Finding Computational Content in Classical Proofs

Robert Constable and Chet Murthy
Cornell University

1 Summary

We illustrate the effectiveness of proof transformations which expose the computational content of classical proofs even in cases where it is not apparent. We state without proof a theorem that these transformations apply to proofs in a fragment of type theory and discuss their implementation in Nuprl. We end with a discussion of the applications to Higman's lemma [13, 21, 10] by the second author using the implemented system.

2 Introduction: Computational content

Informal practice

Sometimes we express computational ideas *directly* as when we say $2 + 2$ reduces to 4 or when we specify an algorithm for solving a problem: "use Euclid's GCD (greatest common divisor) algorithm to reduce this fraction." At other times we refer only indirectly to a method of computation, as in the following form of Euclid's proof that there are infinitely many primes:

> For every natural number n there is a prime p greater than n. To prove this, notice first that every number m has a least prime factor; to find it, just try dividing it by $2, 3, \cdots, m$ and take the first divisor. In particular $n! + 1$ has a least prime factor. Call it p. Clearly p cannot be any number between 2 and n since none of those divide $n! + 1$ evenly. Therefore $p > n$. **QED**

This proof implicitly provides an algorithm to find a prime greater than n. Namely, divide $n! + 1$ by all numbers from 2 to $n! + 1$ and return the first divisor. The proof guarantees that this divisor will be greater than n and prime. We can "see" this algorithm in the proof, and thus we say that the proof has *computational content.*

In day-to-day mathematics people use a variety of schemes to keep track of computational content. There is no single or systematic way. People use phrases like "reduce b to b'" or "apply algorithm f." They also tend to use phrases like "this proof is constructive" or "the proof shows us how to compute." It is also sensible to say "we treat all of the concepts constructively." To understand the last phrasing notice that it is not possible to tell from the usual statement of a theorem alone whether its proofs must be constructive (or whether any proof is). To see this consider the statement:

> There are two irrational numbers, a and b such that a^b is rational.
>
> Classical Proof: 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 does not allow us to compute the first 10 digits of a since there is no way, *given in the proof,* to tell whether $a = \sqrt{2}$ or $a = \sqrt{2}^{\sqrt{2}}$. The statement of the theorem does not tell whether a constructive proof is required, in contrast to the following form, 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. We leave it as an exercise to show that both these numbers follow from the existence of unique prime factorizations.

Often when we say "there exists an object b having property P" we really mean that we can explicitly exhibit an algorithm for computing b. This use of language is so common that to convey computational ideas we can just say that "there exists" is meant constructively. Another example of how language is used to convey computational intention is this.

> Let us show: At least one of $(e + \pi)$ or $(e * \pi)$ is transcendental. Suppose not, then both are algebraic. So the coefficients of

> $(x-\pi)(x-e) = x^2-(e+\pi)*x+(e*\pi)$ are algebraic. But then according to the theorem that the field of all algebraic numbers is algebraically complete, all the roots are algebraic. But this contradicts the fact that e and π are both transcendental.

The use of "or" in this case does not tell us that we can decide which disjunct is true. But when we say every number n is even or odd, we notice that we can decide which. It is also common to say "for all x, $P(x)$ or $Q(x)$," when we mean that we can decide for any x whether it is $P(x)$ or $Q(x)$ that is true. Again, sometimes a proof will tell us that, as in the case of the proof:

> For all natural numbers n, n is prime or n is composite. We can prove this by testing whether n is prime.

This proof shows us how to decide the *or*. So another way to convey computational content is to indicate that *or* is to be interpreted in a constructive sense.

Systematic accounting

There have been various attempts to keep track of computational ideas systematically. The constructive or intuitionistic use of language is one of the oldest and most throughly studied. Another way is to explicitly use algorithms to define functions and limit assertions to equalities between them, as in Skolem's account of arithmetic [27]. There are formal languages in which programs are explicitly used to express algorithms. Another way is to introduce a double set of concepts, say function and computable function, classical and constructive logical operators, and so forth. This leads to a highly redundant language, so there have been efforts to reduce the duplication, which have resulted in certain epistemic logics [25, 17].

The approach of using an Intuitionistic or constructive core language has been extensively studied by philosophers, logicians, mathematicians and computer scientists for most of this century. Many deep and beautiful results about this method are now known, and they are being applied extensively in computer science (already since the late 60's). The applications are to the design of programming languages [24, 2], to programming methodology [19, 4], to program verification and formal methods [5, 4], and to automated reasoning [23]. This paper discusses further applications, in particular to foundational issues about the nature of computation and to problems in the semantics of functional programming languages. We believe that this "constructive language" approach to accounting for computational content has so far been the most direct approach, although for historical

reasons going back to the highly emotional debates between Brouwer and Hilbert, the approach seems a bit "controversial." Moreover, there are strong connections to issues in the philosophy of mathematics which are still unresolved and some people are loath to appear to be taking a stand on issues about which they have little concern or knowledge. It is fair to say that many scientists have used these concepts and results without regard for the philosophical issues. This paper takes no philosophical stand. We try to continue the tradition of using the techniques of constructive logic to shed light on fundamental *scientific issues.*

Constructive language

A natural starting place to understand the systematic use of constructive language is with the notion of *existence.* As far back as the ancient Greeks the notion of existence in mathematics was identified with the concept of a construction (by ruler or compass). The constructive interpretation of the quantifier $\exists x \in T.P$, meaning that there is an object of type T such that property P holds of it, is that we can construct the object in T, call it t, and know that P is true when x refers to t. I.e., P with some description of t substituted for x is true. The computational interpretation here is clear, if we think of a description of t as an algorithm for building the object. Under this interpretation, a key step in understanding constructive language is knowing how to describe the objects in a type. Before we can be specific about existential statements, we must make some commitments about what ways we allow for describing objects. Looking to simple examples like the natural numbers for guidance, we see that there are irreducible descriptions, such as 0, and compound ones, such as $0+0$. Following Martin-Löf [19], we adopt the notion that our language of descriptions will be based on the idea that some are *canonical,* such as 0, and some are not, such as $0+0$. The key principle about noncanonical descriptions is that they can be *computed* to canonical ones. This is a precise version of the notion that a description tells us how to build the object because the canonical name is a direct description of the object.

Once we have settled on a computational interpretation of existence, we then want to make sure that the other quantifiers and logical operations preserve it. This forces us to interpret "P or Q" as meaning that we know which of P or Q is true, since we can build a noncanonical description of an object a based on a case analysis. e.g., in (1.2) below, the computation

of a forces (1.1) to be computational also:

$$\sqrt{2}^{\sqrt{2}} \text{ is rational or not } \sqrt{2}^{\sqrt{2}} \text{ is rational,} \tag{1.1}$$

$$\text{if } \sqrt{2}^{\sqrt{2}} \text{ is rational take } a = \sqrt{2} \text{ else take } a = \sqrt{2}^{\sqrt{2}}. \tag{1.2}$$

Another important feature we would like in our computational understanding of language is that when we assert

for all x of type S there is a y of type T such that P

(symbolized as $\forall x \in S.\exists y \in T.P$), we mean that there is an algorithm to compute a function from S to T which produces the witness object in T given any object s of S. This suggests that we interpret the universal quantifier $\forall x \in S.P$ as asserting that there is an algorithm taking any object of type S and producing a proof of P.

The simplest statements are built out of terms (the statements themselves can be construed as terms). There is no computational content to a meaningful irreducible, such as 0 or a. But for compound terms, such as $2+2$ or $3+1$, we might say that the computational content is the set of all computations or reductions of the term. So the exact meaning depends on how the term is built (e.g., how $+$ is defined.)

The atomic statements we consider have the form $P(a_1;\cdots;a_n)$, for a_i terms, and P an operator. Associated with each atomic statement is a computation or set of computations. We take these to be the computational content. For example, the computation associated with the statement $4 = 2 + 2$ is a justification from the axioms of $+$. Compound statements are formed from $\forall, \exists, \vee, \wedge, \Rightarrow$ with their computations being naturally defined in terms of the computations of the relevant subsidiary statements. Thus, the computations of $a = b \Rightarrow b = a$ are the computations which transform computations of $a = b$ into computations of $b = a$. The computations of $\exists x \in T.P$ are those which compute a t in T and a computation of P where t has been substituted for x.

Seeing computational content

One advantage of constructive language is that it allows us to see computational content directly; we can read it off from a proof of a theorem stated constructively. In fact, constructive proofs are so transparent that we can just see the content and extract it automatically. As an example, consider the theorem

Theorem 0.1 *root:* $\forall n \in \mathsf{N}.\exists r \in \mathsf{N}.r^2 \leq n \leq (r+1)^2$

Proof: Let n be given in N, by *induction* on n show $\exists r.\ r^2 \leq m \& n < (r+1)^2$.

Base: Choose $r = 0$, use simple arithmetic.

Ind: Assume $h: \exists r.\ r^2 \leq n\ \&\ n < (r+1)^2$ using hyp h, choose r_0
$d: (r_0+1)^2 \leq (n+1) \vee (n+1) \leq (r_0+1)^2$ by arithmetic.
Show $\exists r.\ r^2 \leq (n+1)\ \&\ (n+1) < (r_0+1)^2$ by *or elimination* on d
If $(r_0+1)^2 \leq (n+1)$ then $r = (r_0+1)$, use arithmetic
If $(n+1) < (r_0+1)^2$ then $r = r_0$, use arithmetic
QED

■

The computational content of this proof in the Nuprl language [3] is given by the function:

$$\begin{array}{l}\lambda n.\ ind(n; < 0, arith >; \\ \qquad u.\lambda h.\ spread(h: r_0, isroot. \\ \qquad\qquad seq(arith: disjunct. \\ \qquad\qquad\quad decide(disjunct; \\ \qquad\qquad\qquad\qquad lesseq.(r_0+1, arith); \\ \qquad\qquad\qquad\qquad qtr.(r_0, arith))))\end{array}$$

This can be extracted automatically from the proof. It is an algorithm which computes an integer square root and a proof that the number computed is a root.

We can see computational content in classical proofs as well. Consider the following nonconstructive proof that a root exists (nonconstructive because it proceeds by contradiction).

Theorem 0.2 $\forall n \in \mathsf{N}.\exists r \in \mathsf{N}.r^2 \leq n\ \&\ n < (r+1)^2$

Proof: **("Classical"):**
Given any $n \in \mathsf{N}$,
Assume $\forall r \in \mathsf{N}.r^2 > n \vee (r+1)^2 \leq n$

Let $G = \{r : \mathsf{N} | r^2 > n\}$
$S = \{r : \mathsf{N} | (r+1)^2 \leq n\}$

Note 1. $S \cap G =$ void
2. S's elements are smaller than G's

Let $r_0 > n$ be the least element of G
$*$ $r_0^2 > n$ by definition of G

** $r_0 - 1 \in S$ since r_0 is least
$(r_0 - 1 + 1)^2 \leq n$ by def of S
$r_0^2 \leq n$

■

Intuitively, we would expect that the small modifications we made to the first proof to arrive at the second could not destroy the obvious computation inherent in the former. Moreover, we can "see" that the second proof ought to yield an algorithm, and if we are careful, we can see that the algorithm is really the same as that found in the first proof. So we would wonder if there is a way in which we can extract computational content from classical proofs such as the latter, and from more opaque classical proofs. The answer is "yes," and indeed much more can be done. First, there are various (more or less) tortuous methods [11, 15, 26], e.g. Gödel's Dialectica interpretation, which suffice. But in 1978 H. Friedman [9] drastically simplified the method. We will examine these techniques in the next section.

3 Extracting computations by translation : the A-translation

The A-translation is a short name for the method that H. Friedman [9] used to simplify and extend G. Kreisel's theorem that any classical proof of $\exists x \in \mathsf{N}.P$, where P is decidable, can be transformed into a constructive proof of the same statement. This result is itself effective, and an implementation of it would enable us to extract content from certain classical proofs, such as Theorem 0.2, by transforming them in this way and then extracting content from the resulting constructive proof. The usual presentation of this method involves two steps, first translating the classical language into a constructive one, then massaging the transation so as to bring out the content. The first step is due to Gödel.

Gödel translations

In 1932 Gödel showed how to interpret classical language into constructive. The key ideas are to notice that one way to understand the classical meaning of *or* is to say that P *or* Q means that we cannot imagine any other possibility; that is we cannot say that both $\neg(P)$ and $\neg(Q)$. So P *or* Q could be read as $\neg(\neg(P) \wedge \neg(Q))$. The key to understanding the classical use of the existential quantifier is to say that $\exists x \in T.P$ means it cannot be the case that for all x in T, $\neg(P)$ holds. If we use the symbols $\otimes$ and $\textcircled{\exists}$ for the classical disjunction and existential quantifier respectively, then the translations are:

$$
\begin{array}{rclcrcl}
(A\&B)^\circ & = & A^\circ \& B^\circ & \quad & (A \Rightarrow B)^\circ & = & A^\circ \Rightarrow B^\circ \\
\forall x.B^\circ & = & \forall x.\ B^\circ & & (A \otimes B)^\circ & = & \neg(\neg A^\circ\ \&\ \neg B^\circ) \\
(\exists\!\!\!\otimes x.B)^\circ & = & \neg\forall x.\ \neg B^\circ & & \mathit{atomic}^\circ & = & \mathit{atomic} \\
\bot^\circ & = & \bot & & & &
\end{array}
$$

Note: $\neg B = (B \Rightarrow \bot)$
where $\bot$ denotes the false proposition

Example: $(P \otimes \neg(P))^\circ = \neg(\neg(P)\ \&\ \neg\neg(P))$, for P atomic

Figure 1.1: **Gödel Transformation**

$$
\begin{array}{rcl}
P \otimes Q & \equiv & \neg((\neg(P) \wedge \neg(Q))) \\
(\exists\!\!\!\otimes x \in T.P & \equiv & \neg(\forall x \in T.\neg(P)).
\end{array}
$$

Classically these are laws relating the operations mentioned, but constructively they are definitions of new operators. Applying these definitions to a classical statement results in a constructive one that explains the classical statement in computational terms. Gödel defined a complete embedding of classical arithmetic into constructive arithmetic, showin in Figure 1.1.

Theorem 0.3 (Gödel) *If F is provable in classical arithmetic from Γ, written $\Gamma \vdash_{PA} F$, then F° is provable in constructive arithmetic from Γ°, written $\Gamma^\circ \vdash_{HA} F^\circ$.*

Corollary 0.1 *If $\vdash_{PA} \exists x \in \mathsf{N}.\phi(x)$, where $\phi(x)$ is decidable, then $\vdash_{HA} \neg\neg(\exists x \in \mathsf{N}.\phi(x))$.*

Friedman's A-translation

In 1978 Friedman [9] gave a new syntactic proof that one could automatically transform classical proofs of Σ^0_1 sentences (existential sentences where the body is decidable) into constructive proofs. This proof was in two steps, the first consisting of the just-defined double-negation translation. The second step can be defined in numerous ways, depending upon the exact kind of source and destination logics we wish to consider. We define it as Friedman did, and then a simplification, originally due to Leivant [16], which makes the task of A-translation essentially effortless.

The purpose of A-translation is to transform a Gödel-translated proof of a Σ^0_1 sentence back into a proof of the original sentence. Let $\Gamma \vdash_T G$ mean that there exists a proof of G under the assumptions Γ in theory T.

Definition 0.1 (A-Translation) *The A-translation of a proposition ϕ is accomplished by simultaneously disjoining every atomic formula of ϕ with the proposition A, and is written ϕ^A.*

For example, $(a = b \wedge b = c)^A \equiv (a = b \vee A) \wedge (b = c \vee A)$. We can prove the following theorem:

Theorem 0.4 (Friedman) *If $\Gamma \vdash_{HA} F$ then $\Gamma^A \vdash_{HA} F^A$ for any A.*

Consider now a classical proof

$$x : \mathsf{N} \vdash_{PA} \exists y \in \mathsf{N}.\Phi(x, y),$$

and let $A \equiv \exists y \in \mathsf{N}.\Phi(x, y)$. The Gödel-translation theorem lets us construct

$$x : \mathsf{N} \vdash_{HA} \neg\neg(\exists y \in \mathsf{N}.\Phi(x, y))$$

(again, $\Phi(x, y)$ is decidable) from which, by A-translation, we get:

$$x : \mathsf{N} \vdash_{HA} ((\exists y \in \mathsf{N}.\Phi(x, y)^A) \Rightarrow A) \Rightarrow A, \tag{1.3}$$

where, as we said, A is the original goal. Now, when Φ is decidable, $\Phi(x, y)^A \Leftrightarrow (\Phi(x, y) \vee A)$. It is trivial to show

$$x : \mathsf{N} \vdash_{HA} (\exists y \in \mathsf{N}.(\Phi(x, y) \vee A)) \Rightarrow A$$

when $A = \exists y \in \mathsf{N}.\Phi(x, y)$, and we can thus easily show

$$x : \mathsf{N} \vdash_{HA} (\exists y \in \mathsf{N}.\Phi(x, y)^A) \Rightarrow A, \tag{1.4}$$

(which we refer to as "Friedman's top-level trick".) Putting together Figures 1.3 and 1.4, we get

$$x : \mathsf{N} \vdash_{HA} \exists y \in \mathsf{N}.\Phi(x, y).$$

While this passage from the double-negated proof to the A-translated, double-negated proof looks quite complicated, what's really going on is that we first translate the double-negated proof from a constructive logic into a minimal logic, and then replace every instance of $\bot$ with A. Thus, if we were already in a minimal logic, we would not have to do the first step, and we could just replace every $\bot$ with A. In any case, regardless of what version of the A-translation we use, we can prove

Theorem 0.5 (Friedman) *Given a classical proof*

$$x : \mathsf{N} \vdash_{PA} \exists y \in \mathsf{N}.\Phi(x, y),$$

where Φ is decidable, we can construct a constructive proof of the same, $x : \mathsf{N} \vdash_{HA} \exists y \in \mathsf{N}.\Phi(x, y)$, via double-negation translation, followed by A-translation, followed by the top-level trick.

It follows trivially that the same holds for all Π^0_2 sentences provable in PA.

In the following development of the translation of a particular simple argument, we assume that the target logic of the double-negation translation is minimal. Hence, the A-translation simply replaces $\bot$ in the double-negated proof with A. Since the logic is minimal, this replacement operation does not perturb the proof, and so the double-negated proof is still a valid proof of the A-translated sentence.

Let's say that again. If we assume that double-negation translation leaves us in a minimal logic, then a proof of a double-negated sentence is *also* a proof of any A-translation of that sentence. This means that A-translation is really just a "name change", which does nothing of real importance; whereas double-negation translation does all the real work of extracting computations. So we focus upon the effect of double-negation translation, to see how it extracts computational content.

4 Hidden constructions

One advantage of the A-translation method is that it can help to uncover constructions that are difficult to see. We will illustrate this idea with a very simple example. A-translation has also been employed by C. Murthy [20] to obtain a computation (albeit a grossly infeasible one) from a classical proof of Higman's Lemma [10]. The following simple example is enough to illustrate several important concepts.

Let $\mathsf{N} \equiv \{0, 1, 2, \cdots\}$. Consider a function $f : \mathsf{N} \to \mathsf{N}$ whose values are all either 0 or 1. Call such a function *binary*. Here is a trivial fact about binary functions whose proof we will call **pf1**.

> For all $f : \mathsf{N} \to \mathsf{N}$ which are *binary*, there exist $i, j \in \mathsf{N}$, $i < j$ such that $f(i) = f(j)$. Here is a classical proof. Either there are infinitely many 0's in the range of f (abbreviated $Zeros(f)$) or not. If not, then there are infinitely many 1's ($Ones(f)$). In the first case, choose i, j to be two distinct numbers on which f is 0. In the other case, choose i, j to be distinct numbers on which f is 1.

This is a highly nonconstructive proof. Indeed it is quite difficult to see any construction in it at all. But there are quite trivial constructive proofs such as the following which we call **pf2**.

> Consider $f(0), f(1), f(2)$. If $f(0) = f(1)$ then $i = 0, j = 1$. If $f(0) \neq f(1)$, then if $f(0) = f(2)$ take $i = 0, j = 2$. If $f(0) \neq f(2)$, then it must be that $f(1) = f(2)$ since f is *binary*, so take $i = 1, j = 2$.

This proof is very constructive, and we see that the extracted algorithm is just

```
if f(0) = f(1)
then i := 0, j := 1
else if f(0) = f(2)
     then i := 0, j := 2
     else i := 1, j := 2
```

It is interesting that the A-translation of **pf1** does not produce the same algorithm as that obtained from **pf2**. We see the difference below.

We now examine some of the key steps in a more rigorous treatment of the first proof, **pf1**. We will try to outline just enough of the argument that the hidden construction becomes visible. First some definitions. We write $Ones(f)$ to mean that there are infinitely many 1's in the range of f, similarly for $Zeros(f)$. The classical definitions are:

$$\begin{aligned} Ones(f) &\equiv \forall i \in \mathsf{N}.\textcircled{\exists} j \in \mathsf{N}.i < j \wedge f(j) = 1 \\ Zeros(f) &\equiv \forall i \in \mathsf{N}.\textcircled{\exists} j \in \mathsf{N}.i < j \wedge f(j) = 0. \end{aligned}$$

A key lemma is that there are either infinitely many ones or infinitely many zeros. It is expressed as:

$$\forall f \in \mathsf{N} \to \mathsf{N}.binary(f) \Rightarrow Ones(f) \textcircled{\vee} Zeros(f).$$

When Gödel-translated this is

$$\forall f \in \mathsf{N} \to \mathsf{N}.binary(f) \Rightarrow \neg(\neg(Ones(f)) \wedge \neg(Zeros(f)))$$

which is

$$\forall f \in \mathsf{N} \to \mathsf{N}.binary(f) \Rightarrow ((Ones(f) \Rightarrow \bot) \wedge (Zeros(f) \Rightarrow \bot)) \Rightarrow \bot.$$

The informal classical proof employs the fact that either $Ones(f)$ or $\neg(Ones(f))$. If $\neg(Ones(f))$, then there is some point x_0 such that for $y > x_0$, $f(y) = 0$. This means that $Zeros(f)$. Let us look at the proof more

Show:$\forall f \in \mathsf{N} \to \mathsf{N}.binary(f) \Rightarrow ((Ones(f) \Rightarrow \bot) \wedge (Zeros(f) \Rightarrow \bot)) \Rightarrow \bot$
Proof. assume $f : \mathsf{N} \to \mathsf{N}$, $binary(f)$,
$Ones(f) \Rightarrow \bot$, $Zeros(f) \Rightarrow \bot$
"there is an x beyond which no values are 1's"
This corresponds to trying elimination on $Ones(f) \Rightarrow \bot$ to get $\bot$.
Show $\forall x \in \mathsf{N}.\neg\neg(\exists y \in \mathsf{N}.x < y \wedge f(y) = 1)$
Proof. arb $x : \mathsf{N}$, show $(\exists y \in \mathsf{N}.(x < y \wedge f(y) = 1) \Rightarrow \bot) \Rightarrow \bot$
a1:assume $\exists y \in \mathsf{N}.(x < y \wedge f(y) = 1) \Rightarrow \bot$
(this is equivalent to $\forall y \in (x < y \Rightarrow f(y) \neq 1)$., beyond x all values are 0)
plan to get $\bot$ by elim on $Zeros(f) \Rightarrow \bot$
show $Zeros(f)$, i.e. $\forall z \in \mathsf{N}.\neg\neg(\exists y \in \mathsf{N}.z < y \wedge f(y) = 0)$
Proof. arb $z : \mathsf{N}$ show $(\exists y \in \mathsf{N}.(z < y \wedge f(y) = 0) \Rightarrow \bot) \Rightarrow \bot$
a2: assume $\exists y \in \mathsf{N}.(z < y \wedge f(y) = 0) \to \bot$
(now know from a1, a2 that can't have f be either 0 or 1, get $\bot$)
let $w = max(z, x) + 1$
$d : f(w) = 0 \vee f(w) = 1$ by binary (f)
show $\bot$ by cases on d above
if $f(w) = 0$ then $z < w \wedge f(w) = 0$ so
$\exists y \in \mathsf{N}.(z < y \wedge f(y) = 0)$
$\bot$ by elim on a2

if $f(w) = 1$ then $x < w \wedge f(w) = 1$ so
$\exists y \in \mathsf{N}.(x < y \wedge f(y) = 1)$
$\bot$ by elim on a1

$\bot$
Qed so Zeros(f)
$\bot$ by elim on $\neg(Zeros(f))$
Qed Ones(f)
$\bot$ by elim on $\neg(Ones(f))$
QED

Figure 1.2: Constructive proof of **L0**

Show: $Zeros(f) \Rightarrow G$
Proof. a1: assume $\forall x(\exists y(x < y \ \wedge \ f(y) = 0)) \rightarrow \bot) \rightarrow \bot$
show $(\exists i, j \ F \rightarrow \bot) \rightarrow \bot$
a2: assume $(\exists i, j \ . \ F \rightarrow \bot)$, show $\bot$
could try to show $\exists i, j \ . \ F$ and use $\rightarrow$ elim, but that is too direct, we need to use $Zeros(f)$
a3: $(\exists y(0 < y \ \wedge \ f(y) = 0) \rightarrow \bot) \rightarrow \bot$ by allel a1, on 0
show $\exists y(0 < y \ \& \ f(y) = 0) \rightarrow \bot$ for $\rightarrow$ elim on a3
assume $\exists y.(0 < y \ \wedge \ f(y) = 0)$ to show $\bot$
choose y_0 where $0 < y_0 \ \wedge \ f(y_0) = 0$
a4: $(\exists y(y_0 < y \ \& \ f(y) = 0)) \rightarrow \bot) \rightarrow \bot$ by allel a1 on y_0
show $\exists y(y_0 < y \ \wedge \ f(y) = 0) \rightarrow \bot$ for $\rightarrow$ elim on a4
a5: assume $\exists y(y_0 < y \wedge \ f(y) = 0)$, *show* $\bot$
choose y_1 where $y_0 < y_1 \ \wedge \ f(y_1) = 0$.

$\exists F$ by $\exists$ intro y_0, y_1
$\bot$ by elim a2 (showing $\bot$ at a5)
$\bot$ by elim a4
$\bot$ by elim a3 (showing $\bot$ at a2)
QED

Figure 1.3: Constructive proof of **L2**

carefully. We give a constructive proof of the Gödel-translated sentence, **L0**, in Figure 1.2. The other two key steps are to show that $Ones(f)$ and $Zeros(f)$ each imply the goal.

Let $F \equiv i < j \wedge f(i) = f(j)$,and $\exists F \equiv \exists i, j \in \mathsf{N}.F$ and the double-negation translation of this is the classical goal $G \equiv \neg\neg(\exists F)$. We must prove the two lemmas:

$$\begin{array}{lll} \mathbf{L1} & : & Ones(f) \Rightarrow G \ \text{ and} \\ \mathbf{L2} & : & Zeros(f) \Rightarrow G. \end{array}$$

A proof of **L2** is given in Figure 1.3. For brevity we have omitted the types on quantifiers, and we put these proofs together in figure 1.4, to yield

$$\forall f \in \mathsf{N} \rightarrow \mathsf{N}.binary(f) \Rightarrow \neg\neg(\exists F)$$

As we said before, we assume that this proof of $\neg\neg(\exists F)$ is in a minimal logic; hence, we can automatically generate, for any A, a proof of

$$x : \mathsf{N}, binary(f) \vdash_{HA} (\exists F \Rightarrow A) \Rightarrow A,$$

Theorem 0.6 $\forall f \in \mathsf{N} \to \mathsf{N}.\mathit{binary}(f) \Rightarrow \neg\neg(\exists F)$:

Proof. assume $f : \mathsf{N} \to \mathsf{N}$, $b : \mathit{binary}(f)$, show $\neg\neg(\exists F)$
  a1: assume $\neg(\exists F)$, show $\bot$
    $a2$:$\neg(\neg(\mathit{Zeros}(f)) \wedge \neg(\mathit{Ones}(f)))$
      by lemma **L0**, with parameters f, b
    $\bot$ by *function elimination* on $a2$
      show $\neg(\mathit{Zeros}(f)) \wedge \neg(\mathit{Ones}(f))$ by *and introduction*
        show $\neg(\mathit{Zeros}(f))$
          a3: assume $\mathit{Zeros}(f)$, show $\bot$
            h:$\neg\neg(\exists F)$ by lemma **L1** with parameter $a3$
            $\bot$ by elim h on $a1$
        $\neg(\mathit{Zeros}(f))$
        show $\neg(\mathit{Ones}(f))$
          a4: assume $\mathit{Ones}(f)$, show $\bot$
            h:$\neg\neg(\exists F)$ by lemma **L2** with parameter $a4$
            $\bot$ by elim h on $a1$
        $\neg(\mathit{Ones}(f))$
      $\neg(\mathit{Zeros}(f)) \wedge \neg(\mathit{Ones}(f))$
  $\bot$
QED

Figure 1.4: Putting it together

and from this, by Friedman's top-level trick, a proof of

$$x : \mathsf{N}, binary(f) \vdash_{HA} \exists i, j \in \mathsf{N}. i < j \wedge f(i) = f(j).$$

5 Mechanizing the translations in Nuprl

The work presented in preceding sections came out of a two-year long project to understand the meaning of Friedman's translations. As part of this project, we implemented Friedman's metatheorem as a proof transformation procedure that translated proofs in a classical variant of Nuprl into proofs in constructive Nuprl. Unfortunately, we cannot implement Friedman's metatheorem for all of Nuprl; that is, the *entire* Nuprl type theory, with the addition of the axiom of excluded middle, is not a suitable candidate for Friedman's translation. In fact, we arrive at a counter-example (the translated axiom of choice) which demonstrates that we must discard, in a sense, the propositions-as-types principle in order to define a translatable logic. In the following, we spell out restrictions on Nuprl which yield a theory Nuprl°, which *can* be translated.

Having spelled out the subtheory of Nuprl which, when made classical, is translatable, we will briefly discuss the mechanized implementation of the "binary" theorem, and then discuss parts of the translated, extracted computation. Finally, we describe some results that came out of this project. They are described in detail in the second author's thesis [20].

Effective translation

The problem in translating a constructive type theory such as Nuprl is that Nuprl is not really a constructive *logic*. Consider that in a "standard" logic, a proposition is not something which we may quantify over. Rather, we must first "comprehend" it into a set or a type, and then quantify over that. In a type theory, there is no such comprehension, or rather, every proposition is automatically comprehended into a set. But in a classical logic suitable for translation, we should not be able to directly comprehend a classically proven proposition into a type, since we could then apply the axiom of choice to extract a function out of a classically proven $\forall - \exists$ statement. We will show that this pattern of reasoning is not in general translatable.

Said another way, a classical proof of a proposition resembles a guarantee that we can never construct a counter-example to the proposition. There is often no means given to construct evidence for the proposition, and so we should not be able to quantify over inhabitants of the proposition, but only reason from the fact that the proposition is inhabited. This

conforms well with the classical notion that a proposition is simply proven; the proofs of a proposition are indistinguishable, since they are noncomputational. Likewise, we should always be able to reason that if $P \Leftrightarrow Q$, then $\Phi(P) \Leftrightarrow \Phi(Q)$, where P, Q are propositions, and Φ is a predicate on propositions. This means that proposition constructors must bi-implicatively respect bi-implication.

These ideas can be made precise by defining a theory much like Nuprl, called Nuprl°, such that proofs in Classical Nuprl° can always be translated back into Nuprl°. We define a trivial mapping from Nuprl° to Nuprl (an erasing of certain annotations) that finishes the job. A Nuprl° proof is one that satisfies the following:

- Every proposition in every sequent must be statically well-formed. That is, the well-formedness of a proposition should not depend upon the truth (inhabitation) of some other proposition. Some examples of this problem come up in uses of the "set" type, which is used in Nuprl to hide computational content.

- Every proposition constructor should respect $\Leftrightarrow$, as explained before. This restriction is simple to enforce, and it comes about because we wish to enforce the "logical" nature of our language.

- the "proposition-hood" of a term must be syntactically decidable; that is, we must be able to statically decide whether a given term in a given sequent is a proposition or not, and no term which is a proposition in one sequent can be a data-value, or data-type, in another sequent (e.g. a parent sequent or subsidiary sequent). We enforce this condition by annotating every term (and every subterm) with either a P (for proposition) or a D (for data), and verifying that certain compatibility conditions hold between adjacent sequents in proof trees.

With these three conditions, we can show that, for a suitable subset of the type constructors of the Nuprl type theory (e.g., Π-types, Σ-types, universes, higher-order predicates and data, integers, lists, etc,) the Classical Nuprl° theory, with excluded middle, is translatable automatically into constructive Nuprl°. We can then embed Nuprl° back into Nuprl by simply erasing the annotations. Hence, we have a set of sufficient conditions for the success of the translation in Nuprl.

One wonders if these restrictions are really relevant; if these restrictions actually exclude pathological cases of proofs which are intrinsically not translatable, and include important cases of proofs which we must translate. The second part of this question is easily answered: our translation effort,

which succeeded in translating Higman's Lemma, attests to the tractability of doing mathematics within this fragment of Nuprl. To answer the first part of the question, we showed that, for a particular formalization of the axiom of choice, which is trivially provable in Nuprl, its double-negation translation is not provable in Nuprl. It can be written thus:

$$\forall x \in Dom.\exists y \in Rng.\Phi(x,y) \Rightarrow \exists f \in Dom \rightarrow Rng.\forall x \in Dom.\Phi(x,f(x)),$$

and its proof in Nuprl is $\lambda\, h.\langle \lambda\, x.h(x).1, \lambda\, x.h(x).2\rangle$. We can think of this as a function which, given an input function, stands as a proof of $\forall x \in Dom.\exists y \in Rng.\Phi(x,y)$, *splits* the input function into the two parts, the first of which computes values in *Rng*, and the second part of which witnesses the correctness of the first part.

But this proof assigns a name, h, to the proof of

$$\forall x \in Dom.\exists y \in Rng.\Phi(x,y),$$

which we expressly forbade in our conditions above. For this $\forall\exists$ sentence is an object of proof, hence a proposition, and to assign a name to the proofs (inhabitants) of a proposition is to quantify over it, which, again, is forbidden in our fragment of Nuprl. The double-negation translation of this axiom is equivalent to the following sentence (where Φ is double-negated):

$$\forall x \in Dom.\neg\neg(\exists y \in Rng.\Phi(x,y)) \Rightarrow \neg\neg(\forall x \in Dom.\exists y \in Rng.\Phi(x,y)),$$

and we showed that this sentence is not true in the standard model of Nuprl [1]. Thus, we cannot use the axiom of choice to construct functions by induction in our classical proofs. Instead, we must formalize functions as binary relations for which certain existence and uniqueness predicates hold. Using such a formulation, we can then do almost all of the reasoning we wish to about functions, paying only a small price in terms of cumbersomeness of the logic.

We have not yet discussed the A-translation, nor have we discussed the actual implementation of our proof translations. However, the actual implementation details follow rather straightforwardly from the restrictions on the forms of proofs, and the A-translation follows likewise. The difficult part is restricting the Nuprl logic to make translation possible at all. In the end, though, we prove

Theorem 0.7 (Conservative Extension for Nuprl$^\circ$) *Given a proof in Classical Nuprl$^\circ$ of a Π^0_2 statement, we can construct a proof in (constructive) Nuprl$^\circ$ of the same statement.*

Proof: In [20].

```
\f.int_eq(f(1);0;
        int_eq(f(2);0;
                <1,<2,axiom>>;
                int_eq(f(3);0;
                        <1,<3,axiom>>;
                        <2,<3,axiom>>));
          int_eq(f(2);0;
                int_eq(f(3);0;
                        <2,<3,axiom>>;
                        int_eq(f(4);0;
                                <2,<4,axiom>>;
                                <3,<4,axiom>>));
                <1,<2,axiom>>))
```

Figure 1.5: The extracted binary computation

A formalization of the binary theorem

We formalized the binary theorem in Nuprl, giving it a classical proof essentially identical to the proof described earlier in this paper. The sentence actually proven was:

$$\forall f \in \mathsf{N} \to \{0,1\}.\exists i,j \in \mathsf{N}.i < j \wedge f(i) = f(j).$$

As described before, we proved $Ones(f) \otimes Zeros(f)$, and from this we concluded the actual goal. We then employed an automatic double-negation/A-translation tactic, which converted the proof to a constructive one, from which we extracted and executed the computational content. This program yielded the same numeric answers as the program derived earlier. We list in Figure 1.5 the computation that was extracted from this scheme, after normalization and trivial compression, and in Figure 1.6 a pseudo-code version of this program. Note that, as we said before, this strategy requires the first *four* terms of f.

We formalized a version of Nash-Williams' [10] "minimal bad sequence" proof of Higman's Lemma in Nuprl, and translated it using the same apparatus that we developed for the binary theorem, extracting constructive content in the same way. This program was approximately 12 megabytes in size, which made it infeasible to run the program on any nontrivial inputs. Later, we formalized another classical argument, in the same vein as Higman's Lemma, and from which we extracted a program of 19 megabytes in size. We were dismayed by the sizes of the extracted programs, but, as we shall discuss in the next section, this is purely a matter of the exact translation implementation (which was quite inefficient), and not an intrinsic problem with double-negation/A-translation.

```
if f(1)=0 then
  if f(2)=0 then
    <1,2>
  else if f(3)=0 then
    <1,3>
  else
    <2,3>
else if f(2)=0 then
  if f(3)=0 then
    <2,3>
  else if f(4)=0 then
    <2,4>
  else
    <3,4>
else
  <1,2>
```

Figure 1.6: The pseudo-code binary computation

6 The algorithmic content of classical proofs

We have outlined herein a general method of extracting algorithms from classical proofs of Π_2^0 sentences. However, as should be clear, the form of the algorithm extracted from a classical proof via translation is oftentimes difficult to read, understand, and relate to the original classical proof. Thus one wonders if one could extract a correct algorithm directly from the classical proof. In our work on this problem, we build on the work of Griffin [12], who discovered and verified a consistent typing for the operator "control" [7] (written $\mathcal{C}$) in the simply-typed lambda-calculus. Based on his work, we have shown that there is an intimate relation between the proof translations (Kuroda [6, 18], Kolmogorov [14], etc) and continuation-passing-style translation, and that one can in fact extract sensible, meaningful algorithms directly from classical proofs. We can summarize our findings as follows:

Theorem 0.8 *Friedman's method (double-negation/A-translation) is exactly a continuation-passing-style [8, 22] compilation of the extracted "classical witness" from a proof in a classical theory (say, Peano Arithmetic).*

We have learned that many of the different double-negation translations can be understood in terms of their effect upon programs, and not just in terms of their effect upon proofs. To wit, we have shown that several different double-negation translations in fact fix the order of evaluation of expressions in a functional language, thus providing an explanation of

various eager and lazy computation schemes in terms of each other, and hence in a single functional language.

Theorem 0.9

- *A particular modified Kolmogorov double-negation translation fixes a call-by-name evaluation order on functional program expressions.*
- *A modified Kuroda translation fixes an eager (call-by-value) evaluation order on functional programs.*
- *a further modification of the Kolmogorov translation makes pairing an eager operation, and allows either by-value or by-name lambda-application.*

These discoveries show that classical proofs have algorithmic content, and that this algorithmic content is made explicit by the double-negation/A-translations. They show that the proof-translation method of double-negation/A-translation is isomorphic to the program translation method of continuation-passing-style (CPS) translation, and that classical proofs can be interpreted as nonfunctional programs (enriched with a the nonlocal control operator "control") [7]. They provide a rational, *algorithmic* foundation for Friedman's translation method, based on its effect on computations, and upon programs. That is, in a manner which is not characterized by effects upon proofs and sequents, but rather on a "semantic" basis (characterized by effects on programs and program fragments).

With this knowledge, we can evaluate the classical proof of the binary theorem directly, by assigning the operator $\mathcal{C}$ to be the algorithmic content of the rule of double-negation elimination. Moreover, we can utilize results of Plotkin [22] to define more efficient double-negation translations than the one which we originally chose, with the knowledge that these more efficient translations are every bit as powerful as the original ones, and, in addition, produce extensionally equivalent programs.

Acknowledgements

We wish to acknowledge Gabriel Stolzenberg, for his inspiration in opening this area of inquiry, his painstaking proofreading of this work, and his sage advice. Thanks are also due to Tim Griffin, who discussed these issues at length with us. We also wish to acknowledge the referees. This research could not have been conducted without the generous support of the National Science Foundation and the Office of Naval Research, under the auspices of Ralph Wachter, under grants CCR-8616552 and N00014-88-K-0409.

Bibliography

[1] Allen, S.F. (1987). A non-type theoretic definition of Martin-Löf's types. in *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE.

[2] Burstall, R. and Lampson, B. (1984). A kernel language for abstract data types and modules. in *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, eds. G. Kahn, D. MacQueen, and G. Plotkin, volume 173 of *Lecture Notes in Computer Science*, Berlin. Springer-Verlag.

[3] Constable, R. (1985). The semantics of evidence. Technical Report TR 85–684, Cornell University, Department of Computer Science, Ithaca, New York.

[4] Constable, et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey.

[5] Coquand, T. and Huet, G. (1985). Constructions: A higher order proof system for mechanizing mathematics. in *EUROCAL '85: European Conference on Computer Algebra*, ed. B. Buchberger, pages 151–184. Springer-Verlag.

[6] Dalen, D. and Troelstra, A. (1989). *Constructivism in Mathematics.* North-Holland.

[7] Felleisen, M., Friedman, D., Kohlbecker, E., and Duba, E. (1986). Reasoning with continuations. in *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 131–141.

[8] Fischer, M. (1972). Lambda-calculus schemata. in *Proceedings of the ACM Conference on Proving Assertions about Programs*, volume 7 of *Sigplan Notices*, pages 104–109.

[9] Friedman, H.. (1978). Classically and intuitionistically provably recursive functions. in *Higher Set Theory*, ed. Scott, D. S. and Muller, G. H., volume 699 of *Lecture Notes in Mathematics*, pages 21–28. Springer-Verlag.

[10] Gallier, J. (1987). What's so special about Kruskal's Theorem and the ordinal Γ_0. Technical Report MS-CIS-87-27, University of Pennsylvania, Philadelphia, PA.

[11] Girard, J-Y. (1987). *Proof Theory and Logical Complexity, vol. 1.* Bibliopolis, Napoli.

[12] Griffin, T. (1990). A formulae-as-types notion of control. in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages.*

[13] Higman, G. (1952). Ordering by divisibility in abstract algebras. in *Proc. London Math. Soc.*, volume 2, pages 236–366.

[14] Kolmogorov, A. (1967). On the principle of the excluded middle. in *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, ed. J. van Heijenoort, pages 414–437. Harvard University Press, Cambridge, Massachusetts.

[15] Kreisel, G. (1958). Mathematical significance of consistency proofs. *Journal Of Symbolic Logic*, 23:155–182.

[16] Leivant, D. (1985). Syntactic translations and provably recursive functions. *Journal Of Symbolic Logic*, 50(3):682–688.

[17] Lifschitz, V. (1982). Constructive assertions in an extension of classical mathematics. *Journal Of Symbolic Logic*, 47:359–387.

[18] Luckhardt, H. (1973). *Extensional Godel functional interpretation; a consistency proof of classical analysis*, volume 306 of *Lecture Notes in Mathematics*, pages 41–49. Springer-Verlag.

[19] Martin-Löf, P. (1982). Constructive mathematics and computer programming. in *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam. North Holland.

[20] Murthy, C. (1963). *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Department of Computer Science.

[21] Nash-Williams, C. (1963). On well-quasi-ordering finite trees. in *Proc. Cambridge Phil. Soc.*, volume 59, pages 833–835.

[22] Plotkin, G. (1975). Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159.

[23] Pollack, R. (1989). The theory of LEGO. Unpublished draft.

[24] Reynolds, J. (1974). Towards a theory of type structure. in *Proc. Colloque sur la Programmation, Lecture Notes in Computer Science 19*, pages 408–23. NY:Springer-Verlag.

[25] Shapiro, S. (1985). *Epistemic and Intuitionistic Arithmetic*, pages 11–46. North-Holland.

[26] Troelstra, A., editor. (1973) *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics.* Springer-Verlag.

[27] van Heijenoort, J., editor. (1967) *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Massachusetts.