 \\
| \quad n \text{ (where } n \text{ is a numeral).}
\]

We must define an operational semantics for this programming language. We do so using the notion of evaluation contexts [Fel87]. We will write \(E[M] \rightarrow_1 E[N]\) if \(M\) in the evaluation context \(E[\bullet]\) evaluates to \(N\) in the same evaluation context in one reduction step. We can think of such a rule as specifying a rewrite for a term-rewriting lambda-calculus evaluator. Using this formalism, the operational semantics of our language is captured by the single rule of \(\beta\)-reduction:

\[
E[(\lambda x.b)(N)] \rightarrow_1 E[b[N/x]].
\]

Note that we did not place any restrictions upon the form of \(N\), such as a restriction that \(N\) had to be a value (e.g. a lambda-term or integer). Thus, our programming language has lazy semantics, as does Nuprl. We must define a deterministic interpreter for our language, so we can prove our semantic equivalence result. We do so by defining the evaluation contexts, which are terms with a “hole”, which is filled in with a redex. The valid evaluation contexts for our language are:

\[
E \equiv [ ] | E(N)
\]

The idea is that for a redex at top level, the empty evaluation context is valid, and so is a redex at any depth of applications, as long as it is on the left-hand-branch of the applications. This condition makes our evaluator call-by-name.

\(^1\)Griffin [Gri90] did this work for call-by-value CPS-translation, but he was unaware of Friedman’s work, so it appears \textit{ad hoc}
9.5.2 The Constructive Logic

We will work with pure implicational logic over the integers. That is, we will have exactly two base types, the integers and ⊥ (for falsehood). We will admit as proofs of \( \text{Int} \) every numeral, e.g. 1. Of course, there are no proofs of \( ⊥ \). The terms of the logic are simply the implicational formulas constructed with \( \bullet \Rightarrow \bullet \) from the basic types \( \text{Int} \) and \( ⊥ \).

We will now define, for constructive (actually, \textit{minimal}) implicational logic (called \( LJ \Rightarrow \)) \textsuperscript{2} with sole prime formula \( \text{Int} \), the proof rules and constructive witnesses.

\textbf{Hypothesis}

\[ H, x:A \gg A \ \text{ext} \ x \]

\textbf{Cut}

\[ H \gg C \ \text{ext} \ M(N) \]
\[ \text{BY cut } B \]
\[ \gg B \ \text{ext} \ N \]
\[ \gg B \Rightarrow C \ \text{ext} \ M \]

\textbf{Implication Introduction}

\[ H \gg A \Rightarrow B \ \text{ext} \ \lambda \ x.b \]
\[ \text{BY intro} \]
\[ x:A \gg B \ \text{ext} \ b \]

\textbf{Implication Elimination (Redundant)}

\[ H, f:A \Rightarrow B \gg B \ \text{ext} \ f(M) \]
\[ \text{BY elim } f \]
\[ \gg A \ \text{ext} \ M \]

\textbf{Explicit Intro}

\[ H \gg \text{Int} \ \text{ext} \ n \]
\[ \text{BY explicit intro } n \]

Note that we have purposely left out any mention of \( ⊥ \) above.

9.5.3 Extending to Classical Logic

To make the above logic classical, we must add one rule, that of double-negation elimination. We will call this logic \( LK \Rightarrow \).

\textsuperscript{2}Gentzen’s terminology for intuitionistic implicational propositional logic
Double Negation Elimination

\[ H \gg P \text{ ext } CM \]
\[ \text{BY double negation elim} \]
\[ \gg \neg\neg(P) \text{ ext } M \]

After a moment’s reflection, the reader will see that the rule of \( \bot \)-elimination is unnecessary in \( LK \rightarrow \), as every instance of it can be proven in the system already described. The term \( CM \) is the control operator, as described before, with the previously defined operational semantics.

### 9.5.4 Some Simple Results

The constructive system \( (LJ \rightarrow) \) we just presented has the desirable property of “cut-elimination”.

**Theorem 9.5.1 (Cut-Elimination for \( LJ \rightarrow \))** Every proof in \( LJ \rightarrow \) of Int (the only prime formula which is provable) can be converted into a proof in a normal form consisting only of introductions (i.e. an explicit intro of a numeral). Moreover, the normalization procedure corresponds to \( \beta \)-reduction upon the constructive witness, and the constructive witnesses are strongly normalizing under these reductions.

**Proof:** From Gentzen, presented in Prawitz [Pra71]. □

Another important fact about \( LJ \rightarrow \) is that it is a minimal logic.

**Fact 9.5.1 (\( LJ \rightarrow \) is a Minimal Logic)** For a formula \( \phi \) with possible occurrences of \( \bot \), if

\[ \vdash_{LJ \rightarrow} \phi \text{ ext } M \]

then

\[ \vdash_{LJ \rightarrow} \phi[A/\bot] \text{ ext } M. \]

That is, if we uniformly replace \( \bot \) with some other proposition, then the original proof is still valid, and the constructive witness is unchanged. This we can easily show from the absence of a \( \bot \)-elimination rule. The reader will recall that in Section 6.2 we discussed the concept of a minimal logic, and its relevance to the A-translation.

### 9.5.5 CPS-Translation and Double-Negation Translation for Constructive Witnesses

We begin an explication of the CPS-translation by concentrating first upon translations of constructive witnesses, showing our main result, and then extending the machinery to classical witnesses. The CPS-translation upon terms is particularly simple (and has been around for over fifteen years [Fis72,Plo75]):
\[
\begin{align*}
\equiv & \equiv x \\
MN & \equiv \lambda \, k.\, M(\lambda \, m.\, (N)k) \\
\lambda \, x.\, M & \equiv \lambda \, k.\, k(\lambda \, x.\, M) \\
n & \equiv \lambda \, k.\, k(n) \enspace (\text{where } n \text{ is a numeral})
\end{align*}
\]

The intuitive idea behind this translation is to integrate the program being evaluated with the mechanism which chooses redices for contraction. That is, when we are presented with an application term, the translation tells us that we must compute the operator (left-part) first, to a value, assign that value to \( m \), and then apply that value to the translated expression for the operand (right-part) \( N \), and the current \textit{continuation}, \( k \), which is the variable which holds the “rest of the computation”.

We can now define a compatible CPS-translation upon types:

\textbf{Definition 9.5.1 (CPS-Translation on Types for } \textit{LJ} \rightarrow \textit{)} Given a program \( M \) of type \( \phi \), define the type \( \overline{T} \) by structural induction as follows (let \( \overline{\tau}(X) \equiv X \Rightarrow \phi \)):

\[
\begin{align*}
\overline{\text{Int}} & \equiv \overline{\tau}(\text{Int}) \\
\overline{U \Rightarrow V} & \equiv \overline{\tau}(\overline{U} \Rightarrow \overline{V}) \\
\overline{\bot} & \equiv \overline{\phi}
\end{align*}
\]

Notice how much this translation resembles the combination of the Kolmogorov translation and A-translation. To be exact, we can factor this translation into three stages:

1. Replace every instance of \( \bot \) with \( \phi \).
2. Double-negation translate via the standard Kolmogorov translation.
3. Replace every instance of \( \bot \) with \( \phi \), again.

The last two stages, obviously, are exactly a standard Kolmogorov/A-translation. The preliminary A-translation does not change constructive content, since, as we noted before, \( \textit{LJ} \rightarrow \) is a minimal logic. It is relatively easy to show

\textbf{Lemma 9.5.1.} If \( \phi \), the type of our program, contains no instances of \( \bot \), then for any subexpression \( N \) of type \( T \), \( \overline{N} \) has type \( \overline{T} \).

\textit{Proof:} By structural induction on \( N \), with the additional assumption that for every free variable \( x : T' \) in \( N \), \( x \) has type \( \overline{T'} \) in \( N \).

By cases on the form of \( N \):

\( N = \lambda \, x.\, M \): Then \( T = A \Rightarrow B \), so under the assumption \( x \) has type \( \overline{A} \), \( M \) has type \( \overline{B} \). Hence \( \lambda \, x.\, M \) has type \( \overline{A} \Rightarrow \overline{B} \), and \( \lambda \, x.\, M \) has type \( \overline{A} \Rightarrow \overline{B} \).
9.5 A Tractable Example Theory

\[ N = x \text{ a variable: } \text{Let } x : A. \text{ Then since } x \text{ is free, it is given type } \overline{A} \text{ by assumption.} \]

\[ N = M_1(M_2) \text{ an application: } \text{Suppose } M_1 \in A \Rightarrow B, M_2 \in A. \text{ Then by assumption, } M_1 \in \overline{A} \Rightarrow \overline{B}, \text{ and } M_2 \in \overline{A}. \text{ Then it is easy to see } \lambda k. M_1(\lambda m. m(M_2)k) \in \overline{B}. \]

Just note \( k \in \phi(\overline{A} \Rightarrow \overline{B}), m \in \overline{A} \Rightarrow \overline{B} \).

Note that in the previous proof, we only assumed that \( \bot \) did not occur in \( \phi \), the type of the program itself; no such restriction is put on \( T \), the type of the subexpression whose CPS-translation we were deriving a type for. Nevertheless, the fact that our logic is minimal means that if \( \phi = Int \), then \( \bot \) does not occur in a proof of \( \phi \). Thus, for proofs in \( LJ \rightarrow \), the CPS-translation we described above is exactly a combined Kolmogorov/A-translation. It is now trivial to show:

**Lemma 9.5.2.** If \( \phi = Int \), then from a constructive witness \( M \) for \( Int \) we get a constructive witness \( M \) for the type \( \overline{\phi} \equiv (Int \Rightarrow Int) \Rightarrow Int \).

Together, the above two facts tell us that

\[ \vdash_{LJ} \phi \text{ ext } M \]

implies

\[ \vdash_{LJ} (Int \Rightarrow Int) \Rightarrow Int \text{ ext } M. \]

Finally, we prove that \((M)(\lambda x.x)\) is a witness for \( Int \).

\[ \vdash_{LJ} Int \text{ ext } (M)(\lambda x.x). \]

The term \( \lambda x.x \) is referred to as the top-level continuation of type \( Int \), because, when it is fed as argument to a value of type \((Int \Rightarrow Int) \Rightarrow Int\), the result is a value of type \( Int \).

Notice that we never “in so many words” showed that from a proof of \( \phi \), we can construct a proof of \( \overline{\phi} \). However, the typing argument is equivalent to such a proof, for obvious reasons, since from the constructive witness and its typing we can generate a proof in short order.

Another fact to note at this point is the similarity between the sequence of proofs we generated above, and the sequence of proofs that we generated in the Gödel/Friedman translation; there, we took a classical proof of \( \phi \), generated a constructive proof of \( \overline{\phi} \), and from that generated a constructive proof of \( \phi \). Shortly, we will extend the method above to handle translating (classical) proofs in \( LK \rightarrow \) of \( Int \) into constructive proofs of \( Int \).
9.5.6 Semantic Equivalence for $LJ \rightarrow$ 

At this point, we must show that the two programs, $M$, and $(M)(\lambda x.x)$, are semantically equivalent. We will use a method adapted from Plotkin [Pl75] and Griffin [Gri90], with appropriate modifications for call-by-name evaluation order. Our first attack at this problem would be to try to show that whenever $M \rightarrow_1 N$, $M(\lambda x.x) \rightarrow N(\lambda x.x)$. But this approach is doomed to failure, because the relation simply does not hold. For example, consider the two terms $A = (\lambda x, y.x)(M)(N)$ and $B = (\lambda y.M)(N)$. Clearly, $A \rightarrow_1 B$, but we will show that, under normal-order reduction, their CPS-translations are not so related:

\[
(\lambda x, y.x)(M)(N)I = (\lambda k_1.(\lambda x, y.x)(M)(\lambda m.mNk_1))I \\
= (\lambda k_1.(\lambda k_2.(\lambda x, y.x)(\lambda m.mMk_2))(\lambda m.mNk_1))I \\
= (\lambda k_1.(\lambda k_2.(\lambda k_3.\lambda x, k_3.\lambda x, \lambda k_4.k_4(\lambda y.x))(\lambda m.mMk_2)) \\
(\lambda m.mNk_1))I \\
\rightarrow_1 (\lambda k_2.(\lambda k_3.\lambda x, \lambda k_4.k_4(\lambda y.x))(\lambda m.mMk_2)) \\
(\lambda m.mNI) \\
\rightarrow_1 (\lambda k_3.\lambda x, \lambda k_4.k_4(\lambda y.x))(\lambda m.mM(\lambda m.mNI)) \\
\rightarrow_1 (\lambda m.mM(\lambda m.mNI))(\lambda x, \lambda k_4.k_4(\lambda y.x)) \\
\rightarrow_1 (\lambda x, \lambda k_4.k_4(\lambda y.x))M(\lambda m.mNI) \\
\star \\
\rightarrow_1 (\lambda k_4.k_4(\lambda y.M))(\lambda m.mNI) \\
\rightarrow_1 (\lambda m.mNI)(\lambda y.M) \\
\rightarrow_1 (\lambda y.M)NI \\
\star \\
\rightarrow_1 MNI \\
\rightarrow_1 (\lambda y.M)(N)I \\
= (\lambda k_1.(\lambda y.M)(\lambda m.mNk_1))I \\
\rightarrow_1 (\lambda k_1.(\lambda k_2.\lambda y.M)(\lambda m.mNk_1))I \\
\rightarrow_1 (\lambda k_2.\lambda y.M)(\lambda m.mNI) \\
\rightarrow_1 (\lambda m.mNI)(\lambda y.M) \\
\star \\
\rightarrow_1 (\lambda y.M)NI \\
\rightarrow_1 MNI
\]

What does hold, however, is

**Lemma 9.5.3 (Soundness of CPS)** If $M$ is typed $Int$ in $LJ \rightarrow$, and $M \rightarrow N$, then $M(\lambda x.x) = N(\lambda x.x)$.

**Proof:** Trivial. ■

This lemma tells us that CPS translation is sound with respect to the original semantics. But we cannot prove that $M(\lambda x.x)$ reduces to $N(\lambda x.x)$. Upon investigation of why this approach fails, we find that the reduction sequence traveled by $M(\lambda x.x)$ consists of a series of "administrative" reductions, which do the job of simulating a runtime stack, followed by a "proper" reduction, followed by more administrative reductions, and so on. We marked with an asterisk (*) the reductions above which were "proper"; clearly the administrative reductions dominate.
the computation. Plotkin discovered this, and devised a "colon-translation" which contracts many of the administrative redices, so that, though $M(\lambda x.x) \to N(\lambda x.x)$ does not hold, $M : \lambda x.x \to N : \lambda x.x$ does. Plotkin's colon-translation is defined in terms of a translation upon types, the star-translation, and an accompanying translation upon terms:

**Definition 9.5.2 (Double Underbar Translation)**

$$\underline{x} \equiv x \ (x \ a \ variable)$$

$$\underline{M} \equiv \lambda k.(M : k)$$

**Definition 9.5.3 (Star Translation Upon Values)** Define the function $\Psi$, as follows:

$$\Psi(\lambda x.M) \equiv \lambda x.M$$

$$\Psi(n) \equiv n \ (n \ a \ numeral)$$

**Definition 9.5.4 (Colon Translation)** Given a program of type $Int$, we can define the "colon" translation as follows:

$$x : K \equiv x(K) \ (x \ a \ variable)$$

$$V : K \equiv K(\Psi(V)) \ (when \ V \ is \ a \ value)$$

$$M(N) : K \equiv M : \lambda m.m(N)K$$

The idea behind the colon-translation is to systematically contract redices everywhere it is feasible to do so. We must now prove a series of lemmas about this translation, culminating in the final theorem we want:

**Lemma 9.5.4 (Evaluation Context Unwrapping)**

For all evaluation contexts $E$, all terms $M$, and values $K$

$$E[M] : K = M : K^E$$

where $K^E$ is defined inductively on $E$ as

1. $K^[] = K$,
2. $K^{E_1(N)} = (\lambda m.m(N)K)^{E_1},$

**Proof:** Trivial from the definition of the colon-translation. ■

A trivial substitutivity lemma:

**Lemma 9.5.5 (Substitutivity)** If $N$ is a closed value, and $x$ not free in $K$,

$$M[N/x] \to M[N/x]$$

and

$$(M : K)[N/x] \to M[N/x] : K.$$
Proof: A trivial induction on the size of $M$. ■

The plan for our proof will be to take a term $E[M] : \lambda x.x$ which has a redex, $M$, such that $M \rightarrow_1 N$, use the evaluation context unwrapping lemma to convert it to a term $M : (\lambda x.x)^E$, prove that this reduces to the term $N : (\lambda x.x)^E$, and then use the unwrapping lemma to justify that this term is equal to $E[N] : \lambda x.x$.

Lemma 9.5.6 (Colon Mimics Reduction) If $E[M]$ is a witness for Int, where $M$ is a redex, and $E[M] \rightarrow_1 E[N]$, then

$$E[M] : \lambda x.x \rightarrow E[N] : \lambda x.x.$$  

Proof: Unwrap $E[M] : \lambda x.x \equiv M : \lambda x.x^E$, and prove:

$$(\lambda x.b)(t) : \lambda x.x^E = (\lambda m.m(\bar{t}))\lambda x.x^E$$

$$(\lambda m.m(\bar{t})\lambda x.x^E)(\lambda x.b)$$

$$(\lambda t/x)(\lambda x.x^E)$$

$$(\lambda t/x)(\lambda x.x^E)$$

$$(\lambda t/x) : \lambda x.x^E$$

$$E[N] : \lambda x.x.$$

Finally, we connect colon-translation with CPS-translation:

Lemma 9.5.7 (Soundness of Colon) If $M$ is a term of type $T$, in a program of type $Int$, $K$ is a value, then $MK \rightarrow M : K$, and $M \rightarrow M$.

Proof: By induction on the structure of $M$.

Lambda Terms

$$\lambda x.M \rightarrow (\lambda k.k(\lambda x.M))$$

$$\lambda k.k(\lambda x.M)$$

$$\lambda k.k x.M : k$$

$$\lambda x.M$$

$$\lambda x.MK \rightarrow \lambda x.MK$$

$$\lambda x.M : K$$

Applications

$$M(N) = \lambda k.M(\lambda m.mNk)$$

$$\rightarrow \lambda k.M(\lambda m.mNk)$$

$$\rightarrow \lambda k.M : \lambda m.mNk$$

$$\lambda k.M(N) : k$$

$$= M(N)$$

$$M(N)K = (\lambda k.M(\lambda m.mNk))K$$

$$\rightarrow M(\lambda m.mNK)$$

$$\rightarrow M : \lambda m.mNK$$

$$= M(N) : K$$
Variables
\[
x K = x(K) \\
= x : K \\
x = x
\]

Theorem 9.5.2 (Semantic Equivalence) Given a program \( M \) typed \( \text{Int} \) in \( LJ \rightarrow \), \( M \) evaluates to \( b \in \text{Int} \) if and only if \( M(\lambda \ x. x) \) evaluates to \( b \).

Proof: Case \( \Rightarrow: \) If \( M \) evaluates to \( b \), then we invoke the soundness of CPS-translation, and conclude that \( M(\lambda \ x. x) = b(\lambda \ x. x) \), and since \( b(\lambda \ x. x) \rightarrow b \), we know that \( M(\lambda \ x. x) \rightarrow b \).

Case \( \Leftarrow: \) Invoke the colon-soundness lemma to show \( M(\lambda \ x. x) \rightarrow M : \lambda \ x. x \). Then, using the fact that reductions on colon translations mimick reductions on the original term, argue that there could not be an infinite reduction sequence on \( M \), for it would be translated into an infinite reduction sequence on the translated term. Hence conclude that reduction on \( M \) terminates, and since \( M \) is well-typed, and so are all of its contracta, conclude that \( M \rightarrow b \).

9.5.7 The CPS-Translation for Classical Witnesses

What we just proved is that the Kolmogorov-translation is a value-preserving embedding of \( LJ \rightarrow \) back into \( LJ \rightarrow \). By extending the CPS-translation (and Kolmogorov-translation) to classical witnesses of classical proofs, we can prove that the Kolmogorov-translation is a value-preserving embedding of \( LK \rightarrow \) into \( LJ \rightarrow \). The CPS-translation of \( \mathcal{C} \) (and \( \mathcal{A} \)) is

Definition 9.5.5 (CPS-Translation for \( \mathcal{A} \) and \( \mathcal{C} \)) Given a program \( P \) of classical type \( \phi = \text{Int} \), the CPS-translation of subterms \( CM \) and \( AM \) in \( P \) are:

\[
\begin{align*}
AM & \equiv \lambda k. (M(\lambda \ x. x)) \\
CM & \equiv \lambda k. M(\lambda \ m.m(\lambda \ g.g(\lambda \ v.h.v(k)))(\lambda \ x. x)).
\end{align*}
\]

Lemma 9.5.8 (Typing Soundness of CPS-Translation of \( \mathcal{A} \) and \( \mathcal{C} \))

Given a term \( CM \) of classical type \( T \) in a program of classical type \( \phi \), the term \( CM \) has constructive type \( \overline{T} \).

Proof: By induction on term structure, carrying along the proper assumptions about free variables. Note that inductively, \( M \) has type \( \overline{-}(T) \). Then by the following typing argument, \( CM \) has type \( \overline{T} \):
\[ k : \text{\textbf{\textsf{T}}}^{*} \equiv T^{*} \Rightarrow \phi \]
\[ m : \text{\textbf{\textsf{\neg}}}(T \Rightarrow \bar{\phi}) \Rightarrow \bar{\phi} \]
\[ g : \text{\textbf{\textsf{\neg}}}(T \Rightarrow \bar{\phi}) \]
\[ v : T \]
\[ h : \text{\textbf{\textsf{\neg}}}(\phi) \equiv \phi \Rightarrow \phi \]

The reader might have noticed that \( h \) has the same type as \( \lambda x.x \), the continuation embedded in the translation of \( C \). This is not coincidental. In fact, \( h \) is the current continuation of the form, which is immediately discarded. The function which it is packaged up within is responsible for “throw-ing” its argument \( v \) to the continuation \( k \), and in the process it discards \( h \).

By a similar argument, if \( A M \) has classical type \( \perp \) when \( M \) has classical type \( \phi \), we can conclude that \( A M \) has constructive type \( \overline{\phi} \) when \( M \) has constructive type \( \overline{\phi} \). But this is not necessary for establishing the soundness of the type-translation, as \( A \) only shows up in the operational semantics of \( C \), and not in proofs.

And now we get the first major theorem:

**Theorem 9.5.3. (Conservative Extension of \( LJ \rightarrow \) to \( LK \rightarrow \) for Base Types)**

Given a proof

\[ \vdash LK \rightarrow Int \text{ ext } M, \]

the Kolmogorov translation effectively constructs a proof

\[ \vdash LJ \rightarrow (Int \Rightarrow Int) \Rightarrow Int \text{ ext } M. \]

By the same argument as before, we can conclude that we have a proof

\[ \vdash LJ \rightarrow Int \text{ ext } (M)(\lambda x.x). \]

### 9.5.8 Semantic Equivalence for \( LK \rightarrow \)

Finally, we must extend the semantic equivalence arguments to account for \( A \) and \( C \). We must define the colon-translation of the two new forms we introduced, and prove all the lemmas for the cases involving the two new forms. We will do all of this with very little explanation, as by now the method of proof should be very clear.

**Lemma 9.5.9 (Reduction Preserves Typing)** If

\[ \vdash LK \rightarrow \phi \text{ ext } M \]

and \( M \rightarrow_{1} N \), then

\[ \vdash LK \rightarrow \phi \text{ ext } N \]
Proof: This lemma is not so trivial now, because we need to deal with the $A$ operator in order to show that the reduction rules for $A$ and $C$ preserve typings. But this is relatively easy - we add a rule for $A$:

$$H \gg \bot \text{ ext } AM$$
$$\gg \phi \text{ ext } M$$

where $\phi$ is the top-level type of the program. Then we can prove that reduction preserves typings:

Reduction Rule

$$E[CM] \rightarrow M(\lambda x.A(E[x]))$$

Typings

$$E[CM] : \phi$$
$$CM : T$$
$$M : \neg \neg (T)$$
$$A : \phi \Rightarrow \bot$$
$$\lambda x.A(E[x]) : T \rightarrow \bot$$
$$M(\lambda x.A(E[x])) : \phi$$

Reduction Rule

$$E[AM] \rightarrow M$$

Typings

$$E[AM] : \phi$$
$$AM : \bot$$
$$M : \phi$$

Note that during Kolmogorov-translation, the typing of $A$ will be changed from $\phi \rightarrow \bot$ to $\overline{\phi} \rightarrow \overline{\phi}$.

Definition 9.5.6 (Colon Translation of $C$ and $A$)

$$AM : K \equiv M : \lambda x.x$$
$$CM : K \equiv M : \lambda m.m(\lambda g.g(\lambda v,h.v(K))) \lambda x.x$$

The double-underbar translation is defined from the colon-translation, as before.

Lemma 9.5.10 (Substitutivity) If $N$ is a closed value, and $x$ not free in $K$, then

$$M[N/x] \rightarrow M[N/x]$$

and

$$(M : K)[N/x] \rightarrow M[N/x] : K.$$
Proof: Trivial.

The Colon Characterization Lemma is trivial, from the definition of the colon-translations of $A$ and $C$. The Colon Soundness Lemma is easy also:

**Lemma 9.5.11 (Colon Soundness for $A$ and $C$)**

\[
\begin{align*}
AMK &\rightarrow AM : K \\
CMK &\rightarrow CM : K
\end{align*}
\]

Proof: From the definitions. ■

**Lemma 9.5.12 (Colon Mimics Reduction for $C$ and $A$)** If $M$ is a witness for $\phi$, and $M \rightarrow_1 N$ via the rules of evaluation for $A$ or $C$, then

\[
M : \lambda \, x.x \rightarrow N : \lambda \, x.x.
\]

Proof:

**$A$:**

\[
E[AM] : \lambda \, x.x = AM : \lambda \, x.x^E = M : \lambda \, x.x
\]

**$C$:**

\[
\begin{align*}
E[CM] : \lambda \, x.x &= CM : \lambda \, x.x^E \\
&= M : \lambda \, m.m(\lambda \, g.g(\lambda \, v, h.v(\lambda \, x.x^E)))\lambda \, x.x \\
&= M : \lambda \, m.m(\lambda \, g.g(\lambda \, v, h.v : \lambda \, x.x^E))\lambda \, x.x \\
&= M : \lambda \, m.m(\lambda \, g.g(\lambda \, v, h.E[v]) : \lambda \, x.x)\lambda \, x.x \\
&= M : \lambda \, m.m(\lambda \, g.g(\lambda \, v, h.A(E[v])) : h))\lambda \, x.x \\
&= M : \lambda \, m.m(\lambda \, g.g(\lambda \, v.A(E[v])))\lambda \, x.x \\
&= M : \lambda \, m.m(\lambda \, g.g(\Psi(\lambda \, v.A(E[v])))\lambda \, x.x \\
&= M : \lambda \, m.m(\lambda \, g.g(A(E[v]) : g))\lambda \, x.x \\
&= M : \lambda \, m.m(\lambda \, v.A(E[v]))\lambda \, x.x \\
&= M(\lambda \, v.A(E[v])) : \lambda \, x.x
\end{align*}
\]

Finally, the main theorem:

**Theorem 9.5.4 (Value-Preserving Conservative Extension)** The Kolmogorov translation is a value-preserving conservative extension of $LJ \rightarrow$ to $LK \rightarrow$ for $Int$.

Proof: By analogy with that for $LJ \rightarrow$. ■
9.5.9 Summing Up This Trivial Example: A Breather

We must now re-do this work, in more detail, for the programming language and
logic of arithmetic, showing that the Kolmogorov translation is a value-preserving
conservative extension of HA to PA for $\Pi_2^0$ sentences.

There are several major parts of this work that make sense only when considered
together. Since we must consider them each in turn, we will discuss them here, so
that the reader will (hopefully) not get too lost.

The first two steps in the long path ahead are relatively simple. They are to
define a programming language and logic for arithmetic. We will take Heyting
Arithmetic as our constructive logic, and our programming language will be the
constructive witnesses of Nuprl. However, we must make some allowances for the
peculiarities of primitive recursive function definitions in order to avoid being caught
up in insignificant details. For example, we can prove the proposition $5 \times 4 = 4 \times 5$
by either decomposing multiplication into addition via its defining equations, or by
simply taking all such equalities as given. We will do the latter, and assume given
as axioms every equality of the above sort which is true. We will not talk about each
of these axioms separately, but simply refer to them all as the axiom of equality.

In the preceding example, we treated $Int$ as a proposition. We did this basically
because we had no other propositions available. In the following discussion, though,$Int$ will be a data-type, and not a proposition, in the same manner as (and for the
same reasons as) in Chapter 7. Thus, we will not allow double-negation elimination
upon integers. We will be forced to maintain a distinction between propositions
and types, and we will see this distinction show up in the fact that $Int$ is not a
proposition, unlike the preceding example.

Another puzzling concept that we have only barely discussed is that of a top-level
continuation. As noted before, the idea behind a CPS-translation is to integrate the
order of evaluation into the actual structure of the term to be evaluated, in such
a way that under any evaluation order the term computes in the same manner.
CPS-translation accomplishes this by converting each term into a function which
does not "return" a value at all. Instead, the function expects to be applied to a
continuation, which it applies to the value that it itself computes. For instance, the
CPS-translated term $\lambda \, k.k(1)$ is an implicit integer, in that, given any continuation,
this term will apply that continuation to the integer 1.

As we discussed before, the top-level continuation of type $Int$ is the function
$\lambda \, x.x$ of type $Int \to Int$. In fact, for any primitive type which is a proposition and
admits double-negation elimination, the top-level continuation of that type is just
$\lambda \, x.x$. In the following, though, we will need to consider top-level continuations for
composite types. For instance, a classical proof of $\phi \equiv \exists x \in Int.f(x) = 0$,

$$\Gamma \vdash_K \exists x \in Int.f(x) = 0$$

will be translated to the constructive proof

$$\Gamma \vdash_J \phi \phi(\exists x \in Int.\neg\neg(f(x) = 0))$$.
We will apply the constructive witness above to a term of type
\[
\neg (\exists x \in \text{Int}. \neg \neg (f(x) = 0)),
\]
the top-level continuation of type \(\phi^3\), and end up with a term of type
\[
\phi \equiv \exists x \in \text{Int}. f(x) = 0.
\]
The top-level continuation above will be the term
\[
\lambda \text{p.spread}(p; n, r. \text{r}(\lambda m. (n, m))).
\]
At first glance, all of this is a bit confusing. But if we assume that in order to force an expression to yield a value, we must supply it with a continuation, then the above continuation, when applied to an expression which is an implicit pair, will force the pair to a value, taking apart the value into two parts, \(n\) and \(r\). \(n\) is used as-is, which means that we assume \(n\) is not an implicit value, but an explicit value. \(r\), though, is supplied with a continuation which computes the value of \(r\), binds it to \(m\), and returns the pair \((n, m)\). So we can see that this term above is a continuation which takes an implicit pair consisting of an explicit integer and an implicit value of some primitive type, and computes the explicit representation of that implicit composite object.

That said, let us begin our investigation.

### 9.6 Constructive Arithmetic: A Programming Language and its Logic

In this section we present the programming language and logic of constructive arithmetic. The programming language is a relative of Gödel's \(T\) \cite{Ste72}, but is extended with programs as witnesses of equalities. The logic is a subset of Heyting Arithmetic. As in Chapter 7, we must distinguish between terms which inhabit propositions and terms which inhabit datatypes. Since the only data-type is \(N\), we can simply label all terms of type integer with \((\cdot)^D\). We will not do this in presenting the programming language below, but will label terms in the presentation of the rules when they are integer expressions. A crucial feature that makes this simple presentation possible is the absence of predicate symbols and universal quantification over predicates (though this can be taken care of also). In the rest of this chapter, the syntactic class \(Exp\) below will be called \(Exp_J\), to emphasize the fact that it is intended as the domain from which we draw witnesses for intuitionistic proofs. The syntactic subclass of \(Exp_J\) whose terms are all closed will be called \(Prog_J\), for the same reasons.

\(^3\)The connections between this step and Friedman's translation strategy are detailed in subsection 9.9.4
9.6 Constructive Arithmetic: A Programming Language and its Logic

9.6.1 The Programming Language

The syntactic classes of our programming language are $Exp$ and $Var$. The equations are:

\[
Exp \equiv Var_1 \\
| \lambda Var_1.Exp_1 \\
| Exp_1(Exp_2) \\
| 0 \\
| S(Exp_1) \\
| Exp_1 + Exp_2 \\
| Exp_1 \times Exp_2 \\
| ind(Exp_1; Exp_2; Var_1, Var_2.Exp_3) \\
| (Exp_1, Exp_2) \\
| spread(Exp_1; Var_1, Var_2.Exp_2) \\
| inl(Exp_1) \\
| inr(Exp_1) \\
| decide(Exp_1; Var_1.Exp_2; Var_2.Exp_3) \\
| axiom
\]

Several of the terms above bear further explanation, as they were not described in full detail previously. The term $(M, N)$ is the pairing constructor. The term $spread(Exp_1; Var_1, Var_2.Exp_2)$ is an ugly form of:

\[
let (Var_1, Var_2) = Exp_1 in Exp_2;
\]

that is, it is a form which takes apart pairs of values. We prefer the former because it simplifies the parsing of expressions. The evaluation rule is simply $spread((M, N); u, v. b) \succ b[M, N/u, v]$. In the same manner, $inl(M)$ is the left injection into a disjoint union type, and and $inr(N)$ is the right injection. The term $decide(Exp_1; Var_1.Exp_2; Var_2.Exp_3)$ is a form for analyzing such injections, with two computation rules:

\[
\text{decide}(inl(M); u. b; v. b') \succ b[M/u]
\]

and

\[
\text{decide}(inr(N); u. b; v. b') \succ b'[N/v].
\]

The $ind$ form above is basically identical to that for Nuprl, but it omits the “downcase” code; it is defined only for non-negative integers (but all natural numbers are non-negative).

It should be abundantly clear that this programming language is a trivial modification of a subclass of the terms of Nuprl, and as such, we would expect that any term well-formed in this language would also be in Nuprl.
An Evaluation Semantics

In this section, we discuss an evaluation semantics for our programming language. Our language is lazy, so our evaluation semantics will reduce terms in normal-order. We will describe a set of evaluation rules for programs and an evaluator for our language. Intuitively, we try to describe syntactically the form of a value. Then, we would have to show that if a term is not a value, it has a redex, which our evaluator will contract. For our purposes, though, we will not show this, but merely assert it. A proof would be simply too low a level of detail for even this thesis.

The set of values in our programming language is defined recursively:

\[ Val \equiv \begin{array}{c}
\text{Exp}^D_1 \\
\lambda x. \text{Exp}_1 \\
\langle \text{Val}_1, \text{Val}_2 \rangle \\
\text{inl}(\text{Val}_1) \\
\text{inr}(\text{Val}_1) \\
\text{axiom}
\end{array} \]

All the cases above are obvious except for the first two. The first case tells us that a term marked as data is automatically a value. This means that we treat an expression 5 * 7 as equivalent to the expression 35 for purposes of evaluation. Likewise, when we introduce primitive-recursive function symbols, we will treat primitive recursive applications as values. We are choosing to ignore the evaluation semantics of primitive-recursive function symbols, since they are all total and evaluated to integers, in favor of concentrating upon the evaluation semantics of structured data. This conscious "sweeping under the rug" of the evaluation semantics of integer expressions is exactly what the Kolmogorov translation does, since it does not alter data-types such as \( \mathbb{N} \), nor inhabitants such as \( 2^{10} \). The second case tells us that a lambda-term is a value regardless of whether its body is a value. This expresses the nature of the weak-head-normal-form (whnf) characteristic of closures in programming languages.

After Felleisen [Fel87], we now define what we mean by a reducible term (what Felleisen calls an evaluation context\footnote{Contrast this with the standard definition of evaluation in Nuprl, where we do not specify evaluation order; Nuprl does not have a unique, deterministic evaluator}). We define it in terms of three different syntactic classes: \( R \), redices, which are terms available for computation, \( RC \), reducible constructors, and \( RD \), reducible destructors. The idea will be that \( R \) denotes terms which are redices at their top-level, e.g. \( \beta \)-redices, \( RC \) will denote constructors which contain as subterms reducible terms, and \( RD \) will denote destructors which contain reducible subterms. Then the class of reducible terms will be any term in \( RC \). We define \( R \) to contain the left-hand-sides of each of the reduction rules below:
9.6 Constructive Arithmetic: A Programming Language and its Logic

\[
\begin{align*}
\text{ind}(0; x, y.d; b; x, y.u) & \rightarrow_1 b \\
\text{ind}(k_n; b; x, y.u) & \rightarrow_1 u[k_n, \text{ind}(k_{n-1}; b; x, y.u)/x, y] \quad \text{if } n > 0 \\
\text{decide}(\text{inl}(t); x.l; y.r) & \rightarrow_1 l[t/x] \\
\text{decide}(\text{inr}(t); x.l; y.r) & \rightarrow_1 r[t/y] \\
\text{spread}(<t_1, t_2>; x, y.t) & \rightarrow_1 t[t_1, t_2/x, y] \\
(\lambda \ x.b)(t) & \rightarrow_1 b[t/x]
\end{align*}
\]

Figure 9.1: The Reduction Rules For Arithmetic Programs

The classes \( RC \) and \( RD \) are now defined recursively:

\[
\begin{align*}
\text{RC} & \equiv \langle RC_1, Exp_1 \rangle \\
& \mid \langle \text{Val}_1, RC_1 \rangle \\
& \mid \text{inl}(RC_1) \\
& \mid \text{inr}(RC_1) \\
& \mid RD
\end{align*}
\]

\[
\begin{align*}
\text{RD} & \equiv RD_1(Exp_1) \\
& \mid \text{spread}(RD_1; \text{Var}_1, \text{Var}_2. Exp_2) \\
& \mid \text{decide}(RD_1; \text{Var}_1. Exp_2; \text{Var}_2. Exp_3) \\
& \mid [\ ] \text{(the empty context)}
\end{align*}
\]

Definition 9.6.1 (Reducible Term) A reducible term is one which contains a unique (possibly empty) binding-free path from the root of the term to a subterm which is a redex, such that this path consists of constructors, followed by destructors, terminating in the redex.

We could relax this definition a little by allowing there to be several different paths of the sort specified above. If we did, then we would have different evaluation semantics available. For example, we could add the following rule for \( RC \):

\[
\text{RC} \equiv \langle Exp_1, RC_1 \rangle
\]

which would tell us that, given a pair-construct, we can reduce the right component even if the left component is not a value. Since Heyting Arithmetic (and also the programming language we are dealing with) is Church-Rosser, we know that reducing the right component instead of the left component in the term \((M(N), A(B))\) could not change the normal form of the term. It will turn out, though, that when we add \( C \), we must use the strict definition of reducible term in order to guarantee confluence.
The reason is simple: adding $\mathcal{C}$ to a lambda-calculus which does not have a deterministic evaluator pre-specified renders that calculus non-confluent. Only by specifying a deterministic order of evaluation can we regain confluence (which then becomes a meaningless concept).

We would like to be able to prove that every expression in our programming language is either a value or a reducible term. However, this isn’t true (consider the term $1(2)$, which is neither). What we need in order to rule out such counter-examples is well-typed-ness of our expressions. Gentzen [Pra71] proved

**Fact 9.6.1.** Every expression which is a constructive witness for a sentence in Heyting Arithmetic is either a value or a reducible term.

That is, well-typed terms are either values or are reducible.

**A Note about Integer Expressions**

The reader might have noted that in our definition of $RD$, we did not include a clause for

$$RD \equiv \text{ind}(RD_1; Exp_2; Var_1, Var_2; Exp_3).$$

Such a clause would have told us that an induction form (a bounded iteration loop) is reducible when its first argument is reducible. However, since we implicitly assume that the first argument is an expression marked $(\bullet)_D$, and that a redex is always on a binding-free path from the root of the term, this means that the free variables in that first argument will all be replaced by numerals. Hence, the expression itself is computable to a numeral, and we will regard the integer expression as equivalent to the numeral it denotes in all contexts. This is a choice we made when we defined an operational semantics to “fit” the Kolmogorov translation; as we will see in the next chapter, there are other translations which free us from this commitment, and allow us to explicitly talk about integer expression evaluation.

As mentioned before, the Nuprl semantics do not specify an evaluation order. Thus, the Nuprl evaluator can be nondeterministic if we so wish, and still have the Church-Rosser property. Of course, Nuprl does not have the strong normalization property, so only one evaluator is guaranteed to terminate - the head-normal evaluation strategy - which we just spelled out.

**9.6.2 The Logic of Arithmetic in Nuprl**

In this section, we reformulate Heyting Arithmetic (HA) as a theory that

1. is a minimal logic
2. can be extended in a meaningful way to PA
3. the extension to PA admits double-negation translation back into HA.
We define

- the syntax of HA formulas
- the proof rules of HA
- the procedure for computing witnesses for each rule of HA.

The syntax of HA is a subset of the syntax of Nuprl. We describe it via three syntactic classes, Term, Formula, and Dom. Intuitively, Formula is the class of all well-formed propositions, that is, objects of proof. Term is the class of all well-formed integer expressions, and Dom is the class of all quantifiable types. In this instance, the only member of Dom is N. We specifically distinguish Dom from Formula here, as we did in Chapter 7 in order to make the logic amenable to double-negation translation. Let us now define the syntax of HA:

\[
\text{Formula} \equiv \begin{array}{l}
\text{Term}_1 = \text{Term}_2 \\
\text{Formula}_1 \land \text{Formula}_2 \\
\text{Formula}_1 \Rightarrow \text{Formula}_2 \\
\text{Formula}_1 \lor \text{Formula}_2 \\
\exists \text{Var}_1 \in \text{Dom}_1. \text{Formula}_1 (\text{Var}_1 \text{ free in } \text{Formula}_1) \\
\forall \text{Var}_1 \in \text{Dom}_1. \text{Formula}_1 (\text{Var}_1 \text{ free in } \text{Formula}_1) \\
\bot
\end{array}
\]

\[
\text{Term} \equiv \begin{array}{l}
\text{Var}_1 \\
F(\text{Term}_1, \cdots, \text{Term}_n) \\
\text{where } F \text{ is a primitive recursive function symbol} \\
S(\text{Term}_1) \text{ (S is the successor function)} \\
0
\end{array}
\]

\[
\text{Dom} \equiv \text{Int}
\]

\[
\text{Var} \equiv x, y, z, \cdots
\]

We have not defined a syntax for primitive recursive functions, but this is not important for presenting the main result. In any case, the extension of these results to arbitrary primitive recursive functions is trivial. For now, it will suffice to have available only S, +, and \times.

### 9.6.3 The Proof Rules of HA

We will now give the refinement calculus for HA. The calculus is presented in sequent style, with the extraction (witness) explicitly included. Of course, the rules are set up so that the witness can be automatically extracted from the proof tree, and so that, given a proof \( \vdash_{HA} \Phi \ ext p \), the Nuprl semantics will justify \( p \in \Phi \). We will
not prove that the putative witnesses given by the proof rules are actually witnesses; the correspondence with the rules of Nuprl will establish that. In the following presentation, since there is only one data-type, we will explicitly mark a member of \( \mathbb{N} \) with \((-)^D\). Experienced readers may skip this section, as it will be elementary. Inexperienced readers will be tempted to skip this section, and probably can, as the rules are included here only for those who wish a very technical understanding of this work.

**Hypothesis and Cut**

The rule of hypothesis looks like:

\[
H, x : T \Rightarrow T \text{ ext } x
\]

and the “cut” rules are:

\[
H \Rightarrow T \text{ ext } M(N)
\]

BY implication cut \( A \)

\[
\Rightarrow A \text{ ext } N
\]

\[
\Rightarrow A \Rightarrow T \text{ ext } M
\]

\[
H \Rightarrow T[N^D/x] \text{ ext } M(N^D)
\]

BY universal cut \( \forall x \in \mathbb{N}.T \) at \( N^D \)

\[
\Rightarrow \forall x \in \mathbb{N}.T \text{ ext } M
\]

As before, \( T \) and \( A \) must not be \( \mathbb{N} \), so that the proof may be amenable to double-negation translation.

**Equality**

There is only one canonical witness for an equality: \textit{axiom}. Of course, this witness may be computed by beta-reduction, etc, as in Nuprl. With this in mind, let us set forth the axioms of equality:

**The Rule of Equality**

\[
H \Rightarrow n = m \text{ ext axiom}
\]

BY equality (when \( n \) and \( m \) denote the same numeral)

**Reflexivity**

\[
\Rightarrow \forall n \in \mathbb{N}.n = n \text{ ext } \lambda n^D.\text{axiom}
\]

**Symmetry**

\[
n : \mathbb{N}, \ m : \mathbb{N}, \ u : n = m \Rightarrow m = n \text{ ext } \text{axiom}
\]
Transitivity

\(a, b, c : N, \ u : a = b, \ v : b = c \gg a = c\) ext axiom

Note that any proof of an equality with no free variables can automatically be a single use of the rule of equality, and that this is the normal form of such a proof.

Existential Quantification

There are just two rules for existential types: introduction and elimination.

\[H \gg \exists x^D \in N. \Phi \ ext \ \langle A^D, B \rangle\]
\[\text{BY intro } A^D\]
\[\gg A^D \in N\]
\[\gg \Phi[A^D/x^D] \ ext \ B\]

\[H, w : \exists x^D \in N. \Phi \gg T \ ext \ spread(w; u, v.M)\]
\[\text{BY elim } w \ new \ u, v\]
\[u^D : N, v : \Phi[u^D/x^D] \gg T \ ext \ M\]

Conjunction

The rules for conjunction are much like those for existential quantification, except that the first component of the pair is produced by proof, and not explicitly provided. Thus the first component, like the second, inhabits a proposition.

\[H \gg A \land B \ ext \ \langle M, N \rangle\]
\[\text{BY intro}\]
\[\gg A \ ext \ M\]
\[\gg B \ ext \ N\]

\[H, x : A \land B \gg T \ ext \ spread(x; u, v.M)\]
\[\text{BY elim } x \ new \ u, v\]
\[u : A, v : B \gg T \ ext \ M\]

Universal Quantification

\[H \gg \forall n^D \in N. \Phi \ ext \ \lambda x^D.M\]
\[x^D : N \gg \Phi[x^D/n^D] \ ext \ M\]

\[H, f : \forall n^D \in N. \Phi \gg T \ ext \ (\lambda v.M)(f(t^D))\]
\[\text{BY elim } f \ on \ t^D \ new \ v\]
\[\gg t^D \in N\]
\[v : \Phi[t^D/n^D] \gg T \ ext \ M\]

The elimination rule above is unnecessary, as the "universal-cut" rule subsumes it.
Implication

Here again, since the implication is between two propositions, the inhabitant is a lambda-expression that expects an inhabitant of a proposition as argument.

\[ H \gg A \Rightarrow B \text{ ext } \lambda x. M \]
\[ x : A \gg B \text{ ext } M \]

\[ H, f : A \Rightarrow B \gg T \text{ ext } (\lambda v. N)(f(M)) \]
\[ \text{BY elim } f \text{ new } v \]
\[ \gg A \text{ ext } M \]
\[ v : B \gg T \text{ ext } N \]

The elimination rule above is unnecessary, as the "implication-cut" rule subsumes it.

Disjunction

\[ H \gg A \lor B \text{ ext } inl(M) \]
\[ \text{BY intro left} \]
\[ \gg A \text{ ext } M \]

\[ H \gg A \lor B \text{ ext } inr(M) \]
\[ \text{BY intro right} \]
\[ \gg A \text{ ext } M \]

\[ H, x : A \lor B \gg T \text{ ext } decide(x; u. M; v. N) \]
\[ \text{BY elim new } u, v \]
\[ u : A \gg T \text{ ext } M \]
\[ v : B \gg T \text{ ext } N \]

The Void Proposition (⊥)

The void proposition, ⊥, represents falsehood, and has only one rule, that being the rule of hypothesis, explained previously. The effect of this formulation of the void proposition is to make our presentation of HA a minimal logic. A minimal logic is distinguished from other kinds of logics in that one does not have a way of proving \( \bot \gg T \) for arbitrary \( T \). Getting ahead of ourselves, the advantage of such a logic is that the Friedman Α-translation is trivial; since we only prove \( \bot \) from \( \bot \), we can always replace \( \bot \) with some arbitrary type \( A \).

Induction

Finally, we come to the rule of induction. It is formulated as a rule, rather than an axiom, because we wish to instantiate it for each predicate separately, keeping the theory first-order.
9.7 Classical Arithmetic: The Extension

\[ n^D : N \gg T \text{ ext } \text{ind}(n^D; B; i, n^D.F(n - 1^D)(i)) \]
\[ \text{BY induction} \]
\[ \gg T[0^D/n^D] \text{ ext } B \]
\[ \gg \forall m^D \in N. T[m^D/n^D] \Rightarrow T[S(m)^D/n^D] \text{ ext } F \]

The Axioms of \( S(\bullet) \), + and \( \times \)

Monotonicity of \( S(\bullet) \)
\[ n : N, \ u : S(n) = 0 \gg \bot \text{ ext } \text{axiom} \]

The meaning of the above extraction is that \( \text{axiom} \) is a witness for \( \bot \) exactly when \( u \) is a witness for \( S(x) = 0 \).

Injectivity of \( S(\bullet) \)
\[ x, y : N, \ u : x^D = y^D \gg S(x)^D = S(y)^D \text{ ext } \text{axiom} \]

Surjectivity of \( S(\bullet) \)
\[ x, y : N, \ u : S(x)^D = S(y)^D \gg x^D = y^D \text{ ext } \text{axiom} \]

Axioms of +
\[ n : N \gg x + 0^D = x^D \text{ ext } \text{axiom} \]
\[ x, y : N \gg x + S(y)^D = S(x + y)^D \text{ ext } \text{axiom} \]

Axioms of \( \times \)
\[ x : N \gg x \times 0^D = 0^D \text{ ext } \text{axiom} \]
\[ x, y : N \gg x \times S(y)^D = x \times y + x^D \text{ ext } \text{axiom} \]

9.7 Classical Arithmetic: The Extension

We must now extend our formalization of Heyting Arithmetic to Peano Arithmetic. This is trivial, consisting in the addition of one rule of inference, and two new programming language constructs.

New Programming Features

The new programming language constructs are \( CM \) and \( AM \), as presented before. They are unary operators which bind no variables, and have no free variables.
9.7.1 New Logical Features

The only rule we need add is the rule of double-negation elimination. This rule is stated as follows (with witness explicitly shown):

\[ H \implies T \text{ ext } CM \text{ by double-negation elim} \]
\[ \implies \neg \neg (T) \text{ ext } M \]

For purposes of logical completeness, it is nice to have a typing rule for \( \lambda \), though it is not permitted in programs constructed by the user; hence, not necessary:

\[ H \implies \bot \text{ ext AM} \]
\[ \implies \phi \text{ ext } M \]

9.8 Classical Evaluation Semantics: New Constraints for Total Correctness

The evaluation semantics of \( C \) and \( \lambda \) are as before, but we list them here for completeness:

\[ E[AM] \triangleright_1 M \]
\[ E[CM] \triangleright_1 M(\lambda x.\lambda A(E[x])). \]

Unfortunately, we find that we must make other changes to the logic in order to preserve total correctness. To wit, we must modify all our axioms (of equality, \( S(\cdot) \), \( \times \), \( + \)) so that they eagerly evaluate their noncomputational arguments. For instance, we must modify the axiom of commutativity of equality (symmetry), which was stated as

\[ \implies u : n = m \implies m = n \text{ ext axiom} \]

so that the parameter \( u \) is evaluated eagerly; that is, \( (\lambda u.\lambda A.axiom)(u) \), meaning that when we apply this term to an argument \( N \), and we wish the value of the application, we must first evaluate \( N \) to a value, and then we may return \( axiom \). Put simply, the inhabitant of this axiom is call-by-value in the argument \( u \). The call-by-value (CBV) CPS-translation does not differ from the CBN CPS-translation in its behaviour on function values, i.e. lambda-terms; rather, it differs in its behaviour on applications (and variables). Thus, we need a way to decide which applications are being done “by-value” and which are being done “by-name”. We deal with this by explicitly introducing call-by-value lambda-terms and application terms. Since the only places where these will appear, though, are in the axioms below, we will not add call-by-value function-space type constructors to the language. Add the following clauses to the definition of the programming language:

\[ Exp \equiv \lambda V Var_1.Exp_1 \]
\[ \mid Exp_1(v.Exp_2) \]
the following clause to the definition of values:

\[ Val \equiv \lambda^v \; x.\text{Exp}_1 \]

and modify the evaluation rules so that \((\lambda^v \; x.\text{b})(v \; t) \rightarrow^1 b[t/x]\) if \(t\) is a value. We must modify the definition of \(RD\), by adding the following clauses:

\[
RD \equiv RD_1(v \; N) \\
| \; (\lambda^v \; x.b)(v \; RD_1)
\]

All of these changes simply serve to codify the deterministic by-value evaluation order on \(\lambda^v \; x.b\) and \(M(v \; N)\). No new rules need be added to the logic, since, as we said before, the only places where by-value lambda-terms and application-terms appear is within the new axioms.

Later, we will make use of the fact that all instances of by-value lambda-terms discard their eagerly computed arguments (e.g. \(\lambda^v \; u'.\text{axiom}\)). This will allow us to leave the CPS-translation of the use of \(x\) in \(\lambda^v \; x.x\) undefined (since such a term will never occur). This is not, as we shall explain in the next chapter, an intrinsic limitation; it will be dispensed with later. For purposes of simplicity of presentation, though, we ignore this problem for now.

Finally, we replace most of the axioms for equality and successor with “by-value” versions, which are “by-value” in their arguments which are equalities:

The New Axioms of Equality

The Rule of Equality  Unchanged

Reflexivity of  =  Unchanged

Symmetry

\(n : N, \; m : N, \; u : n = m \gg m = n \; \text{ext} \; (\lambda^v \; u'.\text{axiom})(v \; u)\)

Jumping ahead of ourselves again, note that we may not contract \((\lambda^v \; u'.\text{axiom})(v \; u)\) to \text{axiom} since the application is by-value, and not by-name.

Transitivity

\(a, b, c : N, \; u : a = b, \; v : b = c \gg a = c \; \text{ext} \; (\lambda^v \; u', v'.\text{axiom})(v \; u)(v \; v)\)

The New Axioms of \(S(\bullet)\)

Monotonicity of \(S(\bullet)\)

\(n : N, \; u : S(n) = 0 \gg \bot \; \text{ext} \; (\lambda^v \; u'.\text{axiom})(v \; u)\)

The meaning of the above extraction is that \text{axiom} is a witness for \(\bot\) exactly when \(u\) is a witness for \(S(x) = 0\). However, now we must first evaluate \(u\) eagerly, to determine if \(u\) is actually a witness for \(S(x) = 0\), or if it is simply a nonlocal goto (a “throw”).
Injectivity of $S(\bullet)$

$$x, y : \mathbb{N}, \ u : x^D = y^D \Rightarrow S(x)^D = S(y)^D \ \text{ext} \ (\lambda \nu \ u'.axiom)(\nu u)$$

Surjectivity of $S(\bullet)$

$$x, y : \mathbb{N}, \ u : S(x)^D = S(y)^D \Rightarrow x^D = y^D \ \text{ext} \ (\lambda \nu \ u'.axiom)(\nu u)$$

Of course, at this point, there is no good reason for thinking that these modifications are necessary; Later, though, we will find that the Kolmogorov-translation of the axiom for symmetry of equality is the by-value CPS-translation of $(\lambda \nu \ u'.axiom)(\nu u)$ and not the by-name translation of axiom. On the one hand, it is our wish to "guess" a classical axiom which, when translated, yields the known CPS-translated axiom, which drives us to make the above changes. On the other hand, though, it is our knowledge that $(\lambda \ u.axiom)(M)$ is a witness for $a = b$ only if $M$ is a witness for $b = a$, and in a programming language with nonlocal goto's, we can only determine if this is the case by evaluating $M$ eagerly down to a value.

9.9 Call-by-Name (Almost) CPS-Translation and Kolmogorov Translation

In this section, we make the connection between CPS-compilation and double-negation/A-translation. A CPS-translation is defined by specifying three things:

- The translations on types of programs; that is, given a subexpression $M$ of type $T$ in a program of type $\phi$, what is the type of $M$? We must define the fundamental concepts of printable value, implicit value, continuation, and top-level continuation.

- A particular closed expression which we call the "top-level continuation". This expression fixes the evaluation order on $RC$.

- A particular compatible translation on programs, which we must show to be compatible with the previous two choices.

We then demonstrate that we can understand this translation on types and programs as a combined double-negation/A-translation which is much like the original Kolmogorov-translation.

We go on to re-prove Friedman's conservative extension result, both in terms of the double-negation/A-translation and the CPS-translation. Finally, we connect the "top-level trick" of Friedman to the concept of top-level continuation in CPS-translation, thus showing that CPS-translation + top-level continuation is equivalent to double-negation/A-translation + Friedman's top-level trick.
9.9.1 The CPS-Translation on Types: Preliminaries

Before we can start translating programs, we must decide what our encoding of values will be. That is, given a program \( M \in T \), what type will \( M \) have? For the particular translation we have in mind, this is not a difficult problem. We address this question with the following definitions, which should be self-evident.

First, we define what we mean by a "printable type". Intuitively, such a type is one whose values (but not expressions) cannot contain any unevaluated continuations. The idea is that an integer expression, even if it contains unevaluated continuations, can always be computed to a form in which all continuations are evaluated (namely when the expression evaluates to a numeral). An expression which witnesses an equality is much like an integer, for the same reasons. Likewise with all concrete data types. We call types like these printable.

**Definition 9.9.1 (Printable Type)** A printable type is a type such as \( N \), an equality, e.g. \( a = b \), or composed of these via conjunction, disjunction, or existential quantification.

Now we must define what we mean by an "implicit value". That is, if we have a program \( M \in \phi \), and in this program we have a sub-expression \( N \) of type \( T \), what type should \( N \) have? It turns out that this type depends upon the type \( \phi \), and looks a lot like a Kolmogorov/A-translation of \( T \):

**Definition 9.9.2 (CPS-Translation on Types)** Given a type \( T \) for a sub-expression in a program of type \( \phi \), we define the CPS-translated type \( \overline{T} \), as follows (let \( \overline{\gamma}(X) \equiv X \Rightarrow \phi \), and call this the \( \phi \)-ation of a term \( X \)):

\[
\begin{align*}
\overline{a = b} & \equiv \overline{\gamma}(a = b) \\
\overline{A \land B} & \equiv \overline{\gamma}(A \land B) \\
\overline{(A \lor B)} & \equiv \overline{\gamma}(A \lor B) \\
\overline{(\exists x \in A.B)} & \equiv \overline{\gamma}(\exists x \in A.B) \\
\overline{(\forall x \in A.B)} & \equiv \overline{\gamma}(\forall x \in A.B) \\
\overline{A \Rightarrow B} & \equiv \overline{\gamma}(A \Rightarrow B) \\
\overline{1} & \equiv \overline{\beta}
\end{align*}
\]

**Definition 9.9.3 (Star-Translation on Values)** Define \( T^* \) as \( \overline{T} \), but with the outermost "double-\( \phi \)-ation" stripped away. That is, \( \overline{T} \equiv \overline{\gamma}(T^*) \).

And the definition of an implicit value becomes:

**Definition 9.9.4 (Typings of Implicit Values)** An implicit expression of type \( T \) in a program of type \( \phi \) has type

\( I_\phi(T) \equiv \overline{T} \).
We can now define what we mean by a continuation of type \( T \) in a program of type \( \phi \):

**Definition 9.9.5 (Typings of Continuations)** A continuation of type \( T \) in a program of type \( \phi \) has type

\[
K_\phi(T) \equiv (T^* \Rightarrow \phi) \equiv \neg_\phi(T^*).
\]

We have not defined the CPS-translation on programs yet, but we will define top-level continuations now, rather than wait until after defining CPS-translations. A top-level continuation of type \( \phi \) is

**Definition 9.9.6 (Top-Level Continuation)** A top-level continuation of type \( \phi \) is a term \( \tau_\phi \) of type \( T_\phi \equiv K_\phi(\phi) \equiv (\phi^* \Rightarrow \phi) \), such that for any value \( V \) of type \( \phi \), \( \forall_\tau_\phi \rightarrow V \) (\( \forall_\tau_\phi \), the CPS-translation of \( V \), applied to the proper top-level continuation, computes to \( V \)).

This last idea may be a bit unclear. Consider an expression \( b \) in the type \( \exists x \in \text{N}. f(x) = 0 \). It can always be computed to a form without explicit continuations, namely by first computing \( b \) to a pair, then computing each part of the pair down to a value in turn, and finally putting the two new values back together to form a pair again. This operation, though, is much more complex than that required to normalize an integer value. In any case, the operations we described are the task of the top-level continuation in a continuation semantics. Usually we assume that programs compute to some basic type, e.g. \( \text{N} \). In such a case, the top-level continuation is fixed, and is usually \( \lambda x : \text{N}.x \). In the programming languages for which this simple continuation is applicable, the only operations are upon integers; other operations are assumed to be coded up using the integers as a substrate. For our work, though, we have explicit representations of disjoint unions and pairs, and we would like to have a notion of the correct top-level continuation for some structured data-type. Thus, the top-level continuation is different for each different type of the top-level program, and in fact, for each different evaluation order upon \( RC \) (the reducible constructors, defined previously). We will simply list out some examples of top-level continuations for different value of \( \phi \), and note where the dependence upon \( RC \) comes in:

\[
\begin{align*}
\tau_P & \equiv \lambda x.x \ (P \ a \ primitive \ type) \\
n \exists x \in \text{N}. f(m, x) = 0 & \equiv \lambda p.\text{spread}(p; n, v.\lambda m.(n, m))) \\
\tau_P \lor Q & \equiv \lambda \text{d.decide}(d; u.u(\lambda m.\text{inl}(m)); v.v(\lambda n.\text{inl}(n))) \\
(P, Q \ primitive) \\
\tau_P \land Q & \equiv \lambda p.\text{spread}(p; u, v.\lambda n.v(\lambda m.(n, m)))) \\
(P, Q \ primitive).
\end{align*}
\]

In the last case above, we could have instead used

\[
\lambda p.\text{spread}(p; u, v.\lambda n.u(\lambda m.(n, m))))
\]
(switching $u$ and $v$). This would have the effect of specifying that at top-level, pairs are evaluated from right-to-left.

Notice that applying an expression of type $\text{I}_\phi(T)$ to an expression of type $\text{K}_\phi(T)$, yields an expression of type $\phi$. This is exactly what we would expect, since applying an implicit value of type $T$ to a continuation which expects a value of type $T$ should yield us a value whose type is that of the program itself.

One simple fact that we will have occasion to use is the following:

**Fact 9.9.1.** For every printable type $\phi$, there exists a value $\tau_\phi$ of type $T_\phi$ which is a proper top-level continuation.

And finally, some examples to make all of this clear; suppose our program has type $\phi \equiv \exists x \in \text{N}. f(x) = 1$, and the type of our subexpression is $T = \forall x \in \text{N}. g(x) = h(x)$. Then

\[
\overline{\phi} \equiv \text{I}_\phi(\phi) \equiv \exists x \in \text{N}. \neg \neg (f(x) = 0) \\
\phi^* \equiv \exists x \in \text{N}. \neg \neg (f(x) = 0) \\
\text{I}_\phi(T) \equiv \forall x \in \text{N}. \neg \neg (f(x) = g(x)) \\
\text{K}_\phi(T) \equiv \forall x \in \text{N}. \neg (f(x) = g(x)) \\
T_\phi \equiv \text{K}_\phi(\phi) \equiv \exists x \in \text{N}. \neg (f(x) = 0) \\
\tau_\phi \equiv \lambda p. \text{spread}(p; n, v.v(\lambda m.(n, m)))
\]

### 9.9.2 The CPS-Translation on Programs

The CPS-translation on programs is basically the call-by-name CPS-translation from our simple example, but extended for our wider range of data constructors and destructors, by-value lambda-terms and by-value applications, and $C$ and $A$. One of the facts about programs that we will refer to again and again is:

**Fact 9.9.2.** Given a term $M^D$, every subterm of that term is also marked thus.

The translation is as follows:
\[
\begin{align*}
x & \equiv x \text{ (a variable)} \\
MN & \equiv \lambda k.M(\lambda m.mNk) \\
M(N^D) & \equiv \lambda k.M(\lambda m.m(N^D)k) \\
\lambda x.M & \equiv \lambda k.k(\lambda x.M) \\
\lambda v^D.M & \equiv \lambda k.k(\lambda v.M) \\
\langle M, N \rangle & \equiv \lambda k.k(\langle M, N \rangle) \\
\langle M^D, N \rangle & \equiv \lambda k.k(\langle M^D, N \rangle) \\
\text{spread}(M; u, v.N) & \equiv \lambda k.M(\lambda p.\text{spread}(p; u, v.N(k))) \\
\text{spread}(M; u^D, v.N) & \equiv \lambda k.M(\lambda p.\text{spread}(p; u^D, v.N(k))) \\
\text{inl}(M) & \equiv \lambda k.k(\text{inl}(M)) \\
\text{inr}(M) & \equiv \lambda k.k(\text{inr}(M)) \\
\text{decide}(M; u.N; v.R) & \equiv \lambda k.M(\lambda d.\text{decide}(d; u.N(k); v.R(k))) \\
\text{ind}(M; B; n, i.I(n-1)(i)) & \equiv \text{ind}(M; B; n, i.I(n-1)(i)) \\
\text{axiom} & \equiv \lambda k.k(\text{axiom})
\end{align*}
\]

The intrepid reader will note that we translated \(\lambda^v x.M\) as \(\lambda k.k(\lambda x.M)\). Implicitly we assume that \(x\) is not free in \(M\), for otherwise \(\overline{x} \equiv x\) would be ill-typed (more on this when we discuss mixed-mode evaluation in Section 10.2).

We must now show that this CPS-translation on expressions translates an expression \(M\) of classical type \(T\) in a program of classical type \(\tau\) to an expression \(\overline{M}\) of constructive type \(\overline{\tau}(T)\), when \(\phi\) is a printable type. Before we state the theorem, let us discuss for a moment the differences between this theorem and the similar theorem we proved for \(LJ\rightarrow\) and \(LK\rightarrow\). In that case, our programming languages were identical (except for \(C\) and \(A\)) for both theories. We showed that we could map an expression with a particular classical type into an expression with an analogous constructive type.

In the current case, though, our programming languages are quite different. Our classical programming language contains, in addition to \(C\) and \(A\), instances of call-by-value functions and applications. Several of the axioms, i.e. symmetry of equality, have terms which contain call-by-value expressions in their classical witnesses. Our constructive logic and programming language, on the other hand, contain no such call-by-value constructs. So what we must prove is that we can map a "classical program" (with \(C\), \(A\), and by-value constructs) with some classical type \(T\), into a constructive program, without all these extended features, with an analogous type \(\overline{T}\). The problem can be divided up into three different, separate tasks:
1. Show that each of the rules and axioms of the common core of HA and PA, that is, the rules that do not change when we extend to classical reasoning, are correctly translated by the CPS-translation.

2. Show that the new rule of double-negation elimination is translated correctly.

3. Show that the new (by-value) axioms for equality, \( S(\bullet) \), etc, are translated correctly.

We will only show that a few of the rules of the common core are correctly translated by CPS-translation, since these are quite simple. We will do the case of \( C \) and \( A \) in detail, since these are really the heart of the result. And we will do several of the new by-value axioms, so that the reader will get a flavor for the manner in which they are done.

**Theorem 9.9.1 (CPS-Translation Correctly Translates Programs)** Given an expression \( M \) of classical type \( T \) in a program \( P \) of classical type \( \phi \), if \( \phi \) is a printable type, then \( M \) is of constructive type \( I_{\phi}(T) \).

**Proof:** By induction on term structure, and cases on the definition of terms (and the translation). We will do the cases for universals, implications, and some of the axioms, since they are the most difficult.

**Universal Intro: Classical Typing:**

\[
\lambda \; n^D . \; M : \forall n \in N. T \\
n^D : N \\
M : T \ (when \ n : N)
\]

**Constructive Typing:**

\[
\lambda \; n^D . \; M \equiv \\
\lambda \; k . \; k(\lambda \; n . \; M) : \neg \neg (\forall n \in N. \overline{T}) \\
k : \neg (\forall n \in N. T) \\
n^D : N \\
M : \overline{T} \ (when \ n : N)
\]

**Universal Cut: Classical Typing:**

\[
M(N^D) : T[N^D/x] \\
M : \forall x \in N. T \\
N^D : N
\]
Constructive Typing:

\[ M(N^D) \equiv \]
\[ \lambda k.M(\lambda m.m(N^D)k) : T[N^D/x] \]
\[ M : \neg \neg (\forall x \in N.T) \]
\[ N^D : N \]
\[ k : \neg (\forall x \in N.T) \]
\[ m : \forall x \in N.T \]

Implication Intro: Classical Typing:

\[ \lambda x.M : A \Rightarrow B \]
\[ x : A \]
\[ M : B \text{ (when } x : A) \]

Constructive Typing:

\[ \lambda x.M \equiv \]
\[ \lambda k.k(\lambda x.M) : \neg \neg (A \Rightarrow B) \]
\[ k : \neg (A \Rightarrow B) \]
\[ x : \neg A \]
\[ M : B \text{ (when } x : \neg A) \]

Implication Cut: Classical Typing:

\[ M(N) : B \]
\[ M : A \Rightarrow B \]
\[ N : A \]

Constructive Typing:

\[ M(N) \equiv \]
\[ \lambda k.M(\lambda m.mNk) : \neg A \]
\[ M : \neg \neg (\neg A \Rightarrow \neg B) \]
\[ N : \neg A \]
\[ k : \neg (\neg A \Rightarrow \neg B) \]
\[ m : \neg A \Rightarrow \neg B \]

Induction: Classical Typing:

\[ ind(n^D, B; i, n^D.F(n - 1^D)(i)) : T[n/x] \]
\[ n : N \]
\[ B : T[0/x] \]
\[ F : \forall m^D \in N.T[m^D/n^D] \]
\[ \Rightarrow T[S(m)^D/n^D] \]
\[ i : \neg T[n-1/x] \]
\[ F(n-1)(i) : T[n/x] \]
9.9 Call-by-Name (Almost) CPS-Translation and Kolmogorov Translation

Constructive Typing:

\[
\begin{align*}
\text{ind}(n^D; B; i, n^D.F(n-1^D)(i)) & \equiv \text{T}[n/x] \\
\text{ind}(n^D; B; i, n^D.F(n-1^D)(i)) & : T[n/x] \\
\text{B} & : T[0/x] \\
i & : T[n - 1/x] \\
F(n-1^D)(i) & : T[n/x]
\end{align*}
\]

\[C: \text{Classical Typing:}\]

\[CM : T\]
\[M : \neg\neg(M)\]

Constructive Typing:

\[
CM \equiv \\
\lambda k.M(\lambda m.m(\lambda g.g(\lambda v, h.v(k))(\tau_\phi))(\tau_\phi)) : \text{T} \\
M : \neg\neg(\neg\neg(\text{T} \Rightarrow \phi) \Rightarrow \phi) \\
k : \neg(\text{T}^*) \equiv \text{T}^* \Rightarrow \phi \\
m : \neg\neg(\text{T} \Rightarrow \phi) \Rightarrow \phi \\
g : \neg\neg(\text{T} \Rightarrow \phi) \\
v : \text{T} \\
h : \phi^*(\phi) \equiv T_\phi
\]

\[A: \text{Classical Typing:}\]

\[AM : \bot\]
\[M : \phi\]

Constructive Typing:

\[
AM \equiv \\
\lambda k.M_{\tau_\phi} : \phi \\
M : \phi \\
k : \phi^*(\phi^*) \equiv T_\phi
\]

Symmetry of =: Classical Typing:

\[(\lambda^v u'.axiom)(v_u) : m = n \text{ (when } u : n = m)\]
Constructive Typing:

\((\lambda^v \ u'.axiom)(\varphi u)\) ≡
\[\lambda \ k.(\lambda^v \ u'.axiom)(\lambda \ a.u(\lambda \ b.a(b)(k)))\]

\[u : \neg\neg(n = m)\]
\[k : \neg\neg(m = n)\]
\[\lambda^v \ u'.axiom : \neg\neg(n = m)\]
\[\Rightarrow \neg\neg(m = n)\]
\[u' : n = m\]
\[a : n = m \Rightarrow \neg\neg(m = n)\]
\[u : \neg\neg(n = m)\]
\[b : n = m\]

Recall that we stipulated that the argument \(x\) in a by-value lambda-term \(\lambda^v x.M\) not be free (used) in \(M\). We said that otherwise the translation would be ill-typed. Now we can see what the problem is. Consider \(\lambda^v x.x \equiv \lambda \ k.k(\lambda \ x.x)\). If \(\lambda^v x.x : A \to A\), then

\[(\lambda^v \ x.x)(\varphi N) \equiv \lambda \ k.(\lambda \ k.k(\lambda \ x.x))(\lambda \ m.N(\lambda \ n.m(n)(k)))\]
\[\Rightarrow \lambda \ k.N(\lambda \ n.n(k)).\]

Under the induction scheme we must follow, the typings must be:

**Classical Typing:**

\[\lambda^v \ x.x : A \to A\]
\[N : A\]
\[(\lambda^v \ x.x)(\varphi N) : A\]

**Constructive Typing:**

\[\lambda^v \ x.x : A \to A\]
\[N : A\]
\[(\lambda^v \ x.x)(\varphi N) : A\]

Clearly, \(n(k) : \phi\) is ill-typed, and so \(\lambda \ k.N(\lambda \ n.n(k))\) does not have type \(A\). The fix is to make

\[M(\varphi N) \equiv \lambda \ k.M(\lambda \ m.N(\lambda \ n.m(\lambda \ g.g(n))k)).\]
or mark by-value variable bindings specially, and make \(x^\nu \equiv \lambda k.k(x)\). We will provide more detail when we discuss mixed-mode evaluation. Note that the CPS-translation of a by-value lambda-expression is exactly the same as that of a by-name lambda-expression. The by-value nature of the expression is enforced at application time, as evidenced by the translation of call-by-value application.

So we have demonstrated that the CPS-translation we wrote down on types is compatible with the translation we described for terms, in the sense that given a classical program \(M\) of classical type \(T\), the constructive program \(\overline{M}\) has constructive type \(\overline{T}\).

### 9.9.3 The Compatible Kolmogorov/A-Translation on Proofs

At the beginning of this chapter, we said we would show that double-negation/A-translation and CPS-translation were exactly the same thing. Finally, we are ready to demonstrate this fact. First, recall the definition of the CPS-translation upon types that we specified previously. It is a simple fact that we can factor this CPS-translation on types into three translations, which are applied one after the other:

**Preliminary A-Translation** First, replace every instance of \(\bot\) with \(\phi\), the top-level type of the program.

**Double-Negation Translation** Perform a standard Kolmogorov-translation. Since we replaced every instance of \(\bot\) with \(\phi\), and \(\phi\) is assumed printable, we know that there are no instances of \(\bot\) in the proof tree before this step.

**Final A-Translation** Replace every instance of \(\bot\) introduced by double-negation translation with \(\phi\).

**Theorem 9.9.2.** The preceding three steps are, together, equivalent to the CPS-translation upon types.

So now we know that the CPS-translation upon types is just a modified double-negation/A-translation upon types. Since we showed that, given a classical program \(M\) of classical type \(\phi\), the program \(\overline{M}\) is of constructive type \(\overline{T}_\phi(\phi)\), we now know that the CPS-translation upon classical programs is simply a double-negation/A-translation upon them.

### 9.9.4 Equivalence of The Translations : Conservative Extension Revisited

Finally, we can prove that the conservative extension result of Friedman falls out naturally from the the typings generated by CPS-translation. We will prove:
Theorem 9.9.3 (Conservative Extension for Printable Types) Given a program \(M\) of a printable classical type \(\phi\), the program \(M(\tau_\phi)\) has constructive type \(\phi\).

Proof: Trivial. \(M\) has constructive type \(\mathcal{I}_\phi(\phi)\), and \(\tau_\phi\) has constructive type \(\mathcal{K}_\phi(\phi)\). The application then has type \(\phi\).

Corollary 9.9.1 (Conservative Extension for \(\Pi^0_2\) Sentences) For a \(\Pi^0_2\) sentence \(\Phi \equiv \forall x \in \mathbb{N}. \exists y \in \mathbb{N}. R(x, y)\), and a PA proof with classical witness \(\lambda x. M\), we can obtain a constructive proof of \(\Phi\) whose constructive witness is

\[
\lambda x. \left( M \tau_\exists y \in \mathbb{N}. R(x, y) \right).
\]

Proof: \(\exists y \in \mathbb{N}. R(x, y)\) is a printable type. Therefore the previous theorem applies. The universal quantification is handled by quantifying over all numerals.

In Chapter 6, we got as far as showing that we could constructively prove

\[ x : \text{Dom} \vdash_I ( (\exists y \in \text{Rng.} \Phi(x, y) \lor A) \Rightarrow A ) \Rightarrow A \]

where \(A = \exists y \in \text{Rng.} \Phi(x, y)\). Then it was argued that a simple argument sufficed to prove \(\exists y \in \text{Rng.} \Phi(x, y)\). Since \(\Phi(x, y)\) is decidable, \(\Phi(x, y) \lor A\) is equivalent to \(\neg \neg (\Phi(x, y))\), and now we are reduced to proving \((\exists y \in \text{Rng.} \neg \neg (\Phi(x, y))) \Rightarrow A\), which is just the top-level continuation of type \(A = \exists y \in \text{Rng.} \Phi(x, y)\).

And now we have come full circle, to see that Friedman’s top-level trick is simply the proof of existence of a top-level continuation. So we see that Friedman’s entire method is simply (and beautifully!) a CPS-compilation upon classical programs.

### 9.10 Semantic Equivalence: Value-Preserving CPS-Compilation

In this chapter we have shown that a modified Kolmogorov/A-translation upon classical sequent calculus proofs is equivalent to CPS-compilation upon the classical witnesses from those classical proofs. We already have a notion of the evaluation semantics of classical programs, via the evaluation rules for \(C\) and \(A\).

All that is left to do is show that a classical program \(M\) of printable type \(\phi\) evaluates to the same value as the constructive program \(M(\tau_\phi)\). That is, we must show that the two programs are semantically equivalent.

**Desired Theorem 9.10.1 (Semantic Equivalence)** Given a proof of a printable type \(\phi\),

\[ \vdash_{PA} \phi \ ext \ M, \]

and the Kolmogorov/A-translated proof, applied to the proper top-level continuation,

\[ \vdash_{HA} \phi \ ext \ (M)(\tau_\phi), \]
9.10 Semantic Equivalence: Value-Preserving CPS-Compilation

\( M \) evaluates to the same term as \((M)_{\tau_{\phi}}\).

The proof of this theorem is laborious, and involves two steps: First, we show

**Lemma 9.10.1 (Soundness of CPS-translation)**
Whenever \( M \to N \), \((M)_{\tau_{\phi}} = (N)_{\tau_{\phi}}\).

*Proof:* Trivial - just check that the property holds for every reduction rule. ■

This establishes that if a reduction sequence on \( M \) terminates in a value \( N \), then we can prove \((M)_{\tau_{\phi}} = (N)_{\tau_{\phi}}\). As we observed previously, given a value \( N \) of type \( \phi \), \((N)_{\tau_{\phi}} = N \). Thus,

**Theorem 9.10.1.** If a reduction sequence starting with \( M \) terminates, then it terminates with a value \( N \) equal to that produced by reducing \((M)_{\tau_{\phi}}\).

*Proof:* An easy consequence of the soundness of CPS-translation. ■

### 9.10.1 Motivation

We have still not shown that a classical program \( M \) of printable type \( \phi \) will terminate. To do so, we resort to a method developed by Plotkin [Plo75], and extended by Griffin - the "colon-translation". Plotkin showed the semantic equivalence of a constructive program \( M \) with its CPS-translation for the lambda-calculus, without the plethora of different constructors in our language. Griffin extended Plotkin’s result to classical programs in a similar lambda-calculus.

Plotkin's method is based on the (simple) observation that the reduction sequences that translated terms undergo are well-defined. When one looks at reduction sequences on \( M_{\tau_{\phi}} \), one notices that they follow a certain pattern: zero or more administrative reductions, which serve to do the "bookkeeping" associated with a particular evaluation strategy, followed by a single proper reduction, followed by more administrative reductions, and so on. If we could contract all the administrative reducts in \( M_{\tau_{\phi}} \) and \( N_{\tau_{\phi}} \), then we could prove that our modified \( M' \) reduces to our modified \( N' \). So what we do is define a new translation, \( M : \tau_{\phi} \), such that whenever \( M \to N \), \( M : \tau_{\phi} \to N : \tau_{\phi} \). This is the approach we will take. As we noted before, it is a direct extension of work of Plotkin and Griffin.

### 9.10.2 The Colon Translation

We will define in parallel three different translations upon terms: the star translation upon values, the colon translation upon terms, and the double-underbar translation upon terms. In the rest of this chapter, we assume that negation of a formula \( T \) is written as \( T \Rightarrow \phi \), where \( \phi \) is implicitly the type of the program we are working with.
Definition 9.10.1 (Double Underbar Translation)

\[ \bar{x} \equiv x \ (x \ a \ variable) \]
\[ \overline{M} \equiv \lambda k.M : k \]

Definition 9.10.2 (Star Translation Upon Values)
Define the function \( \Psi \), as follows:

\[ \Psi(\lambda x.M) \equiv \lambda x.M \]
\[ \Psi(\lambda^o x.M) \equiv \lambda x.M \]
\[ \Psi((M, N)) \equiv <M, N> \]
\[ \Psi(inl(M)) \equiv inl(M) \]
\[ \Psi(inr(M)) \equiv \psi(axiom) \equiv axiom \]

Definition 9.10.3 (Colon Translation) Given a program of type \( \phi \), if we have a term \( \tau_\phi \) (\( \phi \) is a printable type), we can define the “colon” translation as follows:

\[ x : K \equiv x(K) \ (x \ a \ variable) \]
\[ V : K \equiv K(\Psi(V)) \]
\[ (when \ V \ is \ a \ value \ and \ K \ is \ not \ \tau_\phi) \]
\[ M(N) : K \equiv M : \lambda m.m(N)K \]
\[ M(N^D) : K \equiv M : \lambda m.m(N^D)K \]
\[ \text{spread}(M; u, v.T) : K \equiv M : \lambda p.\text{spread}(p; u, v.(T : K)) \]
\[ \text{decide}(M; u.T; v.F) : K \equiv M : \lambda d.\text{decide}(d; u.(T : K); v.(F : K)) \]
\[ \text{ind}(E; B; n, i.I(n - 1)(i)) : K \equiv (\text{ind}(E; B; n, i.I(n - 1)(i))(K)) \]
\[ \text{(when E is greater than zero)} \]
\[ \mathcal{C}M : K \equiv M : \lambda m.m(\lambda g.g(\lambda v, h.v(K)))\tau_\phi \]
\[ \mathcal{A}M : K \equiv M : \tau_\phi \]
\[ M(v.N) : K \equiv M : \lambda n.(N : \lambda n.m(n)K) \]
\[ V(v.N) : K \equiv N : \lambda n.\Psi(V)(n)K \]
\[ V_1(vV_2) : K \equiv \Psi(V_1)\Psi(V_2)K \]

It is trivial to prove

Fact 9.10.1. For every value \( V \) of type \( T \), \( \Psi(V) \) has type \( T^* \).

Fact 9.10.2. For every term \( M \) of type \( T \), \( \overline{M} \) has type \( \overline{T} \).

We can prove the following very useful theorem about this translation (adapted from Griffin):

Lemma 9.10.2 (Evaluation Context Unwrapping) For all evaluation contexts \( E \) consisting only of destructors (i.e. defined via \( RD \)), all terms \( M \), and values \( K \)

\[ E[M] : K = M : K^E \]

where \( K^E \) is defined inductively on \( E \) as
1. $K^\emptyset = K$,
2. $K^{E_1(N)} = (\lambda m.m(N)K)^{E_1}$,
3. $K^{E_1(N^D)} = (\lambda m.m(N^D)K)^{E_1}$,
4. $K^{spread(E_1; u, v.T)} = (\lambda p.spread(p; u, v.T : K))^E_1$,
5. $K^{decide(E_1; u.T; v.F)} = (\lambda d.decide(d; u.T : K; v.F : K))^E_1$,
6. $K^{E_1(v^N)} = (\lambda m.(N : \lambda n.m(n)K))^E_1$

Proof: By the definition of the colon-translation. ■

Conspicuous by their absence are the rules for unwrapping constructors. That is, the colon-translation as defined above only translates a term $E[M] : K$ to $M : K^E$ when $E[]$ is in the syntactic class $RD$, that of reducible destructors. If we had $inl(M) : K$, we would not be able to show this equal to $M : K'$ for some $K'$. As a result, we would not be able to show that $inl(M) : K \rightarrow inl(M') : K$ when $M \rightarrow M'$. But such a situation (where we wanted to reduce a subterm of a constructor) would only come up if the subterm were being evaluated at top-level. All previous expositions of CPS-translation that we have come across have skipped over this point, by requiring that all final values be atomic; that is, no value has a value as a subterm. Hence, top-level continuations for all previous CPS-translations were just $\lambda x.x$.

We already defined the colon-translation for constructors when $K$ is not the top-level continuation, and so all that we need to do is handle this special case. In defining the colon-translation for constructors at top-level, we are aided by the knowledge that the constructor appears at the top-level of an evaluation context, e.g. as an instance of $RC$. Thus, we will assume knowledge of the top-level continuation in what follows.

Given $\phi$, the type of a program $M$, we define by induction upon terms the meaning of $M : \tau_\phi$ for the cases where $M$ is not at top-level either a redex or a destructor (i.e. $M$ falls in the class $RC$). We do this for three different instances of $\phi$, and hope that the reader will discern a pattern, as expressing it would require invention of an outrageous amount of notation. The basic idea is that any "reasonable" inhabitant of $T_\phi \equiv \phi^* \Rightarrow \phi$ will suffice, though different inhabitants may give different top-level evaluation semantics.

**Definition 9.10.4 (Colon Translation for $RC$)** Given $\phi$, the type of a program $M$, define by term induction the meaning of $M : \tau_\phi$ when $M$ is at top-level a constructor ($M, N$ are terms, $V$ are values, $K$ are continuations):

$\phi \equiv \exists n \in N.f(x) = 0$:

\[
\begin{align*}
\tau_\phi & \equiv \lambda p.spread(p; m, v.v(\lambda n.(m, n))) \\
\langle M^D, N \rangle : \tau_\phi & \equiv N : \lambda n.(M^D, n) \\
\langle M^D, V \rangle : \tau_\phi & \equiv \langle M^D, V \rangle
\end{align*}
\]
\( \phi \equiv P \lor Q \): where \( P, Q \) are atomic, e.g. equalities.

\[
\begin{align*}
\tau_{\phi} & \equiv \lambda d. \text{decide}(d; u.u(\lambda m. \text{inl}(m)); v.v(\lambda n. \text{inr}(n))) \\
\text{inl}(M) : \tau_{\phi} & \equiv M : \lambda m. \text{inl}(m) \\
\text{inl}(V) : \tau_{\phi} & \equiv \text{inl}(V) \\
\text{inr}(N) : \tau_{\phi} & \equiv N : \lambda n. \text{inr}(n) \\
\text{inr}(V) : \tau_{\phi} & \equiv \text{inr}(V)
\end{align*}
\]

\( \phi \equiv P \land Q \): where \( P, Q \) are atomic.

\[
\begin{align*}
\tau_{\phi} & \equiv \lambda p. \text{spread}(p; u, v.u(\lambda m. v(\lambda n. \langle m, n \rangle))) \\
\langle M, N \rangle : \tau_{\phi} & \equiv M : \lambda m. (N : \lambda n. \langle m, n \rangle) \\
\langle V, N \rangle : K & \equiv N : \lambda n. \langle V, n \rangle \\
\langle V_1, V_2 \rangle : \tau_{\phi} & \equiv \langle V_1, V_2 \rangle
\end{align*}
\]

We trust that the reader can extend this to a definition for any printable type \( T \), and also note that the most important case, that of \( \phi \equiv \exists n \in N. f(x) = 0 \), has been covered completely.

At this point, we would like to prove a lemma analogous to Lemma 9.10.2, which would tell us that, for every evaluation context \( E \), every term \( M \), and for every value \( K, E[M] : K = M : K^E \). Unfortunately, since some of the colon-translation rules only apply at top-level, we cannot prove this result. But we can prove something weaker, which is in some sense a more appropriate result:

**Lemma 9.10.3 (Evaluation Context Unwrapping for RC)** For all evaluation contexts \( E \), all terms \( M \), if \( E[M] \) is a program of type \( \phi \), then

\[
E[M] : \tau_{\phi} = M : \tau_{\phi}^E
\]

where \( \tau_{\phi}^E \) is defined inductively on \( E \) as

1. if \( E \) is in \( RD \), then \( \tau_{\phi}^E \) is defined as in Lemma 9.10.2,
2. if \( \phi \equiv \exists n \in N. f(x) = 0 \), \( E[M] = \langle A^D, E_1[M] \rangle \), \( E_1 \) is in \( RD \), then \( \tau_{\phi}^E = (\lambda n. (A^D, n))^{E_1} \), where the latter is defined via Lemma 9.10.2,
3. if \( \phi \equiv P \lor Q \), there are two cases:
   (a) \( E[M] = \text{inl}(E_1[M]) \), \( E_1 \) is in \( RD \), then \( \tau_{\phi}^E = (\lambda m. \text{inl}(m))^{E_1} \), where the latter is defined as before.
   (b) \( E[M] = \text{inr}(E_1[M]) \), \( E_1 \) is in \( RD \), then \( \tau_{\phi}^E = (\lambda n. \text{inr}(n))^{E_1} \), where the latter is defined as before.
4. if \( \phi \equiv P \land Q \), there are two cases:
   (a) \( E[M] = \langle E_1[M], N \rangle \), \( E_1 \) in \( RD \), then \( \tau_{\phi}^E = (\lambda m. (N : \lambda n. \langle m, n \rangle))^{E_1} \), where the latter is defined as before.
(b) \( E[M] = \langle V, E_1[M] \rangle, E_1 \) in \( RD \), then \( \tau_{\phi}^E = (\lambda \ n. (V, n))^E_1 \), where the latter is defined as before.

Proof: The proof of this theorem is easy from the definition of the colon-translation. The reader will note that we do not need to handle all the cases of the definition of \( RC \) (in the definition of evaluation contexts) because the well-typedness of \( E[M] \) guarantees that any evaluation context is well-typed, and thus is of a form determined by the top-level theorem we are trying to prove.  

The plan for the semantic equivalence result should now be clear; we use the unwrapping lemma to convert an evaluation context \( E[M] : \tau_{\phi} \) to \( M : \tau_{\phi}^E \), reduce \( M \) one step, to \( N : \tau_{\phi}^E \), and then claim that by the unwrapping lemma, \( E[N] : \tau_{\phi} = N : \tau_{\phi}^E \). But this is not true. To be exact, in the case where \( M \) was a redex, the innermost term in \( E \) is a constructor, and \( N \) is a value, the last equality does not hold. So we have to prove a "re-wrapping" lemma, which lets up wrap up a just-reduced term in its evaluation context. Before progressing any further, we gather up the definition of \( K^E \):

**Definition 9.10.5 (Unwrapped Evaluation Context)** An unwrapped evaluation context \( E \) for top-level continuation \( \tau_{\phi} \) is written \( \tau_{\phi}^E \), and is defined by induction upon \( E \) and cases upon \( \phi \):

**Independent of \( \phi \):**

\[
K^\emptyset = K
\]
\[
K^{E_1(N)} = (\lambda \ m. m(N)K)^{E_1}
\]
\[
K^{E_1(N^D)} = (\lambda \ m. m(N^D)K)^{E_1}
\]
\[
K^{\text{spread}(E_1; u, v.T)} = (\lambda \ p. \text{spread}(p; u, v.T : K))^E_1
\]
\[
K^{\text{decide}(E_1; u.T; v.F)} = (\lambda \ d. \text{decide}(d; u.T : K; v.F : K))^E_1
\]
\[
K^{E_1(vN)} = (\lambda \ m. (N : \lambda \ n. m(n)K))^E_1
\]
\[
K^{V(vE_1)} = (\lambda \ n. \Psi(V)(n)K)^E_1
\]

\( \phi \equiv \exists n \in \mathbb{N}. f(x) = 0 \):

\[
\tau_{\phi} = (\lambda \ p. \text{spread}(p; m, v.v(\lambda \ n. (m, n))))
\]
\[
\tau_{\phi}(A^{D}, E_1) = (\lambda \ n. (A^{D}, n))^E_1
\]

\( \phi \equiv P \lor Q \): where \( P, Q \) are atomic, e.g. equalities.

\[
\tau_{\phi} = (\lambda \ d. \text{decide}(d; u.u(\lambda \ m. \text{inl}(m))); v.v(\lambda \ n. \text{inr}(n)))
\]
\[
\tau_{\phi}^{\text{inl}(E_1)} = (\lambda \ m. \text{inl}(m))^E_1
\]
\[
\tau_{\phi}^{\text{inr}(E_1)} = (\lambda \ n. \text{inr}(n))^E_1
\]
\( \phi \equiv P \land Q \): where \( P, Q \) are atomic.

\[
\tau_\phi \equiv \lambda p.\text{spread}(p; u, v.u(\lambda m.v(\lambda n.(m, n))))
\]

\[
\tau_\phi(E_1, N) = (\lambda m.(N : \lambda n.(m, n)))^E_1
\]

\[
\tau_\phi(V, E_1) = (\lambda n.(V, n))^E_1
\]

**Lemma 9.10.4 (Evaluation Context Rewrapping Lemma)**

If \( E[M] : \tau_\phi = M : \tau_\phi^E \), and \( M : \tau_\phi^E \rightarrow N : \tau_\phi^E \), then \( N : \tau_\phi^E \rightarrow E[N] : \tau_\phi \).

**Proof:** By cases: 

- **Case \( N \) is not a value:** Then by a trivial argument on the structure of \( N : \tau_\phi^E \), we can conclude equality with \( E[N] : \tau_\phi \). The idea in this case is that \( M \rightarrow N \), but the evaluator doesn’t have to “refocus” its attention to some new redex, so there is nothing to do.

- **Case \( N \) a value, and the innermost term of \( E \) is a destructor:** Again, by a simple argument we can conclude the equality. The basic idea is to argue from the structure of the enclosing destructor, and from the well-typedness of \( N : \tau_\phi^E \). The idea here is that we just computed a value, and now we have to “back up” one term in the evaluation context in order to use the enclosing destructor to destruct that same value. Again, the evaluator doesn’t have to do anything to refocus, because of the structure of the colon-translation.

  - **Case \( N \) a value, \( E[N] \equiv E'[N(vR)] \):** Then

    \[
    E[M] : \tau_\phi \equiv M : \tau_\phi^E
    \]

    \[
    \rightarrow N : \tau_\phi^E
    \]

    \[
    \equiv (\lambda m.R(\lambda n.m(n)\tau_\phi^{E'}))\Psi(N)
    \]

    \[
    \rightarrow R(\lambda n.\Psi(N)n\tau_\phi^{E'})
    \]

    \[
    \rightarrow N(vR) : \tau_\phi^{E'}
    \]

    \[
    \equiv E[N] : \tau_\phi
    \]

  - **Case \( N \) a value, \( E[N] \equiv E'[V(vR)] \):**

    \[
    E[M] : \tau_\phi \equiv M : \tau_\phi^E
    \]

    \[
    \rightarrow N : \tau_\phi^E
    \]

    \[
    \equiv N : \lambda n.\Psi(V)n\tau_\phi^{E'}
    \]

    \[
    \equiv \Psi(V)\Psi(N)\tau_\phi^{E'}
    \]

    \[
    \equiv V(vN) : \tau_\phi^{E'}
    \]

    \[
    \equiv E[N] : \tau_\phi
    \]

  - **Case \( N \) a value, the innermost term of \( E \) is not a destructor, not a by-value application:** We have just computed a value, but there is no enclosing destructor waiting to take that value apart. This case only happens when \( \phi \), the top-level
formula, is something structured, e.g. an existential, rather than primitive, like an equality. So the argument will have to be from the structure of the evaluation context, for which we will need to know \( \phi \). So we do a case analysis on \( \phi \), and inside of that, a case analysis on \( E \):

\[
\phi \equiv \exists n \in \mathbb{N}. f(x) = 0: \text{Two cases:}
\]

\[
E[\ ] = [\ ]:
\]

\[
N : \tau^E_\phi = N : \tau_\phi
\]

\[
= \langle A^D, B \rangle : \tau_\phi
\]

\[
= \tau_\phi((A^D, \emptyset))
\]

\[
= (\lambda p. \text{spread}(p; m, v.v(\lambda n.(m, n))))((A^D, \emptyset))
\]

\[
= B(\lambda n.(A^D, n))
\]

\[
= B : \lambda n.(A^D, n)
\]

If \( B \) is not a value, we can conclude \( B : \lambda n.(A^D, n) = \langle A^D, B \rangle : \tau_\phi \). If not, then we continue with the next evaluation context case.

\[
E[\ ] = \langle A^D, [\ ] \rangle:
\]

\[
N : \tau^E_\phi = N : \lambda n.(A^D, n)
\]

\[
= (\lambda n.(A^D, n))N
\]

\[
= (A^D, N)
\]

\[
= \langle A^D, N \rangle : \tau_\phi
\]

\[
\phi \equiv P \lor Q: E[\ ] = [\ ]: \text{There are two cases here, one for each injection. We will do the left injection case.}
\]

\[
N : \tau^E_\phi \equiv N : \tau_\phi
\]

\[
= \lambda m.\text{inl}(m)
\]

\[
= A : \lambda m.\text{inl}(m)
\]

If \( A \) is not a value, then the last term is equal to \( \text{inl}(A) : \tau_\phi \), otherwise continue.

\[
E[\ ] = \text{inl}([\ ]):
\]

\[
N : \tau^E_\phi \equiv N : \lambda m.\text{inl}(m)
\]

\[
= \text{inl}(N)
\]

\[
= \text{inl}(N) : \tau_\phi
\]

\[
\phi \equiv P \land Q: E = [\ ]:
\]

\[
N : \tau^E_\phi \equiv N : \tau_\phi
\]

\[
= \lambda m.\text{B}(\lambda n.(m, n))
\]

\[
= A : \lambda m.B : \lambda n.(m, n)
\]

Again, if \( A \) is not a value, the last term is equal to \( \langle A, B \rangle : \tau_\phi \). Otherwise, continue.
\[ E = (\lambda \, \), B): \]

\[
N : \tau^E \equiv N : \lambda \, m. B : \lambda \, n.(m, n) \\
\rightarrow B : \lambda \, n.(N, n)
\]

and the same conditions hold here as for the previous case.

\[ E = (A, \lambda \, ) : \]

\[
N : \tau^E \equiv N : \lambda \, n.(A, n) \\
\rightarrow (A, N) \\
= (A, N) : \tau^\phi
\]

The whole point of the preceding lemma is that, after computing a value which is not going to be immediately destructed, we can refocus the evaluator to the next redex by administrative reduction. At this point, we must prove a trivial substitutivity lemma:

**Lemma 9.10.5 (Substitutivity)** If \( N \) is a closed value, and \( x \) not free in \( K \),

\[
M[N/x] \rightarrow M[N/x]
\]

and

\[
(M : K)[N/x] \rightarrow M[N/x] : K.
\]

**Proof:** An induction on the size of \( M \). The proof method is a structural induction, with the base case a proof that

\[
(x : K)[N/x] \rightarrow N : K.
\]

The reader will note that if the reducibility relation above were replaced by equality, the preceding lemma would be very, very false. The problem is that in some places we have terms like \( x : K \), where \( x \) is a variable. If \( x \) gets substituted for by \( N \), we have to do reductions to get to \( N : K \), since \( x : K = x(K) \), and \( N : K \) can be much more complicated. The colon-translation has a very nice property relating it to double-negation translation and continuations:

**Lemma 9.10.6 (Colon Characterization Lemma)** If \( M \) is of type \( T \), and \( K \) is of type \( K^\phi(T) \), then \( M : K \) is of type \( \phi \).

**Proof:** By induction on the translation. Trivial, and omitted.
9.10.3 The Main Proof

Now we come to the meat of the problem:

**Theorem 9.10.2 (Colon Mimics Reduction)** If $M$ is a witness for $\phi$, and $M \rightarrow_1 N$, then

$$M : \tau_\phi \rightarrow N : \tau_\phi.$$ 

**Proof:** By cases on the reduction rules. The pattern is to use the evaluation context unwrapping lemma to unwrap an evaluation context, uncovering the redex, reduce the redex by a direct argument, and then use the evaluation context rewrapping lemma to convert that back to a top-level application of colon. All we need do is verify that, when $M$ is a redex, $M \rightarrow_1 N$ implies $M : K \rightarrow N : K$. This we will do for a few cases:

**$\beta$-reduction:** Show: $(\lambda x.b)(t) : K \rightarrow b[t/x] : K$:

$$
(\lambda x.b)(t) : K = \lambda x.b : \lambda m.(t)K \\
= (\lambda m.m(t))(\lambda x.b) \\
\rightarrow (b[t/x])(K) \\
\rightarrow (b[t/x] : K)
$$

**Base Case of $\text{ind}$ Form:** Show: $\text{ind}(0; b; n, i.\text{I}(n)(i)) : K \rightarrow b : K$:

$$
\text{ind}(0; b; n, i.\text{I}(n-1)(i)) : K = (\text{ind}(0; b; n, i.\text{I}(n)(i-1)))(K) \\
\rightarrow bK \\
= (\lambda k.b : k)(K) \\
\rightarrow b : K
$$

**Inductive Step of $\text{ind}$ Form:**

Show:

$$
\text{ind}(E; B; n, i.\text{I}(n-1)(i)) : K \\
\rightarrow I(E-1)\text{ind}(E-1; B; n, i.\text{I}(n-1)(i)) : K
$$

$$
\text{ind}(E; B; n, i.\text{I}(n-1)(i)) : K \\
= \text{ind}(E; b; i, n.\text{I}(n-1)(i))K \\
\rightarrow (I(n)(i)|E-1, \text{ind}(E-1; b; i, n.\text{I}(n-1)(i))/n, i))(K) \\
\rightarrow (\lambda k.I(E-1)(\text{ind}(E-1; b; i, n.\text{I}(n-1)(i)) : k)(K) \\
\rightarrow I(E-1)\text{ind}(E-1; b; i, n.\text{I}(n-1)(i)) : K
$$

**Spread Terms:** Show: $\text{spread}((M, N); u, v.t) : K \rightarrow t[M, N/u, v] : K$: 

...
\[
\text{spread}((M, N); u, v.T) : K = (M, N) : \lambda p.\text{spread}(p; u, v.T(K)) \\
= (\lambda p.\text{spread}(p; u, v.T(K)))(M, N) \\
\rightarrow (T[M, N/u, v])(K) \\
\rightarrow (T[M, N/u, v])(K) \\
\rightarrow T[M, N/u, v] : K
\]

**By-Value Applications:** Show: \((\lambda^v x. M)(vV) : K \rightarrow M[V/x] : K:\)

\[
(\lambda^v x. M)(vV) : K = \Psi(\lambda^v x. M)\Psi(V)K \\
\equiv (\lambda^v x. M)\Psi(V)K \\
\rightarrow MK \ (x \ \text{not free in} \ M) \\
\rightarrow M : K \\
\equiv M[V/x] : K
\]

There is only one step left before we can conclude semantic equivalence - we have yet to connect the colon-translation and the CPS-translation. To do so, we must prove a lemma showing that \(M\tau_\phi \rightarrow M : \tau_\phi\). Because we added special colon-translation rules for the cases where we had constructors at top-level, our proof must necessarily use information about the top-level type of the program. So we will prove the lemma in two parts, one for a general continuation \(K\) and one specialized to the top-level cases.

**Lemma 9.10.7 (Soundness of Colon)** If \(M\) is a term of type \(T\), in a program of type \(\phi\), \(K\) is a value, then \(MK \rightarrow M : K\) and \(M \rightarrow M\).

**Proof:** Again, an induction on the structure of \(M\). We have to be careful and distinguish the cases where \(M\) is a top-level constructor, and \(K\) is \(\tau_\phi\), or some value derived from it via the rules for colon-translation for \(RC\). We do the proof for some of the cases below. For the constructors, as pointed out, we need to know the value and type of the continuation \(K\). We will do the three cases we have done all long, for a disjunction, conjunction, and existential.

**Lambda Terms:**

\[
\lambda x. M = (\lambda k.k(\lambda x. M)) \\
\rightarrow \lambda k.k(\lambda x. M) \\
= \lambda k.((\lambda x. M) : k) \\
= \lambda x. M
\]

\[
\lambda x. MK \rightarrow \lambda x. MK \\
\rightarrow \lambda x. M : K
\]
9.10 Semantic Equivalence: Value-Preserving CPS-Compilation

Spread Terms:

\[ \text{spread}(M; u, v.N) = \lambda k. M(\lambda p. \text{spread}(p; u, v.N(k))) \]
\[ \quad \rightarrow \lambda k. M : \lambda p. \text{spread}(p; u, v.N : (k)) \]
\[ = \text{spread}(M; u, v.N) \]
\[ \text{spread}(M; u, v.N)K = (\lambda k. M(\lambda p. \text{spread}(p; u, v.N(k))))K \]
\[ \quad \rightarrow M : \lambda p. \text{spread}(p; u, v.N : (K)) \]
\[ = \text{spread}(M; u, v.N) : K \]

Applications \( M(N) \):

\[ M(N) = \lambda k. M(\lambda m.mNk) \]
\[ \quad \rightarrow \lambda k. M(\lambda m.mNk) \]
\[ \quad \rightarrow \lambda k. M : \lambda m.mNk \]
\[ = \lambda k. M(N) : k \]
\[ = M(N) \]
\[ M(N)K = (\lambda k. M(\lambda m.mNk))K \]
\[ \quad \rightarrow M(\lambda m.mNK) \]
\[ \quad \rightarrow M : \lambda m.mNK \]
\[ = M(N) : K \]

Variables:

\[ xK = x(K) \]
\[ = x : K \]
\[ x = \text{\#} \]

\( C \):

\[ CM = \lambda k. M(\lambda m.m(\lambda g.g(\lambda v, h.v(k))))_{\tau_{\phi}} \]
\[ \quad \rightarrow \lambda k. (M : (\lambda m.m(\lambda g.g(\lambda v, h.v(k))))_{\tau_{\phi}}) \]
\[ = \lambda k. (CM : k) \]
\[ = CM \]
\[ CMK = (\lambda k. M(\lambda m.m(\lambda g.g(\lambda v, h.v(k))))_{\tau_{\phi}})K \]
\[ \quad \rightarrow M(\lambda m.m(\lambda g.g(\lambda v, h.v(K))))_{\tau_{\phi}} \]
\[ \quad \rightarrow M : (\lambda m.m(\lambda g.g(\lambda v, h.v(K))))_{\tau_{\phi}} \]
\[ = CM : K \]

\( A \):

\[ AM = \lambda k. M_{\tau_{\phi}} \]
\[ \quad \rightarrow \lambda k. M : \tau_{\phi} \]
\[ = \lambda k. AM : k \]
\[ = AM \]
\[ AMK = (\lambda k. M_{\tau_{\phi}})K \]
\[ \quad \rightarrow M_{\tau_{\phi}} \]
\[ \quad \rightarrow M : \tau_{\phi} \]
\[ = AM : K \]
By-Value Lambda Terms:

\[
\begin{align*}
\lambda^v \ x. M & \quad = \quad (\lambda \ k.(\lambda \ x. M)) \\
& \quad \to \quad \lambda \ k.(\lambda^v \ x. M) \\
& \quad = \quad \lambda \ k.(\lambda^v \ x. M : k) \\
& \quad = \quad \lambda^v \ x. M \\
\lambda^v \ x. M K & \quad \to \quad \lambda^v \ x. M K \\
& \quad \to \quad \lambda^v \ x. M : K 
\end{align*}
\]

By-Value Applications \textit{M}(\textit{N}): Two cases:

\textit{M} not a value:

\[
\begin{align*}
\textit{M}(vN) & \quad = \quad \lambda \ k.\textit{M}(\lambda \ m.\textit{N}\lambda \ n.m(n)k) \\
& \quad \to \quad \lambda \ k.\textit{M}(\lambda \ m.(\textit{N} : \lambda \ n.m(n)k)) \\
& \quad \to \quad \lambda \ k.\textit{M} : \lambda \ m.(\textit{N} : \lambda \ n.m(n)k) \\
& \quad = \quad \textit{M}(vN) 
\end{align*}
\]

\textit{M} a value = \lambda^v \ x. \textit{M'}:

\[
\begin{align*}
(\lambda^v \ x. \textit{M'})(vN) & \quad = \quad \lambda \ k.(\lambda^v \ x. \textit{M'})(\lambda \ m.\textit{N}\lambda \ n.m(n)k) \\
& \quad \to \quad \lambda \ k.(\textit{N}(\lambda \ n.\Psi(\lambda^v \ x. \textit{M'})(n)k)) \\
& \quad \to \quad \lambda \ k.\textit{N} : (\lambda \ n.\Psi(\lambda^v \ x. \textit{M'})(n)k)) \\
& \quad = \quad \lambda \ k.\lambda^v \ x. \textit{M'}(vN) : k \\
& \quad = \quad (\lambda^v \ x. \textit{M'})(vN) 
\end{align*}
\]

Existential Pairs \textit{(M}, \textit{N}):

\[
\begin{align*}
\textit{(M}, \textit{N}) & \quad = \quad \lambda \ k.\textit{k}(\textit{(M}, \textit{N})) \\
& \quad \to \quad \lambda \ k.\textit{k}(\textit{(M}, \textit{N})) \\
& \quad = \quad \textit{(M}, \textit{N}) 
\end{align*}
\]

To prove \textit{(M}, \textit{N})K \to \textit{(M}, \textit{N} : K, we must know the value of \textit{K}. If \textit{K} is not \textit{r}_{\phi}, then we can easily show \textit{(M}, \textit{N})K \to \textit{(M}, \textit{N} : K because the only rules for colon-translation which care about the continuation provided them are the ones which expect \textit{K} = \textit{r}_{\phi}. If \textit{K} = \textit{r}_{\phi}, \phi = \exists x \in \textit{N}.f(x) = 0, then we must argue from the form of \textit{N}:

\textit{N} is not a value:

\[
\begin{align*}
\textit{(M}, \textit{N})r_{\phi} & \quad = \quad (\lambda \ k.\textit{k}(\textit{(M}, \textit{N})))r_{\phi)} \\
& \quad \to \quad (\textit{(M}, \textit{N}))r_{\phi} \\
& \quad \to \quad \textit{N} : \lambda \ n.(\textit{M}, \textit{n}) \\
& \quad = \quad \textit{(M}, \textit{N}) : r_{\phi} 
\end{align*}
\]
9.10 Semantic Equivalence: Value-Preserving CPS-Compilation

\[ N = V_1 \text{ a value:} \]

\[ \langle M^D, N \rangle_{\tau_\phi} \rightarrow \langle M, V_1 \rangle \]

\[ = \langle M, N \rangle : \tau_\phi \]

Propositional Pairs \( \langle M, N \rangle \):

\[ \langle M, N \rangle = \lambda k.k((M, N)) \]

\[ \rightarrow \lambda k.k((M, N)) \]

\[ = \langle M, N \rangle \]

To prove \( \langle M, N \rangle K \rightarrow \langle M, N \rangle : K \), we must know the value of \( K \). If \( K \) is not \( \tau_\phi \), then we can show \( \langle M, N \rangle K \rightarrow \langle M, N \rangle : K \). If \( K = \tau_\phi \), where \( \phi = P \land Q \), then we must argue from the form of \( M, N \):

Neither \( M, N \) are values:

\[ \langle M, N \rangle_{\tau_\phi} = (\lambda k.k((M, N)))(\tau_\phi) \]

\[ \rightarrow M(\lambda m.N(\lambda n.(m, n))) \]

\[ \rightarrow M : \lambda m.N : \lambda n.(m, n) \]

\[ = \langle M, N \rangle : \tau_\phi \]

\( M = V_1 \), a value, \( N \) is not:

\[ \langle M, N \rangle_{\tau_\phi} = (\lambda k.k((V_1, N)))(\tau_\phi) \]

\[ \rightarrow N(\lambda n.(V_1, n)) \]

\[ \rightarrow N : \lambda n.(V_1, n) \]

\[ = \langle M, N \rangle : \tau_\phi \]

Both \( M, N \) are values:

\[ \langle M, N \rangle_{\tau_\phi} \rightarrow \langle M, N \rangle \]

\[ = \langle M, N \rangle : \tau_\phi \]

\[ \]

We need one final, simple lemma:

Lemma 9.10.8 (Reduction Preserves Typing) If

\[ \vdash_{HA} \phi \text{ ext } M \]

and \( M \rightarrow_1 N \), then

\[ \vdash_{HA} \phi \text{ ext } N \]

\[ \]

Proof: Trivial - this is the content of the standard SN proof for HA. \[ \]

Finally, we can formulate the major result of this chapter:
Theorem 9.10.3 (CPS Preserves Operational Semantics) For any program $M$ of type $\phi$, if $\phi$ is a printable type, and there exists a top-level continuation $\tau_\phi$ (i.e. such that for values $V \in \phi$, $\bar{V}_{\tau_\phi} = V$, then for all values $b \in \phi$,

$$M \rightarrow b \iff M_{\tau_\phi} \rightarrow b.$$  

Proof: Case $\Rightarrow$: Trivial. Invoke the soundness of CPS-translation to conclude that $M_{\tau_\phi} = b_{\tau_\phi}$. Then, since $b_{\tau_\phi} \rightarrow b$, conclude $M_{\tau_\phi} \rightarrow b$.

Case $\Leftarrow$: Use the colon soundness lemma to show $M_{\tau_\phi} \rightarrow M : \tau_\phi$. Then by the well-typed-ness of $M$, and the preservation of typing under reduction, conclude that every reduction sequence on $M$, if it terminates, must do so in a value. By the fact that reduction on $M$ is mimicked by reduction on $M : \tau_\phi$, conclude that there are no infinite reduction sequences on $M$. Therefore $M \rightarrow b$. 

We can also prove the following corollary, tying together everything in this chapter:

Corollary 9.10.1 (CPS is a Value-Preserving Embedding) Given a $\Pi^0_2$ formula, $\Phi \equiv \forall x \in \mathbb{N}. \exists y \in \mathbb{N}. f(x, y) = 0$, and a proof

$$\vdash_{HA} \Phi \text{ ext } \lambda x. M,$$

we can effectively construct another proof,

$$\vdash_{HA} \Phi \text{ ext } \lambda x. (M_{\tau_\phi}).$$

Moreover, for every $n \in \mathbb{N}$,

$$(\lambda x.M)(n) \rightarrow b \iff (\lambda x.M_{\tau_\phi})(n) \rightarrow b.$$  

Proof: By appeal to the preceding lemma, and Corollary 9.9.3. 

9.11 Extensions: Primitive Recursive Function Symbols

We can extend the machinery of this chapter to account for other primitive recursive function symbols in addition to $S(\bullet), \times$, and $\oplus$. The basic idea is to write down the defining equations for these primitive recursive functions, and the substitutivity axioms, telling us when we can substitute equals for equals in the arguments to one of these functions. We prove these axioms in Nuprl, and then import them into the theory of HA directly, and we use the Nuprl constructive witness as the constructive witness for the axiom in HA.
9.12 A Recap of the Chapter

The work of this chapter has been to connect double-negation translation and CPS-translation in the context of classical logic.

We began with a simple example theory, $LJ \rightarrow$, and its programming language, the simply-typed lambda-calculus, and showed that the Kolmogorov translation upon sequent proofs in this logic corresponded exactly to the CPS-translation of the constructive witnesses of those theorems. We invoked work of Plotkin and Griffin on the simply-typed lambda-calculus to claim that any proof of a primitive type (i.e. a program which computes an integer) in the classical variant $LK \rightarrow$ was equal in the natural call-by-name operational semantics of this language to its CPS-translation, when applied to the proper top-level continuation, as is prescribed for CPS-translations.

We went on to extend this result to arithmetic (classical and constructive), and its programming language. We showed that the Kolmogorov translation from sequent proofs in PA to proofs in HA was identical to a particular CPS-translation applied to the classical witnesses of those proofs. Finally, we showed that the original program, $M$, a classical witness for the original proof of $\phi$, and the new program, $M(\tau_\phi)$ (the CPS-translation of $\phi$, applied to the proper top-level continuation), had the same value, and in addition, the new program exactly mimicked the operational semantics of the original program.

In the next chapter, we extend this analysis to other translations, and show that double-negation translation fixes the operational semantics of of evaluation. We also show some negative results, and extend this work to higher-order logics.
Chapter 10

Double-Negation Translation: Extensions

In this chapter, we extend the work of the last chapter to other translations, higher-order theories, and show some negative results. We investigate alternative translations, connecting different translation methodologies with different evaluation semantics. We demonstrate several pathologies associated with witnesses for PA programs, among them, non-Church-Rosser-ness, and their connections with evaluation semantics. We also discuss the properties of the colon-translation, and its connection with more efficient parsimonious proof transformation methods.

10.1 Other Translations and Evaluation Strategies

We will now discuss other translation methods, and the evaluation strategies they encode. First, let us recall the evaluation strategy that the Kolmogorov double-negation translation captures. As we mentioned before, the Kolmogorov translation is a call-by-name translation only for the propositional inhabitants of our programming language. It says nothing about the evaluation semantics of integer-valued expressions. In fact, in a sense that can be made precise, the Kolmogorov translation ignores the question of operational semantics for integer expression evaluation entirely, except it requires that, when an integer expression is printed (i.e. at toplevel) or inducted upon, it must be computed down to a numeral.

As we pointed out before, this means that if we take a classical program intended for a CBN evaluator, we can run the CPS-translation of that program in either a CBN or CBV evaluator, and expect that the program will follow the same reduction sequence, except for the reductions that happen upon integer expressions. If the CBV evaluator were lazy in the way it evaluated integer expressions, we could expect that the CBV evaluator, when applied to the CPS-translated program, would produce the same reduction sequence as the CBN evaluator would.
10.1 Other Translations and Evaluation Strategies

What the Kolmogorov translation does is produce a partial definitional interpreter for our programming language. Recall that a definitional interpreter for a programming language $\mathcal{L}$ is a translation of programs in $\mathcal{L}$ into another language $\mathcal{M}$, the metalanguage, such that the semantics of the language $\mathcal{L}$ do not depend upon the evaluation order of $\mathcal{M}$. We are dealing with a strongly normalizing metalanguage in this case, so the evaluation order of the metalanguage, HA, cannot affect the semantics of the defined language, PA. However, the results herein presented could (we believe) be extended to a language which is only normalizing, in which case one would want a definitional interpreter which exactly duplicated the reduction semantics of the language being interpreted. Regardless, there are many other reasons, esthetic, pedantic, pedagogical, and realistic, for wanting a partial definitional interpreter to be as faithful to the original language as possible.

Since our metalanguage and our defined language are both call-by-name, it turned out that our simulation of PA in HA exactly mimicked the evaluation rules of PA, even for integer expressions. To belabor an already weary point, if our metalanguage had been call-by-value, integer expression evaluation would not have been mimicked. This is a specific instance of a general problem: Can one specify interpreters for languages which have widely varying evaluation orders via translation into a fixed language, by specifying the translations upon types?

The answer turns out to be “yes”. The previously-specified Kolmogorov translation is lazy in its evaluation of constructors and functions, and is uncommitted in its evaluation of integer expressions. We will now define several other translations:

The Kuroda Translation: The Kuroda translation is a double-negation translation that conservatively extends HA to PA, and (it will turn out) CPS-translates $M$ into a program which, in either a CBV or CBN evaluator, mimicks call-by-value reduction rules, with eager constructors and function application. The Kuroda translation is uncommitted in its evaluation of integer expressions, and we can think of this translation as the natural counterpart of the Kolmogorov-translation.

The Pervasive Kolmogorov Translation: The pervasive Kolmogorov translation is a Kolmogorov translation that in addition translates integer expressions. Given a program $M$, it produces a CPS-translated program which, when run in either a CBN or CBV evaluator, mimicks CBN reduction on constructors, functions, and integer expressions. Thus we can think of this translation as the natural extension of the Kolmogorov translation to integer expressions.

The (Nonexistent) Pervasive Kuroda Translation: We would like to exhibit a translation that, given a program $M$, returns a program that, when run in either a CBN or CBV evaluator, will mimick call-by-value semantics on both structured data and integers. Unfortunately, there is no such program, for the same reasons that Plotkin in his seminal paper [Plo75] used to show that one needed a special call-by-name language in order to simulate a call-
by-value language in it; basically, one cannot simulate call-by-value addition with call-by-name addition in a neat way.

The general principle being demonstrated with these translations is we can specify evaluation strategies for the lambda-calculus by specifying a translation upon types, and then specifying a compatible translation upon terms. In the next section, we will discuss this idea on a more general level.

10.1.1 The Kuroda Translation: Call-by-Value

Griffin studied a CPS-translation that mimicked call-by-value evaluation order for the simply-typed lambda-calculus. It turns out that his double-negation translation was in fact a variant of the Kuroda negative translation, restricted to simple types. We will sketch the Kuroda translation on types and on programs, and call upon results of Griffin, Plotkin and Chapter 9 to show that, in fact, the resulting programs are imbued with a call-by-value evaluation order. Again, however, the Kuroda translation fails to translate integer expressions in a meaningful way, so if a Kuroda-translated program were to be evaluated in a call-by-name evaluator, though all the constructors would be evaluated in a call-by-value fashion, the integer expressions would still be evaluated call-by-name.

We proceed by defining the Kuroda translation on formulas and sequents, the compatible CPS-translation upon programs, showing that the CPS-translation is compatible with the Kuroda translation, and arguing for the semantic equivalence of an original program with its Kuroda-translation.

In the following, we assume evaluation is by-value (eager, applicative) everywhere (for applications and all constructors and destructors). We will not develop the definitions and theory needed to carefully formalize this, due to lack of space, but it is a simple reworking of the call-by-name evaluation semantics.

The original Kuroda translation upon formulas (types) is very simple; define

\[
\begin{align*}
(A \, op \, B)^* & \equiv A^* \, op \, B^* \,(op \in \{\lor, \land\}) \\
(\exists x \in N. \Phi(x))^* & \equiv \exists x \in N. (\Phi(x))^* \\
(A \, \Rightarrow \, B)^* & \equiv A^* \, \Rightarrow \, B^* \\
(\forall x \in N. \Phi(x))^* & \equiv \forall x \in N. \Phi(x) \\
P^* & \equiv P \text{ (P prime)} \\
\bot^* & \equiv \bot \\
\overline{T} & \equiv \neg (T^*)
\end{align*}
\]

Of course, we would then apply the A-translation, substituting \(\phi\), the top-level printable type, for \(\bot\). Instead, we will use the slightly modified translation in which \((A \Rightarrow B)^* \equiv A^* \Rightarrow B^*\):
\( (A \text{ op } B)^* \equiv A^* \text{ op } B^* \) (\text{op} \in \{\lor, \land\})
\( (\exists x \in \mathbb{N}. \Phi(x))^* \equiv \exists x \in \mathbb{N}. (\Phi(x))^* \)
\( (A \Rightarrow B)^* \equiv A^* \Rightarrow B \)
\( (\forall x \in \mathbb{N}. \Phi(x))^* \equiv \forall x \in \mathbb{N}. \Phi(x) \)
\( P^* \equiv P \) (\text{P prime})
\( \bot^* \equiv \phi \)
\( \overline{T} \equiv \overline{\phi}(T^*) \)

We call \((-)^*\) the star-translation, and \((-)\) the Kuroda translation. Put simply, the Kuroda translation consists of double-\(\phi\)-ating the outside of a term, and the range (rhs) of every universal and implication.

The compatible CPS-translation upon terms is:

\[
\begin{align*}
\overline{x} & \equiv \lambda k.k(x) \ (x \ a \ variable) \\
\overline{x}^D & \equiv x \\
\overline{MN} & \equiv \lambda k.M(\lambda m.N(\lambda n.m(n)k)) \\
\overline{M(N^D)} & \equiv \lambda k.M(\lambda m.m(N^D)k) \\
\overline{\lambda x.M} & \equiv \lambda k.k(\lambda x.M) \\
\overline{\lambda y^D.M} & \equiv \lambda k.k(\lambda y.M) \\
\overline{(M, N)} & \equiv \lambda k.M(\lambda m.N(\lambda k(m, n))) \\
\overline{(M^D, N)} & \equiv \lambda k.N(\lambda n.k(M^D, n)) \\
\overline{\text{spread}(M; u, v.T)} & \equiv \lambda k.M(\lambda p.\text{spread}(p; u, v.T)k)) \\
\overline{\text{spread}(M; u^D, v.T)} & \equiv \lambda k.M(\lambda p.\text{spread}(p; u^D, v.T)k)) \\
\overline{\text{inl}(M)} & \equiv \lambda k.\text{inl}(M) \\
\overline{\text{inr}(M)} & \equiv \lambda k.\text{inr}(M) \\
\overline{\text{decide}(M; u^N, v.R)} & \equiv \lambda k.M(\lambda d.\text{decide}(d; u^Nk, v.R)k)) \\
\overline{\text{ind}(M; B; n, i.I(n)(i))} & \equiv \text{ind}(M; B; n, i.I(n - 1)(i)) \\
\overline{\text{axiom}} & \equiv \lambda k.k(\text{axiom}) \\
\overline{\zeta M} & \equiv \lambda k.M(\lambda m.m(\lambda v, h.k(v))(\lambda x.x)) \\
\overline{\lambda M} & \equiv \lambda k.(M(\lambda x.x))
\end{align*}
\]

We can now define the Kuroda translation of a sequent:

**Definition 10.1.1 (Kuroda Translation of Sequent)**
The Kuroda Translation of a sequent \(\Gamma \vdash \phi\) is the sequent \(\Gamma^* \vdash \overline{\phi}\), where \(\Gamma^*\) is \(\Gamma\) with the star-translation (not the Kuroda translation) applied to every propositional hypothesis.

The next step is to prove that the Kuroda translation is a conservative extension. In the interests of brevity, we will only sketch the result.

**Theorem 10.1.1 (Translation Property of the Kuroda Translation)**
Given a proof

\[ \Gamma \vdash_{PA} T \text{ ext } M \]
we can construct a proof
\[ \Gamma^* \vdash_{HA} \overline{T} \text{ ext } M, \]
where the \( M \) is the CPS-translation just defined.

**Proof:** By induction on size of proofs and cases on the rules of HA. We will do it instead by showing that we can consistently transfer typings from the original witness \( M \) to the translated witness \( \overline{M} \). Thus, we will show that for each of the possible forms that \( M \) can take, the CPS-translated form \( \overline{M} \) has the proper type. The inductive assumption is that the CPS-translations of all subterms have types which are Kuroda-translations of their original types.

**Integer Variables** \( x^D \): \( M \) can never be a simple integer variable.

**Propositional Variables** \( x \): Classical Typing:
\[ x : T \]

**Constructive Typing:**
\[ \lambda \ k. k(x) : \neg \neg (T^*) \]

**Applications** \( MN \): Typings are:

**Classical:**
\[
\begin{align*}
M &: A \Rightarrow B \\
N &: A \\
MN &: B 
\end{align*}
\]

**Constructive:**
\[
\begin{align*}
M &: \neg \neg (A^* \Rightarrow \overline{B}) \\
N &: A \equiv \neg \neg (A^*) \\
MN &= \lambda \ k. M(\lambda \ m. N(\lambda \ n. m(n)k)) : \overline{B} \\
k &: \neg (B^*) \\
m &: A \Rightarrow B^* \equiv A^* \Rightarrow \overline{B} \\
n &: A^* 
\end{align*}
\]

**Applications** \( M(N^D) \): Typings are:

**Classical:**
\[
\begin{align*}
M &: \forall x \in N. \Phi(x) \\
N^D &: N \\
M(N^D) &: \Phi(N^D) 
\end{align*}
\]
10.1 Other Translations and Evaluation Strategies

Constructive:

\[
\begin{align*}
M & : \neg(\forall x \in \text{N.} \Phi(x)) \\
M(N^D) & \equiv \\
\lambda k.M(\lambda m. m(N^D)k) & : \Phi(N^D) \\
k & : \neg(\Phi(N^D)^*) \\
m & : \forall x \in \text{N.} \Phi(x) \\
n & : \text{N}
\end{align*}
\]

Lambda Expressions \(\lambda x.M\): If

\[\Gamma^*, x : A^* \vdash HA \overline{T} \text{ ext } M,\]

then

\[\Gamma^* \vdash HA \neg(A^* \Rightarrow \overline{T}) \text{ ext } \lambda k.k(\lambda x.M)\]

Propositional Pairs \(\langle M, N \rangle\): Classical:

\[
\begin{align*}
M & : A \\
N & : B
\end{align*}
\]

Constructive:

\[
\begin{align*}
M & : \overline{A} \\
N & : \overline{B} \\
\langle M, N \rangle & \equiv \\
\lambda k.M(\lambda m. N(\lambda n. k(m, n))) & : \neg(\overline{A^*} \land \overline{B^*}) \\
k & : \neg(\overline{A^*} \land \overline{B^*}) \\
m & : A^* \\
n & : B^*
\end{align*}
\]

Existential Pairs \(\langle M^D, N \rangle\): Classical:

\[
\begin{align*}
M^D & : \text{N} \\
N & : \Phi(M^D) \\
\langle M^D, N \rangle & : \exists x \in \text{N.} \Phi(x)
\end{align*}
\]

Constructive:

\[
\begin{align*}
N & : \overline{\Phi(M^D)} \\
\langle M^D, N \rangle & \equiv \\
\lambda k.N(\lambda n. k(M^D, n)) & : \neg(\exists x \in \text{N.} \Phi(x)^*) \\
k & : \neg(\exists x \in \text{N.} \Phi(x)^*) \\
n & : \Phi(M^D)^* 
\end{align*}
\]
Propositional Pair Elim \(\text{spread}(M; u, v.T):\) Classical:

\[
\begin{align*}
M &: (A, B) \\
T &: C \ (\text{where } u : A, v : B) \\
\text{spread}(M; u, v.T) &: C
\end{align*}
\]

Constructive:

\[
\begin{align*}
M &: \overline{A \land B} \equiv \neg \neg (A^* \land B^*) \\
T &: \overline{C} \ (\text{where } u : A^*, v : B^*) \\
\text{spread}(M; u, v.T) &\equiv \\
\lambda \ k. M(\lambda \ p. \text{spread}(p; u, v.Tk)) &: \overline{C} \\
k &: \neg (C^*) \\
p &: A^* \land B^* \\
u &: A^* \\
V &: B^*
\end{align*}
\]

Existential Pair Elim \(\text{spread}(M; u^D, v.T):\) Classical:

\[
\begin{align*}
M &: \exists x \in N. \Phi(x) \\
T &: C \ (\text{where } u : N, v : \Phi(u)) \\
\text{spread}(M; u, v.T) &: C
\end{align*}
\]

Constructive:

\[
\begin{align*}
M &: \overline{\exists x \in N. \Phi(x)} \equiv \neg \neg (\exists x \in N. \Phi(x)^*) \\
T &: \overline{C} \ (\text{where } u : N, v : \Phi(u)^*) \\
\text{spread}(M; u^D, v.T) &\equiv \\
\lambda \ k. M(\lambda \ p. \text{spread}(p; u^D, v.Tk)) &: \overline{C} \\
k &: \neg (C^*) \\
p &: \exists x \in N. \Phi(x)^* \\
u &: N \\
v &: \Phi(u)^*
\end{align*}
\]

\textit{inl, inr, and decide:} As for pairs and spreads.

\textbf{Induction Form:} Classical:

\[
\begin{align*}
B &: \Phi(0) \\
I &: \forall n \in N. \Phi(n) \Rightarrow \Phi(n + 1) \\
M^D &: N \\
\text{ind}(M^D; B; n, i. I(n - 1)(i)) &: \Phi(M^D)
\end{align*}
\]
Constructive:

\[ B : \Phi(0) \]
\[ I : \forall n \in \mathbb{N}. \Phi(n) \Rightarrow \Phi(n + 1) \]
\[ \text{ind}(M^D; B; n, i. I(n - 1)(i)) : \Phi(M^D) \]
\[ n : \mathbb{N} \quad (n > 0) \]
\[ i : \Phi(n - 1) \]
\[ I(n)(i) : \Phi(n) \]

\( C \) : Classical:

\[ M : \neg \neg (T) \]
\[ CM : T \]

Constructive:

\[ M : \phi \neg \neg (T) \equiv \phi \neg \neg ((T^* \Rightarrow \phi \neg \neg (\phi)) \Rightarrow \phi \neg \neg (\phi)) \]

\[ CM \equiv \lambda k. M(\lambda m. m(\lambda v, h. k(v))(\lambda x. x)) : T \]
\[ \quad k : \neg \neg (T^*) \]
\[ \quad m : (T^* \Rightarrow \neg \neg (\phi)) \Rightarrow \neg \neg (\phi) \]
\[ \quad (\lambda v, h. k(v)) : T^* \Rightarrow \neg \neg (\phi) \]
\[ \quad v : T^* \]
\[ \quad h : \neg (\phi) \]

\[ \]

**Theorem 10.1.2 (Conservative Extension)** If \( \phi \) is a printable type, and

\[ \vdash PA \phi \text{ ext } M \]

then the Kuroda translation generates

\[ \vdash HA \phi \text{ ext } M \]

and

\[ \vdash HA \phi \text{ ext } M(\lambda x. x). \]

**Proof:** Trivial. Simply note that \( \phi \) has neither universals nor implications; hence, \( \bar{\phi} = \phi \neg \neg (\phi) \); hence \( M(\lambda x. x) : \phi \). \( \]

The next step is to show that Kuroda translation of a proof/witness of a \( \Sigma_1^0 \) sentence computes to the same value as the original proof/witness. This is done in the same manner as before, and we will not go into the details. The reader will note that this time, a proof of \( \phi = \exists x \in \mathbb{N}. f(x) = 0 \) is translated to \( \neg \neg (\exists x \in \mathbb{N}. f(x) = 0) \).

Thus, the proper top-level continuation is always \( \lambda x. x \). We can prove, using the methods developed in the previous chapter, that
Theorem 10.1.3 (Semantic Equivalence for the Kuroda Translation)
If $M$ is a classical witness for a $\Sigma^0_1$ sentence $\phi$, then $M(\lambda x.x)$ is a constructive witness for $\phi$, and moreover for every value $b \in \phi, M \to b$ under by-value evaluation rules if and only if $M(\lambda x.x) \to b$.

Proof: Analogously to the proof for the Kolmogorov Translation. Special care must be taken in the definition of the colon-translation to account for the call-by-value reduction rules. ■

And finally the last theorem:

Theorem 10.1.4 (Value-Preserving Conservative Extension)
The Kuroda translation is a value-preserving conservative extension of HA to PA for $\Sigma^0_1$ sentences under by-value evaluation semantics.

Proof: By composing the two previous proofs. ■

Corollary 10.1.1 (Value-Preserving Conservative Extension for $\Pi^0_2$)
The Kuroda translation is a value-preserving conservative extension of HA to PA for $\Pi^0_2$ sentences under by-value evaluation semantics.

Note that the proof of Theorem 10.1.3 will require proving lemmas similar to the evaluation context unwrapping and rewrapping lemmas 9.10.2 and 9.10.4, the colon soundness lemma 9.10.7, and the colon mimicking lemma 9.10.2. The proof of the unwrapping lemma in particular is difficult, because the problems of “re-focusing” the evaluator happen again and again. For example, if we evaluate $spread((I(1), I(axiom)); u, v.v)$, where $I = \lambda x.x$, call-by-value, in the top-level context, we will see the following sequence of reduction steps:

$$
spread((I(1), I(axiom)); u, v.v): \lambda x.x = (I(1), I(axiom)) : K_1 \\
= I(1) : K_2 \\
\to 1 : K_2
$$

Unfortunately, we must convert this to $spread((1, I(axiom)); u, v.v): \lambda x.x$, but this is difficult, because we have to refocus the evaluator from the first component of the pair to the second. This involves a context rewrapping lemma. The proof is not hard, but it must be done for each and every constructor, several destructors, and also at top-level.

10.1.2 The Pervasive Kolmogorov Translation

The pervasive Kolmogorov translation is an extension of the standard Kolmogorov translation which CPS-translates integer expressions. We will only describe the translations, eschewing full proofs of type-correctness and semantic equivalence, as they would only be repetetive.
10.2 Evaluation, Typing, and Proof Translation

The idea behind the pervasive Kolmogorov translation is that, just as \( \neg\neg(f(x) = 0) \) is the type of implicit proofs of \( f(x) = 0 \), the type \( \neg\neg(N) \) should be the type of implicit proofs of \( N \). However, this does not work, because if we wrote down \( \neg\neg(\forall x \in \neg\neg(N) . \Phi(x)) \), the term \( \Phi(x) \) would be meaningless. The root of the problem is that one cannot "extract" the "implicit integer" from a proof of \( \neg\neg(N) \). However, if we chose instead the type \( \Delta N \equiv \Delta T : U_1 . (N \Rightarrow T) \Rightarrow T \) for our type of implicit integers, we could then write \( \neg\neg(\forall x \in \Delta N . \Phi(x(\lambda x . x))) \), and this expression would be well-typed. We must now define the CPS-translations of all the different expressions that can inhabit \( N \):

\[
\begin{align*}
n & \equiv \lambda \, k . k(n) \ (n \ a \ numeral) \\
A + B & \equiv \lambda \, k . A(\lambda \, m . B(\lambda \, n . k(n + m))) \\
A \times B & \equiv \lambda \, k . A(\lambda \, m . B(\lambda \, n . k(n \times m))) \\
S(A) & \equiv \lambda \, k . A(\lambda \, m . k(m + 1)).
\end{align*}
\]

We can prove that if \( E \in N \), then \( E \in \Delta N \). The double-negation translation of an equality, \( M = N \), is \( M(\lambda \, x . x) = N(\lambda \, x . x) \). The other rules of double-negation translation are the same, except, as noted before, we also translate \( N \) to \( \Delta N \).

One can prove that the double-negation translations of theorems of \( \text{PA} \) are theorems in \( \text{Nuprl} \) with the impredicative delta-type we make use of here, and that the original program extraction and the pervasively translated extraction are semantically equivalent, even down to the operational semantics of evaluation of integer expressions.

10.2 Evaluation, Typing, and Proof Translation

The Curry-Howard Isomorphism tells us that we can view the task of reducing (evaluating) a constructive witness as equivalent to the task of reducing the proof which it stands as witness for. That is, \( \beta \)-reduction upon a program can be viewed, and simulated by, \( \beta \)-reduction upon the proof from which we extracted that program. The results of this and the preceding chapter tell us, in essence, that we can do the same for proofs of \( \Sigma_1 \) sentences in Peano Arithmetic, if we fix the reduction strategy beforehand. That is, if we fix the order of evaluation of our PA programs, we can view reduction on PA programs as the same as reduction on PA proofs, and as reduction on the translated \( \text{HA} \) proofs. Now, as we showed before, the combination of the Kolmogorov translation on types (sentences) and the compatible Kolmogorov translation upon programs (proofs) performs a call-by-name (except for integer expressions) CPS-translation upon the programs. Likewise, the combination of the Kuroda translation upon types and the compatible Kuroda translation upon proofs (programs) performs a call-by-value CPS-translation upon the programs. We could prove an analogous result for the pervasive Kolmogorov translation.

The pattern we see is that, given a particular evaluation semantics for a language \( \mathcal{L} \), we can define a translation on types and a compatible translation upon programs...
which translates programs in $\mathcal{L}$ into $\mathcal{M}$ such that the translated program mimicks
the original evaluation semantics, and under any evaluation order in $\mathcal{M}$.

For instance, we could add call-by-value applications at the user level to our
language $\mathcal{L}$, the programming language of Peano Arithmetic, by adding a new type,
$\bullet \to \bullet$, its witness, $\lambda^v \ x.b$, and new destructor, $M_{(v)N}$. The evaluation rule will be:

$$\lambda^v \ x.b(vV) \to_1 b[V/x],$$

where $V$ is a value. The translation back into $\mathcal{M}$, the standard programming lan-
guage of Heyting Arithmetic, would be (modified from the Kolmogorov translation
presented before):

Translation Upon Types:

$$A \to^v B \equiv \neg\neg(A \to B)$$

Translation Upon Terms:

$$\lambda^v x.M \equiv \lambda k.k(\lambda x.M)$$
$$M_{(v)N} \equiv \lambda k.M(\lambda m.N(\lambda n.m(\lambda k.k(n))k))$$

One can prove quite easily that $(\lambda^v x.b)(V) : K \to b[V/x] : K$. Looking at the
translation upon terms, this result is unsurprising, since the translation explicitly
computes the value of $M$, then the value of $N$, and then applies the value of $M$, $m$,
to the "implicit form" of the value of $N$, $\lambda k.k(n)$.

The reader will note that this differs from the previous by-value translation in
that

$$M_{(v)N} \equiv \lambda k.M(\lambda m.N(\lambda n.m(\lambda k.k(n))k)),$$

rather than $\lambda k.M(\lambda m.N(\lambda n.m(n)k))$. The difference is technical, and concerns
the inductive typing invariant that we wish to maintain. In the case of the original
Kolmogorov translation, and this translation of $\lambda^v$, the typing invariant we wish to
maintain is that whenever

$$\Gamma \vdash_{PA} T \text{ ext } M$$

we can show

$$\overline{\Gamma} \vdash_{HA} \overline{T} \text{ ext } M.$$

With the Kuroda translation, we wanted to show instead

$$\Gamma^* \vdash_{HA} \overline{T} \text{ ext } M.$$

The difference is that under this modification to the Kolmogorov-translation,
function-types still expect to be applied to "implicit" values, rather than explicit
values as in the Kuroda translation. This difference could be eliminated, but only
at some cost in the complexity of the translation.
One could add the entire set of call-by-value constructors and destructors to our call-by-name language in a like manner, and then, by selecting the proper ones, our intrepid programmer could get just the right amount of laziness for his/her application. The key to properly translating the program then becomes annotation of the original program so that we can identify which reductions must be evaluated in a call-by-value manner, and which in a call-by-name manner. Of course, as above, we do this annotation explicitly by marking both types and values with a $v$ if they are to be evaluated in an eager manner.

As we said before, the overall theme of these extensions is the encoding of evaluation strategies into types and proof translations. The utility of typings in capturing the notion of evaluation order, while still allowing us to express total correctness properties, is startling, and points the way to new ways of specifying evaluators and interpreters for applicative and lazy languages.

### 10.3 Negative Results

These past two chapters have been full of positive results. If only the picture were truly so bright. This section is about negative results. One might think, after reading the past two chapters, that classical logic was ready to be included into a type theory today. That we could program in Peano Arithmetic, with knowledge that if our specifications were $\Pi^0_2$, we could simply evaluate the classical programs in an evaluator which understood $C$ and $A$, and everything would be wonderful. Unfortunately, we can show that classical programs are not observationally congruent with their CPS-translations. That is, we will exhibit a classical program $I$, its CPS-translation, $M_{\tau_\phi}$, and a context $E[\ ]$, such that the two programs are equal in the empty program context, but have different values in the context $E$.

Thus we will conclude that extending, for instance, Nuprl, with classical reasoning in the most direct manner, e.g. allowing one to prove $\Pi^0_2$ sentences classically, would be unsound.

**Theorem 10.3.1 (Unsoundness)** Classical programs are not observationally congruent with their CPS-translations; if

$$\begin{align*}
\phi &= \exists n \in \mathbb{N} . n = 1 \\
M &= C \lambda k . \langle 0, C \lambda g . k((1, \text{axiom})) \rangle \\
E[\ ] &= \text{spread}(\ ); u, v, u)
\end{align*}$$

then

$$M = M_{\tau_\phi}$$

but

$$E[M] \neq E[M_{\tau_\phi}] .$$

**Proof:** This is easy to see: $E[M] \to 0$, while $E[M_{\tau_\phi}] \to 1$. $\blacksquare$
Moreover, we can show quite easily that there exists a classical program which, under two different evaluation strategies, yields two different values:

\[ \langle A(0, \text{axiom}), A(1, \text{axiom}) \rangle. \]

This program uses \( A \), but the reader should be able to satisfy herself that it could easily have been written using \( C \). In any case, the program when evaluated from left-to-right yields \( \langle 0, \text{axiom} \rangle \), and from right-to-left, \( \langle 1, \text{axiom} \rangle \). This difference is wholly due to the fact that we chose a different evaluation sequence, and programs with \( C \) may not have unique normal forms. However, as we have seen, every normal form of a classical program witnessing a \( \Pi_2^0 \) sentence is in fact a true witness for that sentence. Hence, we can only conclude that classical programs embody a kind of nondeterminism that is frozen by the top-level continuation, since it is the job of the top-level continuation to fix the order in which subterms of top-level constructors are normalized.

### 10.4 Extension to Higher-Order Theories

The extension of the work on translations to higher-order type theories, e.g. \( HA^\omega \) or some subset of Nuprl, is simple. We show how to double-negation translate some particular formalization of a comprehension axiom, and how to double-negate higher-order predicate applications. All of this is easy. The only important detail which we cannot miss is to make sure that every instance of a predicate is translated correctly. Let us assume our formalization of second-order arithmetic is as in [DT89], where they define second-order logic as a standard formulation of intuitionistic first-order logic without equality, with the addition of the \textit{axiom of full (impredicative comprehension)}:

\[ \exists X^n. \forall x_1, \ldots, x_n. A(x_1, \ldots, x_n) \leftrightarrow X^n(x_1, \ldots, x_n), \]

where \( A(x_1, \ldots, x_n) \) is any formula not containing \( X^n \) free. To prove the double-negation of this formula, we are reduced to showing that there exists a predicate symbol which \textit{comprehends} the formula \( A(x_1, \ldots, x_n)^* \). This is an instance of the comprehension axiom itself, so we are done.

We conjecture, with the substantial evidence of our work in Nuprl and our proof for Heyting Arithmetic behind us, that

**Conjecture 10.4.1.** (Value-Preserving Conservative Extension of \( HAS \) to \( PAS \)) One can consistently extend the Kolmogorov translation presented here in the past two chapters with the above impredicative comprehension axiom in such a way as to show that, for a call-by-name version of \( HAS \), and its classical extension, \( PAS \):

1. the Kolmogorov translation from \( PAS \) to \( HAS \) is a conservative extension for \( \Pi_2^0 \) sentences,
2. the above Kolmogorov translation corresponds exactly to a local CPS-translation which can be proven, using the machinery developed in this chapter, to preserve the values of programs of type $\Sigma_1^0$, hence also $\Pi_2^0$, and also the operational semantics of evaluation.

**Justification:** The translation of higher-order formulas is trivial. The only work to be done is to verify that the Kolmogorov-translation of the new rule corresponds to the CPS-translation of its content, which is trivial, and to check that CPS-translation still preserves the operational semantics of call-by-name evaluation. This, again, is easy. 

It should be a relatively simple matter to rework this result for $HA^\omega$, and for many other intuitionistic theories.

### 10.5 Conclusions

We have demonstrated that double-negation/A-translations on proofs and sentences specify and implement definitional interpreters of various sorts for programming languages based on the lambda-calculus. We showed how to provide a by-value double-negation translation and how to provide for mixed by-value and by-name evaluation. We discussed the fact that the lambda-calculus with $C$ and $A$ is not Church-Rosser, and that double-negation translations impose a predetermined evaluation order on terms, hence enforcing Church-Rosser properties.

Finally, we discussed the extension of these results to higher-order theories. In the next chapter, we tie together the many strands in this thesis, and sketch out directions for future research.
Chapter 11

Conclusions and Directions for Future Research

This thesis has been about the Gödel/Friedman double-negation/A-translation in the Nuprl proof development system and its use in translating the classical impredicative proof of Higman's Lemma, and about the knowledge concerning proof construction, translation, and programming that we gained via that effort.

We began by describing the classical impredicative proof of Higman's Lemma, and its connections with the background of mathematics and programming. We then sketched out the formalization of Higman's Lemma in classical, impredicative Nuprl, and outlined the places where we found it impossible to remain true to the original, informal argument. We then justified the impredicative concepts used in the formalization, i.e. the impredicative II-type. Then, we described the implementation of our proof translation system at a very high level, and went on to detail our ideas for a more efficient translation system, including the employment of computational metatheory and the harnessing of parallelism in proof translation.

We then turned to the "program" content of the double-negation/A-translation, and showed that it corresponded exactly to a continuation-passing-style translation on classical program extractions. We showed moreover that different double-negation/A-translations correspond exactly to different evaluation strategies for these classical programs.

We used this to show that for $\Pi^0_2$ sentences, a classical proof computed the same evidence as its double-negation translation, when the translation was compatible with the evaluation semantics we imposed upon the classical proof (program).

The major results of this thesis are three-fold:

1. First, we have shown that mechanical double-negation/A-translation is a valid and tractable way of extracting the computational content of classical proofs. We constructed a proof translation system, albeit inefficient, and used it to translate several nontrivial (actually, quite large) theorems of mathematics.

2. Second, we have discovered the equivalence, in total-correctness (and higher-
order) type theories, of the double-negation/A-translation and CPS-translation. We have shown that we can view program extracts from classical proofs as simply being programs in a nonfunctional, non-Church-Rosser programming language with explicit manipulation of the control stack. We found that different double-negation translations corresponded exactly with different CPS-translations, and that we could understand the process of double-negation translation as fixing an evaluation order on the extracts of classical proofs.

3. Finally, we have proven that impredicative II-quantification is sound in Nuprl, showing that predicative Nuprl can be soundly extended to an impredicative theory with impredicative II-types, in addition to the previous extensions by Mendler to inductive, co-inductive, and impredicative $\Delta$-types.

Our directions for future work are manifold. We can divide them up into roughly three different areas:

1. Extend the work done on the equivalence of double-negation translation and CPS-compile to encompass more type-constructors, in particular lists and streams. Along with this work we must find more efficient ways of engineering double-negation/A-translation, since the efficiency of the system is of paramount importance. The general idea here is to mimic what was done in Chapter 9, where we "guessed" the proper execution semantics of the classical axioms for equalities by working backward from the form of their double-negation translations. The major problem which we feel must be solved here is to reconcile noncomputational goals, e.g. the Nuprl "set" type, with classical reasoning.

2. Use the theorem we proved in Chapter 9 regarding the equivalence of CPS-compile and double-negation/A-translation, and begin developing programs in Classical Nuprl. This work is perhaps the most interesting, as we finally have the ability to give direct total-correctness proofs to programs with instances of $C$ and $A$. Thus we can reason in a total-correctness fashion about, for instance, catch/throw, escape, and other nonlocal control operations. Moreover, by slight modifications in the Nuprl logic (to bring its program extraction rules into conformance with those of our formulation of Heyting Arithmetic), we can dispense with the double-negation/A-translation phase, and directly execute the classical program extractions from classical Nuprl proofs. This work would almost certainly include finding consistent reasoning systems for more sophisticated control operators such as Felleisen's first-class prompt [Fel88] and the shift/reset operators of Filinski and Danvy [FD90].

3. Rebuild the translation system and use it to investigate foundational questions in mathematical logic and well-quasi-order theory concerning the constructive
sensibility of impredicative quantification and the proof-theoretic strength of various minimal bad sequence arguments. The work to be done here is perhaps the most interesting, challenging, and also least practical (as it seems to be with almost all pure science). This line of research depends critically upon developing a more sophisticated and efficient theorem-proving engine. We expect that the ideas we outlined in Chapter 8 will be relevant to the construction of such a system. A distributed version of the theorem-prover is almost certainly necessary, as is a tail-recursion-optimizing version of the evaluator. In addition, direct translation of tactic metalanguage statements, by first parsing them, would aid in efficient translation. The goal of this effort would be to classically formalize large portions of mathematics, and extract constructively valid proofs by translation.

4. Combining some of all of the above directions, another very interesting line of work is research into the intensional properties of programs extracted from minimal bad sequence arguments. Here, we would seek to

- generate the simplest classical minimal bad sequence argument possible
- extract its classical algorithmic witness
- by various intensional analysis techniques (yet to be invented), discover the actual algorithm behind the proof technique.

The intuitive reason making this work practicable and attractive is our joint work with James Russell on a direct, predicative, constructive proof of Higman's Lemma [RM90] (discovered previously by Simpson [?] and concurrently by Richman and Stolzenberg [?]). There, we gave a simple, constructive proof of Higman's Lemma that was arrived at directly, without recourse to minimal bad sequences, translations, or any other arguments. In fact, the proof was via a simple, doubly-nested induction argument. The translated proof, on the other hand, could be formalized in impredicative second-order arithmetic, and not in any predicative second-order fragment, in its current form. Thus, we would naturally ask the question: Could the impredicative proof somehow be converted (semi)automatically into the predicative proof. Of course, this question is completely open, and might even be ill-posed. Nevertheless, this area seems to be a perfect target for computer-aided theorem-proving, since mathematicians are quite daunted by the size of the theorems and constructive program extractions involved.

To sum up, our research into, on the one hand, nonconstructive reasoning, and on the other, nonfunctional programming, has shown us that the Godel/Friedman translation is in reality a compiler from a nonfunctional language to a pure functional language. We have outlined above several different lines of research, and all of these can be phrased in terms this idea of the translation being a compiler.
The research we propose for the future ranges from extending the range of applicability of the compiler, to developing totally-correct programs with the compilation system, to investigating the behaviour of certain pathological programs (Kruskal's Theorem), to investigating the properties of a particular programming vehicle (the minimal bad sequence argument). Our contribution to the understanding of these issues has been to provide a first step towards understanding total-correctness reasoning of programs with explicit control operators. We would not go so far as to say that we have solved the problem entirely, since there are many programs which we cannot give types to with the rules we have developed. Indeed, we have only taken the first halting steps towards reasoning directly about explicit control. In the future, we hope that other workers will develop this theory, extend the type-theoretic treatment of explicit control operators to useful type theories and programming languages, and investigate the new logics which will arise out of such an endeavour.
Bibliography


[Hue] The calculus of constructions: Documentation and user's guide. Technical Report 110, INRIA.


Appendix A

The Nuprl Rules

VOID

formation

$H \gg U; \text{ ext void by intro void}$
$H \gg \text{ void in } U; \text{ by intro}$

noncanonical

$H, z:\text{void} \gg T \text{ ext any}(z) \text{ by elim } z$

$H \gg \text{ any}(e) \text{ in } T \text{ by intro}$
$\gg e \text{ in void}$
INT

formation

goal \( H \gg U; \) ext \( \text{int by intro} \) \( \text{int} \)
\[
H \gg \text{int in } U; \text{by intro}
\]

canonical

\( H \gg \text{int ext } c \text{ by intro } c \)

\( H \gg c \text{ in int by intro where } c \text{ must be an integer constant.} \)

noncanonical

\( H \gg -t \text{ in int} \)
\[ \gg t \text{ in int} \]

\( H \gg \text{int ext } m \text{ op } n \text{ by intro op} \)
\[
\gg \text{int ext } m \\
\gg \text{int ext } n
\]

\( H \gg m \text{ op } n \text{ in int by intro} \)
\[
\gg m \text{ in int} \\
\gg n \text{ in int} \\
\text{where op must be one of } +, -, *, /, \text{ or mod.}
\]

\( H, x: \text{int}, H' \gg T \text{ ext ind}(x;y,z.t_d;t_b;y,z.t_u) \text{ by elim } x \text{ new } z[,y] \)
\[ y: \text{int}, y<0, z:T[y+1/x] \gg T[y/x] \text{ ext } t_d \]
\[ \gg T[0/x] \text{ ext } t_b \]
\[ y: \text{int}, 0<y, z:T[y-1/x] \gg T[y/x] \text{ ext } t_u \]

The optional new name must be given if \( x \) occurs free in \( H' \).

\( H \gg \text{ind}(e;x,y.t_d;t_b;x,y.t_u) \text{ in } T[e/z] \)

by intro \([\text{over } z . T] \) \([\text{new } u,v] \)

\[ \gg e \text{ in int} \]
\[ u: \text{int}, u<0, v:T[u+1/z] \gg t_d[u,v/x,y] \text{ in } T[u/z] \]
\[ \gg t_b \text{ in } T[0/z] \]
\[ u: \text{int}, 0<u, v:T[u-1/z] \gg t_u[u,v/x,y] \text{ in } T[u/z] \]

\( H \gg \text{int.eq}(a;b;t;t') \text{ in } T \text{ by intro} \)
\[ \gg a \text{ in int} \]
\[ \gg b \text{ in int} \]
\[ a=b \text{ in int} \gg t \text{ in } T \]
\[ (a=b \text{ in int})\rightarrow \text{void} \gg t' \text{ in } T \]
\( H \gg \text{less}(a;b;t;t') \) in \( T \) by intro
   >> \( a \) in int
   >> \( b \) in int
   \( a < b \gg t \) in \( T \)
   \( (a < b) \rightarrow \text{void} \gg t' \) in \( T \)

**computation**

\( H \gg \text{ind}(nt;x,y.t_d;t_b;x,y.t_u) = t \) in \( T \) by reduce 1 down
   >> \( t_d[nt,(\text{ind}(nt+1;x,y.t_d;t_b;x,y.t_u))/x,y] = t \) in \( T \)
   >> \( nt < 0 \) \( H \gg \text{ind}(nt;x,y.t_d;t_b;x,y.t_u) = t \) in \( T \) by reduce 1 down
   >> \( t_d[nt,(\text{ind}(nt+1;x,y.t_d;t_b;x,y.t_u))/x,y] = t \) in \( T \)
   >> \( nt < 0 \)

\( H \gg \text{ind}(zt;x,y./t_d;t_b;x,y.t_u) = t \) in \( T \) by reduce 1 base
   >> \( t_b = t \) in \( T \)
   >> \( zt = 0 \) in int

\( H \gg \text{ind}(nt;x,y.t_d;t_b;x,y.t_u) = t \) in \( T \) by reduce 1 up
   >> \( t_u[nt,(\text{ind}(nt-1;x,y.t_d;t_b;x,y.t_u))/x,y] = t \) in \( T \)
   >> \( 0 < nt \)

\( H \gg \text{int.eq}(a;a;t;t') = t'' \) in \( T \) by reduce 1
   >> \( t = t'' \) in \( T \)

\( H \gg \text{int.eq}(a;b;t;t') = t'' \) in \( T \) by reduce 1
   >> \( t' = t'' \) in \( T \)
   where \( a \) and \( b \) are canonical int terms, and \( a \neq b \).

\( H \gg \text{less}(a;b;t;t') = t'' \) in \( T \) by reduce 1
   >> \( t = t'' \) in \( T \)
   where \( a \) and \( b \) are canonical int terms such that \( a < b \).

\( H \gg \text{less}(a;b;t;t') = t'' \) in \( T \) by reduce 1
   >> \( t' = t'' \) in \( T \)
   where \( a \) and \( b \) are canonical int terms such that \( a \geq b \).
LESS

formation

\[ H \gg U; \text{ext } a < b \text{ by intro less} \]
\[ H \gg \text{int ext } a \]
\[ H \gg \text{int ext } b \]

\[ H \gg a < b \text{ in } U; \text{ by intro} \]
\[ H \gg a \text{ in int} \]
\[ H \gg b \text{ in int} \]

equality

\[ H \gg \text{axiom in } a < b \]
\[ H \gg a < b \]
LIST

formation

\[ H \gg U_i \text{ ext } A \text{ list by intro list} \]
\[ \gg U_i \text{ ext } A \]

\[ H \gg A \text{ list in } U_i \text{ by intro} \]
\[ \gg A \text{ in } U_i \]

canonical

\[ H \gg A \text{ list ext } nil \text{ by intro } nil \text{ at } U_i \]
\[ \gg A \text{ in } U_i \]

\[ H \gg nil \text{ in } A \text{ list by intro at } U_i \]
\[ \gg A \text{ in } U_i \]

\[ H \gg A \text{ list ext } h.t \text{ by intro} . \]
\[ \gg A \text{ ext } h \]
\[ \gg A \text{ list ext } t \]

\[ H \gg a.b \text{ in } A \text{ list by intro} \]
\[ \gg a \text{ in } A \]
\[ \gg b \text{ in } A \text{ list} \]

noncanonical

\[ H, x:A \text{ list}, H' \gg T \text{ ext list}\_\text{ind}(x; t_b; u, v, w.t_u) \]
\[ \text{ by elim } x \text{ new } w, u, v, u[v,x] \]
\[ \gg T[\text{nil/x}] \text{ ext } t_b \]
\[ u:A, v:A \text{ list, w:T[v/x]} \gg T[u.v/x] \text{ ext } t_u \]

\[ H \gg \text{list}\_\text{ind}(e; t_b; x, y, z.t_u) \text{ in } T[e/z] \]
\[ \text{ by intro [over } z.T] \text{ using } A \text{ list } [\text{new } u, v, w] \]
\[ \gg e \text{ in } A \text{ list} \]
\[ \gg t_b \text{ in } T[\text{nil/z}] \]
\[ A, v:A \text{ list, w:T[v/x]} \gg t_u[u, v, w/x, y, z] \text{ in } T[u.v/z] \]

computation

\[ H \gg \text{list}\_\text{ind}(\text{nil}; t_b; u, v, w.t_u) = t \text{ in } T \text{ by reduce 1} \]
\[ \gg t_b = t \text{ in } T \]

\[ H \gg \text{list}\_\text{ind}(a.b; t_b; u, v, w.t_u) = t \text{ in } T \text{ by reduce 1} \]
\[ \gg t_u[a, b, \text{list}\_\text{ind}(b; t_b; u, v, w.t_u)/u, v, w] = t \text{ in } T \]
UNION

formation

\[ H \triangleright A \mid B \text{ by intro union} \]
\[ \triangleright U_i \text{ ext } A \]
\[ \triangleright U_i \text{ ext } B \]
\[ H \triangleright A \mid B \text{ in } U_i \text{ by intro} \]
\[ \triangleright A \text{ in } U_i \]
\[ \triangleright B \text{ in } U_i \]

canonical

\[ H \triangleright A \mid B \text{ ext inl}(a) \text{ by intro at } U_i \text{ left} \]
\[ \triangleright A \text{ ext } a \]
\[ \triangleright B \text{ in } U_i \]
\[ H \triangleright \text{ inl}(a) \text{ in } A \mid B \text{ by intro at } U_i \]
\[ \triangleright a \text{ in } A \]
\[ \triangleright B \text{ in } U_i \]
\[ H \triangleright A \mid B \text{ ext inr}(b) \text{ by intro at } U_i \text{ right} \]
\[ \triangleright B \text{ ext } b \]
\[ \triangleright A \text{ in } U_i \]
\[ H \triangleright \text{ inr}(b) \text{ in } A \mid B \text{ by intro at } U_i \]
\[ \triangleright b \text{ in } B \]
\[ \triangleright A \text{ in } U_i \]

noncanonical

\[ H, z:A \mid B, H' \triangleright T \text{ ext decide}(z;x.t_l;y.t_r) \text{ by elim } z \text{ [new } x,y] \]
\[ x:A, z=\text{inl}(x) \text{ in } A \mid B \triangleright T[\text{inl}(x)/z] \text{ ext } t_l \]
\[ y:B, z=\text{inr}(y) \text{ in } A \mid B \triangleright T[\text{inr}(y)/z] \text{ ext } t_r \]
\[ H \triangleright \text{ decide}(e;x.t_l;y.t_r) \text{ in } T[e/z] \]
by intro [over } z.T] \text{ using } A \mid B \text{ [new } u,v] \]
\[ \triangleright e \text{ in } A \mid B \]
\[ u:A, e=\text{inl}(u) \text{ in } A \mid B \triangleright t_l[u/x] \text{ in } T[\text{inl}(u)/z] \]
\[ v:B, e=\text{inr}(v) \text{ in } A \mid B \triangleright t_r[v/y] \text{ in } T[\text{inr}(v)/z] \]
computation

\[ H \gg \text{decide}(\text{inl}(a); x.t_1; y.t_r) = t \text{ in } T \text{ by reduce 1} \]
\[ \gg t_1[a/x] = t \text{ in } T \]

\[ H \gg \text{decide}(\text{inr}(b); x.t_1; y.t_r) = t \text{ in } T \text{ by reduce 1} \]
\[ \gg t_r[b/y] = t \text{ in } T \]
FUNCTION

formation

\[ H \gg U_i \text{ ext } x:A \rightarrow B \text{ by intro function } A \text{ new } x \]
\[ \gg A \text{ in } U_i \]
\[ x:A \gg U_i \text{ ext } B \]

\[ H \gg x:A \rightarrow B \text{ in } U_i \text{ by intro [new } y] \]
\[ \gg A \text{ in } U_i \]
\[ y:A \gg B[y/x] \text{ in } U_i \]

\[ H \gg U_i \text{ ext } A \rightarrow B \text{ by intro function} \]
\[ \gg U_i \text{ ext } A \]
\[ \gg U_i \text{ ext } B \]

\[ H \gg A \rightarrow B \text{ in } U_i \text{ by intro} \]
\[ \gg A \text{ in } U_i \]
\[ \gg B \text{ in } U_i \]

canonical

\[ H \gg x:A \rightarrow B \text{ ext } \lambda y.b \text{ by intro at } U_i \text{ [new } y] \]
\[ y:A \gg B[y/x] \text{ ext } b \]
\[ \gg A \text{ in } U_i \]

\[ H \gg \lambda x.b \text{ in } y:A \rightarrow B \text{ by intro at } U_i \text{ [new } z] \]
\[ z:A \gg b[z/x] \text{ in } B[z/y] \]
\[ \gg A \text{ in } U_i \]

noncanonical

\[ H,f:(x:A \rightarrow B),H' \gg T \text{ ext } t[f(a)/y] \text{ by elim } f \text{ on } a \text{ [new } y] \]
\[ \gg a \text{ in } A \]
\[ y:B[a/x], y=f(a) \text{ in } B[a/x] \gg T \text{ ext } t \]

\[ H,f:(x:A \rightarrow B),H' \gg T \text{ ext } t[f(a)/y] \text{ by elim } f \text{ [new } y] \]
\[ \gg A \text{ ext } a \]
\[ y:B \gg T \text{ ext } t \]

The first form is used when \( x \) occurs free in \( B \),
the second when it doesn’t.

\[ H \gg f(a) \text{ in } B[a/x] \text{ by intro using } x:A \rightarrow B \]
\[ \gg f \text{ in } x:A \rightarrow B \]
\[ \gg a \text{ in } A \]
equality

\[ H \gg f = g \text{ in } x:A\rightarrow B \text{ by extensionality [new } y] \]
\[ y:A \gg f(y) = g(y) \text{ in } B[y/x] \]
\[ \gg f \text{ in } x:A\rightarrow B \]
\[ \gg g \text{ in } x:A\rightarrow B \]

computation

\[ H \gg (\lambda x.b)(a) = t \text{ in } B \text{ by reduce } 1 \]
\[ \gg b[a/x] = t \text{ in } B \]
PRODUCT

formation

\[ H \gg U_i \text{ ext } x:A\#B \text{ by intro product } A \text{ new } x \]
\[ \gg A \text{ in } U_i \]
\[ x:A \gg U_i \text{ ext } B \]

\[ H \gg x:A\#B \text{ in } U_i \text{ by intro [new } y] \]
\[ \gg A \text{ in } U_i \]
\[ y:A \gg B[y/x] \text{ in } U_i \]

\[ H \gg U_i \text{ ext } A\#B \text{ by intro product} \]
\[ \gg U_i \text{ ext } A \]
\[ \gg U_i \text{ ext } B \]

\[ H \gg A\#B \text{ in } U_i \text{ by intro} \]
\[ \gg A \text{ in } U_i \]
\[ \gg B \text{ in } U_i \]

canonical

\[ H \gg x:A\#B \text{ ext } <a,b> \text{ by intro at } U_i \text{ a [new } y] \]
\[ \gg a \text{ in } A \]
\[ \gg B[a/x] \text{ ext } b \]
\[ y:A \gg B[y/x] \text{ in } U_i \]

\[ H \gg <a,b> \text{ in } x:A\#B \text{ by intro at } U_i \text{ [new } y] \]
\[ \gg a \text{ in } A \]
\[ \gg b \text{ in } B[a/x] \]
\[ y:A \gg B[y/x] \text{ in } U_i \]

\[ H \gg A\#B \text{ ext } <a,b> \text{ by intro} \]
\[ \gg A \text{ ext } a \]
\[ \gg B \text{ ext } b \]

\[ H \gg <a,b> \text{ in } A\#B \text{ by intro} \]
\[ \gg a \text{ in } A \]
\[ \gg b \text{ in } B \]

noncanonical

\[ H, z:(x:A\#B), H' \gg T \text{ ext spread}(z;u,v,t) \text{ by elim } z \text{ new } u,v \]
\[ u:A, v:B[u/x], z=<u,v> \text{ in } x:A\#B \gg T[u,v>/z \text{ ext } t] \]
H >> spread(e;x,y,t) in T[e/z]
by intro [over z:T] using w:A#B [new u,v]
>> e in w:A#B
u:A,v:B[u/w],e=<u,v> in w:A#B >> t[u,v/x,y] in T[< u,v >/z]

computation
H >> spread(<a,b>;x,y,t) = s in T by reduce 1
>> t[a,b/x,y] = s in T
EQUALITY

formation

\[ H \gg U_i \ ext \ a_1=\ldots=a_n \ in \ A \ by \ intro \ equality \ A \ n \]
\[ \gg A \ in \ U_i \]
\[ \gg A \ ext \ a_1 \]

\[ \ldots \]
\[ \gg A \ ext \ a_n \]

The default for \( n \) is 1.

\[ H \gg (a_1=\ldots=a_n \ in \ A) \ in \ U_i \ by \ intro \]
\[ \gg A \ in \ U_i \]
\[ \gg a_1 \ in \ A \]

\[ \ldots \]
\[ \gg a_n \ in \ A \]

canonical

\[ H \gg \text{axiom in } (a \ in \ A) \ by \ intro \]
\[ \gg a \ in \ A \]

\[ H, x:T, H' \gg x \ in \ T \ by \ intro \]

This doesn't work when \( T \) is a set or quotient term, since intro will invoke the equality for the set or quotient type, respectively. In any case, the equality can be used.
Appendix B

Kinding Rules for Logic in Nuprl

The basic rule is that the goal must always be a proposition.

Void Rules

There is only one void formation rule, since void is always a proposition.

Void Formation

Void proposition formation.

\( H \gg \text{void}^P \text{ in Prop}_i \text{ by intro} \)

\( H, z:\text{void}^P \gg T^P \text{ by elim z at } U_i \)

\( \gg T^P \in \text{Prop}_i \)

Int Rules

There is only one int formation rule, since int is always a data type.

\( H \gg \text{int}^D \text{ in Data}_i \text{ by intro} \)

Proof by induction. \( T \) is a proposition. \( x \) is an integer datum. The upward inductive case basically says that if you have the value \( y \), a proof that \( 0 < y \), a proof that \( T[y - 1/x] \), but not witnesses for either of these last two, you can prove that \( T[y/x] \). Likewise for the downcase.

\( H, x^D: \text{int}^D, H' \gg T^P \)

by elim \( x \) new \( z[,]y] \)

\( y^D: \text{int}^D, (y^D < 0)^P, T[y^D + 1/x]^P \gg T[y^D/x]^P \)

\( \gg T[0/x]^P \)

\( y^D: \text{int}^D, (0 < y^D)^P, z^P:T[y^D - 1/x]^P \gg T[y^D/x]^P \)
Inductive data type formation.

\[ H \gg (\text{ind}(e^D, x^D, y^D.t_d^D, t_b^D, x^D, y^D.t_u^D))^D \in \text{Data}_i \]

by intro over \( z.U_i \) new \( u, v \)

\[ \gg e^D \in \text{int}^D \]
\[ u^D : \text{int}^D, (u^D < 0)^P, v^D : \text{Data}_i \gg t_d[u^D, v^D/x, y]^D \in \text{Data}_i \gg t_b^D \in \text{Data}_i \]
\[ u^D : \text{int}^D, (0 < u^D)^P, v^D : \text{Data}_i \gg t_d[u^D, v^D/x, y]^D \in \text{Data}_i \]

Inductive proposition formation.

\[ H \gg (\text{ind}(e^D, x^D, y^P.t_d^P, t_b^P, x^D, y^P.t_u^P))^P \in \text{Prop}_i \]

by intro over \( z.U_i \) new \( u, v \)

\[ \gg e^D \in \text{int}^D \]
\[ u^D : \text{int}^D, (u^D < 0)^P, v^P : \text{Prop}_i \gg t_d[u^D, v^P/x, y]^P \in \text{Prop}_i \gg t_b^P \in \text{Prop}_i \]
\[ u^D : \text{int}^D, (0 < u^D)^P, v^P : \text{Prop}_i \gg t_d[u^D, v^P/x, y]^P \in \text{Prop}_i \]

Less Rules

All less types are propositions.

List Rules

These are analogous to the rules for int.

Union Rules

Union proposition formation.

\[ H \gg (A^P | B^P)^P \in \text{Prop}_i \]

\[ \gg A^P \in \text{Prop}_i \]
\[ \gg B^P \in \text{Prop}_i \]

Union datatype formation.

\[ H \gg (A^D | B^D)^D \in \text{Data}_i \]

\[ \gg A^D \in \text{Data}_i \]
\[ \gg B^D \in \text{Data}_i \]

Union data inhabitant formation.

\[ H \gg \text{inl}(a^D)^D \in (A^D | B^D)^D \text{ by intro} \]

\[ \gg a^D \in A^D \]
\[ \gg B^D \in \text{Data}_i \]

Union data inhabitant formation.
B. Kinding Rules for Logic in Nuprl

\[ H \gg \text{inr}(a^D)^D \in (A^D|B^D)^D \text{ by intro} \]
\[ \gg a^D \in B^D \]
\[ \gg A^D \in \text{Data}_i \]

Intro and Elim forms for union types must all take propositions only.

Function Rules

Proposition Formation

Universal propositions may be formed either by quantification over a data-type, e.g. 
\( \text{Int} \) or \( \forall A \rightarrow Prop \), or a universe of propositions, e.g. \( \forall A \in U^1 \rightarrow A \rightarrow \rightarrow \rightarrow A \). Neither type has explicit inhabitants.

Universal proposition formation. Domain must be data.

\[ H \gg (x^D : A^D \rightarrow B^P)^P \text{ in Prop}_i \text{ by intro [new y]} \]
\[ \gg A^D \in \text{Data}_i \]
\[ y^D : A^D \gg B[y^D/x]^P \text{ in Prop}_i \]

Universal proposition introduction - domain Data

\[ H \gg (x^D : A^D \rightarrow B^P)^P \text{ by intro at } U_i \text{ [new y]} \]
\[ \gg A^D \in \text{Data}_i \]
\[ y^D : A^D \gg B[y^D/x]^P \]

Universal elim - domain Data

\[ H,(x^D : A^D \rightarrow B^P)^P,H' \gg T^P \text{ by elim } f \text{ on } a \]
\[ \gg a^D \in A^D \]
\[ B^P[a^D/x] \gg T^P \]

Universal proposition formation. Domain must be propositions.

\[ H \gg (x^P : \text{Prop}_j \rightarrow B^P)^P \text{ in Prop}_i \text{ by intro [new y]} \]
\[ \gg \text{Prop}_j \in U_i \]
\[ y^P : \text{Prop}_j \gg B[y^P/x]^P \text{ in Prop}_i \]

Universal proposition introduction - domain Prop

\[ H \gg (x^P : \text{Prop}_i \rightarrow B^P)^P \text{ by intro at } U_j \text{ [new y]} \]
\[ \gg \text{Prop}_i \in U_j \]
\[ y^P : \text{Prop}_i \gg B[y^P/x]^P \]

Universal elim - codomain Prop.

\[ H,(x^P : \text{Prop}_i \rightarrow B^P)^P,H' \gg T^P \text{ by elim } f \text{ on } a \]
\[ \gg a^P \in \text{Prop}_i \]
\[ B^P[a^P/x] \gg T^P \]
$D : D \rightarrow D$

Formation of functions from data to data, e.g. $\lambda x.x + 1$.

Dependent function formation. Formation of functions from data-types to data-types. Codomain must contain data-inhabitants. function type $D:D \rightarrow D$

$H \gg (x^D : A^D \rightarrow B^D)^D$ in $Data_i$ by intro [new $y$]
   $\gg A^D$ in $Data_i$
   $y^D : A^D \gg B[y^D/x]^D$ in $Data_i$

$H \gg \lambda x^D.b^D \in (y^D : A^D \rightarrow B^D)^D$ by intro at $U_i$ [new $z$]
   $\gg A^D \in Data_i$
   $z^D : A^D \gg b[z^D/x]^D \in B[z^D/y]^D$

$H \gg (f^D(a^D))^D \in B[a^D/x]^D$ by intro using $(x^D : A^D \rightarrow B^D)^D$
   $\gg f^D \in (x^D : A^D \rightarrow B^D)^D$
   $\gg a^D \in A^D$

$D : D \rightarrow P$

Dependent function type from data-types to propositions. Inhabitants of codomain are data.

$H \gg (x^D : A^D \rightarrow Prop_i)^D$ in $Data_j$ by intro [new $y$]
   $\gg A^D$ in $Data_j$
   $y^D : A^D \gg Prop_i$ in $Prop_j$

Function from data-type to propositions. Codomain is data.

$H \gg \lambda x^D.b^P \in (y^D : A^D \rightarrow Prop_i)^D$ by intro at $U_i$ [new $z$]
   $\gg A^D \in Data_i$
   $z^D : A^D \gg b[z^D/x]^P \in Prop_i$

$H \gg (f^D(a^D))^D \in Prop_i$ by intro using $(x^D : A^D \rightarrow Prop_i)^D$
   $\gg f^D \in (x^D : A^D \rightarrow Prop_i)^D$
   $\gg a^D \in A^D$

$P : P \rightarrow P$

Dependent function type from propositions to propositions, e.g. $Not \equiv \lambda A.A \rightarrow void$.

Dependent function type from propositions to propositions.

$H \gg (x^P : Prop_i \rightarrow Prop_j)^D$ in $Data_k$ by intro [new $y$]
   $\gg Prop_i$ in $U_k$
   $y^P : U_i^P \gg Prop_j$ in $U_k$
Function from prop to prop.

\[ H \gg \lambda x^P. b^P \in (y^P : \text{Prop}_i \rightarrow \text{Prop}_j)^D \text{ by intro at } U_k \text{ [new } z] \]
\[ \gg \text{Prop}_i \in U_k \]
\[ z^P : \text{Prop}_i \gg b[z^P/x]^P \in \text{Prop}_j \]

Apply a function \( P : P \rightarrow P \) to \( P \), yielding \( P \).

\[ H \gg (f^D(a^P))^P \in \text{Prop}_j \text{ by intro using } (x^P : \text{Prop}_i \rightarrow \text{Prop}_j)^D \]
\[ \gg f^D \in (x^P : \text{Prop}_i \rightarrow \text{Prop}_j)^D \]
\[ \gg a^P \in \text{Prop}_i \]

**Propositional Implication**

Propositional implication formation.

\[ H \gg (A^P \Rightarrow B^P)^P \text{ in } U_i^P \text{ by intro} \]
\[ \gg A^P \text{ in } U_i^P \]
\[ \gg B^P \text{ in } U_i^P \]

Implication elim.

\[ H, n : (A^P \Rightarrow B^P)^P, H' \gg T^P \text{ by elim } n \]
\[ \gg A^P \]
\[ B^P \gg T^P \]

**Product Rules**

Product types can only be propositions independent product types are composed of propositions dependent product types are composed of a data-type, with a data-value, and a proposition.

**Equality Types**

The eq-type of an I-type is always a data-type. The other two can be anything, as long as they match.