**Classical Proofs as Programs:
How, What and Why**

Chetan R. Murthy

TR 91-1215
July 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported in part by an NSF graduate fellowship and NSF grant CCR-8616552 and
ONR grant N00014-88-K-0409.

Classical Proofs as Programs: How, What and Why

Chetan R. Murthy*

Department of Computer Science

Cornell University

Ithaca, NY 14853 USA

murthy@cs.cornell.edu

July 1, 1991

Abstract

We recapitulate Friedman’s conservative extension result of (suitable) classical over constructive systems for Π_2^0 sentences, viewing it in two lights: as a translation of programs from an almost-functional language (with $\mathcal{C}$) back to its functional core, and as a translation of a constructive logic for a functional language to a classical logic for an almost-functional language. We investigate the computational properties of the translation and of classical proofs and characterize the classical proofs which give constructions in concrete, computational terms, rather than logical terms. We characterize different versions of Friedman’s translation as translating slightly different almost-functional languages to a functional language, thus giving a general method for arriving at a sound reduction semantics for an almost-functional language with a mixture of eager and lazy constructors and destructors, as well as integers, pairs, unions, etc. Finally, we describe how to use classical reasoning in a disciplined manner in giving classical (yet constructivizable) proofs of sentences of greater complexity than Π_2^0 . This direction offers the possibility of applying classical reasoning to more general programming problems.

1 Introduction

It is well-known that constructive type theories and logics can serve as reasoning systems for functional programming languages. In [Mur91], we demonstrated that classical arithmetic can serve in a like fashion as a total-correctness reasoning system for programs written in an almost-functional programming language based on the integers, and incorporating the $\mathcal{C}$ [FFED86] operator (pronounced “control”, a relative of call/cc [CR86]). This result follows from Friedman’s conservative extension of (suitable) classical over constructive logics for Π_2^0 sentences, which shows that a classical proof of a Π_2^0 sentence can be translated to a constructive proof of the same sentence, and

*Supported in part by an NSF graduate fellowship and NSF grant CCR-8616552 and ONR grant N00014-88-K-0409

Griffin’s [Gri90] work showing a correspondence between the classical axiom and $\mathcal{C}$. In [Mur91] we used these results to show the soundness of a particular almost-call-by-name¹ evaluation semantics for proofs in classical arithmetic (regarded as programs), by the method of Friedman specialized to a modified Kolmogorov translation, hence showing that classical proofs of Π_2^0 sentences compute *evidence* [Con85] for those propositions in a direct and explicit manner. That is, we showed that classical proofs of Π_2^0 sentences could be construed as computing evidence in the same sense as constructive proofs, and, moreover, the programming language associated with such proofs is a functional language augmented with the nonlocal control operator $\mathcal{C}$. Here we seek to extend this work, and address three problems:

- Friedman’s work is restricted to Π_2^0 sentences. Why? And can we understand this computationally? Why does this restriction come about? We will find that the restriction to Π_2^0 is natural and can be accounted for in a manner which is (we feel) satisfying to the computer scientist, as well as to the logician. Moreover, we will give simple examples which demonstrate that, in general, classical proofs of Π_3^0 and Σ_2^0 sentences do *not* compute evidence for the propositions they stand as classical witnesses for.
- There are other translations besides the Kolmogorov translation for which Friedman’s work applies, and which we can use in proving the same conservative extension results. We define a modified Kuroda translation, give the translations on programs which accompanies it, and assert the soundness of an almost-call-by-value evaluation semantics for proofs in classical arithmetic. We go on to show a “mixed-Kolmogorov” translation which allows us to prove the soundness of an evaluation semantics for a programming language with both by-name and by-value applications. This result extends to eager/lazy pairing and injections also.
- We use this same methodology and demonstrate how to use classical reasoning in a disciplined manner to give classical, but automatically constructivizable, proofs of sentences which are more complex than Π_2^0 . We give a simple extension to the proof system of classical arithmetic which allows us to accomplish this goal.

Our work is based on the principle that we can understand a nonfunctional language by understanding its translation into a functional language; equally, we can give a logic for a nonfunctional language by “pulling back” a logic for a functional language thru the appropriate translation.

¹The “almost” prefix is due to the nature of the Kolmogorov translation, which prohibits CPS-translation from translating integer-typed expressions; hence, these expressions cannot be guaranteed to evaluate in a by-name manner. However, the type system guarantees that these expressions will evaluate to integer constants, and will never abort.

2 Constructive Type Theories

Constructive type theories are based on the Curry-Howard isomorphism, under which propositions are identified with types, and their proofs with members of types. This isomorphism relies heavily on the notion of *evidence*. Evidence is informally defined as values in a programming language, and is assigned to propositions as follows:

- Evidence for $a = b$, where a, b contain no free variables, would be completely axiomatic, consisting in the computation of a, b down to numerals (they could be, for instance, $2*5 = 5+5$, which would need to be computed down to $10 = 10$).
- Evidence for $A \wedge B$ would be evidence for A and for B . This could be given as a pair, $\langle u, v \rangle$, where u is evidence for A , and v is evidence for B .
- Evidence for $A \Rightarrow B$ would be a function which, when given evidence for A , would compute evidence for B .
- Evidence for $A \vee B$ would be evidence for A , or for B , and a tag telling us which disjunct we were getting evidence for. This could be represented as $inl(u)$ (inject-left), where u is evidence for A , or $inr(v)$, where v is evidence for B .
- Evidence for $\exists x \in \mathbb{N}. R(x)$ would be an integer, n , and evidence for $R(n)$. This, again, could be represented as a pair.
- Evidence for $\forall x \in \mathbb{N}. R(x)$ would be a function which, given $n \in \mathbb{N}$, would compute evidence for $R(n)$.

This definition can be equally well viewed as assigning types to program values, and it can be extended to expressions in a confluent programming language in a straightforward way. An example of a constructive type theory is Heyting Arithmetic (HA) [DT89], which is essentially Peano Arithmetic (PA), or number theory, without the axiom of excluded middle. In such a type theory, a constructive proof of $\forall x. \exists y. R(x, y)$ is evidence for this proposition - i.e., a function which, will compute evidence for $\exists y. R(X, y)$. This in turn is a pair $\langle Y, Z \rangle$, such that Z is evidence for $R(X, Y)$.

We use a particular format for expressing the typing judgments of our logic (alternatively, for expressing the rules of our logic). We will write $\Gamma \vdash_T M : \phi$, where Γ is a (possibly empty) list of typing assumptions $x : A$, ϕ is a proposition, and M is a program phrase whose free variables are among those declared in Γ . The reading of this sequent is that, under the typing assumptions Γ , program phrase M has type ϕ . E.g.,

$$\begin{array}{l}
\Gamma \vdash M(N):B \\
\text{BY modus ponens} \\
\vdash M:A \Rightarrow B \\
\vdash N:A
\end{array}$$

is a sequent-calculus version of the typing rule for function application, as well as being the constructive rule of modus ponens.

3 “Denotational” Translations of Nonconstructive Logics

It is by now common to use denotational semantics to interpret nonfunctional languages into pure functional ones. These target languages enjoy beautiful properties, among them the availability of workable correctness logics - the constructive type theories. Given that every reasonable sequential programming language can be thus translated, one wonders what sort of (type) theory would result if one were to “pull back” a type system thru the semantic interpretation function. For example, if τ is the translation, and if $\tau(M)$ has type ϕ , is there some meaningful type we can assign to $\tau^{-1}(\phi)$? It is also usually true that $M \succ M'$ (where $\rightarrow$ is reduction) implies $\tau(M) = \tau(M')$, in some appropriate equational theory of functional languages. Does this also mean that $M : \tau^{-1}(\phi)$ implies $M' : \tau^{-1}(\phi)$? This would be a type-preservation theorem about the relation of our typing system to evaluation. Finally, for values, say, observable datatypes, does $M : \phi$ imply $\tau(M) : \phi$? The latter would tell us that the translation on values of datatypes was the identity.

Here we will show that the call-by-name CPS-translation, $\bullet$, a semantic translation of $\lambda + \mathcal{C}$ (a functional language augmented with the “control” operator) into λ , induces a classical type sytem on $\lambda + \mathcal{C}$. It will turn out that the classical type system can be viewed as arising first, as a type sytem for a non-Church-Rosser language, and that we can think of the CPS-translation as fixing which reductions are valid by stipulating that $M \succ M'$ only if $\underline{M} = \underline{M}'$.

4 CPS-Translation Basics

The continuation-passing-style (CPS) translation is a mapping from lambda-terms to lambda-terms which mimicks the operational semantics of evaluation of the original term by explicit reductions of the translated term. We will not go into a lengthy discussion of the various translations here; let it suffice to say that there are at least two well-known translations in the literature, the call-by-value

translation and the call-by-name translation, each of which mimicks the respective operational semantics. For further information and tutorial, the reader is encouraged to consult [Plo75,Fis72,Gri90]. One central fact about CPS-translation is that, for call-by-value (resp. by-name) evaluation, a lambda-calculus program M converges to a value b iff its call-by-value (resp. by-name) CPS-translation, applied to the proper top-level continuation (usually $\lambda x.x$) converges to this same value.

5 How: Friedman's Translation

Friedman showed that one could translate a classical arithmetic (PA) proof of a Π_2^0 sentence into a constructive arithmetic (HA) proof of the same by a simple extension of the method of double-negation translation. We give the proof for the case where the double-negation translation used is the Kolmogorov translation [Kol67].

Definition 1 (Kolmogorov Translation) Given a sentence ϕ in PA, define $\overline{\phi}$, the Kolmogorov double-negation translation of ϕ , as being the simultaneous double-negation of every propositional position in ϕ :

$$\begin{array}{ll} \overline{(A \vee B)} & \mapsto \neg\neg(\overline{A} \vee \overline{B}) \\ \overline{(A \wedge B)} & \mapsto \neg\neg(\overline{A} \wedge \overline{B}) \\ \overline{(\exists x \in A. B)} & \mapsto \neg\neg(\exists x \in A. \overline{B}) \\ \overline{(\forall x \in A. B)} & \mapsto \neg\neg(\forall x \in A. \overline{B}) \\ \overline{(A \Rightarrow B)} & \mapsto \neg\neg(\overline{A} \Rightarrow \overline{B}) \\ \overline{P} & \mapsto \neg\neg(P) \quad (P \text{ prime}) \end{array}$$

Theorem 1. (Double-Negation Embedding) If $\vdash_{PA} \phi$, then $\vdash_{HA} \overline{\phi}$.

This theorem tells us that the Kolmogorov translation converts a classically provable sentence into a constructively provable sentence, but it does *not* tell us what form the constructive proof will take. Friedman's discovery was that (if one gave the constructive proof properly) one could replace instances of falsehood ($\perp$) with an arbitrary proposition, in particular ϕ . Friedman showed:

Theorem 2 (Friedman's A-Translation) If $\vdash_{PA} \phi$, where ϕ is Σ_1^0 , and is $\perp$ -free, then $\vdash_{HA} \overline{\phi}[\phi/\perp]$.

In fact, for correctly-constructed translations, the A-translation does not disturb the constructive extracts. I.e., if $\vdash_{HA} M : \overline{\phi}$, then $\vdash_{HA} M : \overline{\phi}[\phi/\perp]$. Friedman then showed:

Theorem 3 (Conservative Extension) If we have a proof $\vdash_{PA} \phi$, where ϕ is Σ_1^0 , then we can construct a proof $\vdash_{HA} \phi$.

Proof: Suppose $\phi \equiv \exists y \in \mathbb{N}. f(y) = 0$, and let $A \equiv \phi$. Let $\neg_{\phi}(T) \equiv T \Rightarrow \phi$. Then from a classical proof of ϕ , we obtain a constructive proof of

$$\neg_{\phi} \neg_{\phi} (\exists y \in \mathbb{N}. \neg_{\phi} \neg_{\phi} (f(y) = 0)).$$

To recover a proof of ϕ , it suffices to prove

$$\neg_{\phi} (\exists y \in \mathbb{N}. \neg_{\phi} \neg_{\phi} (f(y) = 0)),$$

which we refer to as “Friedman’s top-level trick.” This is trivial, but a deep analysis of the ramifications of the choice of the “top-level trick” would shed light on the essentially non-lazy features of the interactive top-loop in lazy languages, and we omit it for reasons of space. ■ The result for Π_2^0 sentences follows by considering free integer variables.

6 What: Classical Proofs as Programs

So given a classical proof π_K of a Π_2^0 sentence, we can obtain a proof π_J (corresponding to a functional program) by translation which stands as a constructive witness for that sentence, and computes evidence for it. But we can do more. By “guessing” algorithmic content for the classical rules and verifying them via translation, we can extract evidence from PA proofs directly. As an example, consider the following pair of (original, double-negation-translated) rules:

$$\begin{array}{ll} \Gamma \vdash \text{<what?>} : P & \bar{\Gamma} \vdash \lambda k. \underline{M}(\lambda m. m(\lambda g. g(\lambda v. h.v(k))) \lambda x. x) : \bar{P} \\ \text{BY } \neg\neg\text{-elim} & \text{BY translated } \neg\neg\text{-elim} \\ \vdash M : \neg\neg(P) & \vdash M : \neg\neg(\bar{P}) \quad (= \neg\neg(\neg\neg(\bar{P} \Rightarrow \neg\neg(\perp)) \Rightarrow \neg\neg(\perp))) \end{array}$$

If one looks carefully, one sees that the translated extract is the *call-by-name CPS-translation of the term CM*! This operator, pronounced “control,” and described in [FFED86], is a nonlocal control operator, and its evaluation semantics are (informally) expressed as

$$\begin{array}{ll} E[CM] & \rightarrow_1 M(\lambda x. \mathcal{A}(E[x])) \\ E[AM] & \rightarrow_1 M. \end{array}$$

where $\mathcal{A}$ is the “abort” operator, syntactic sugar for $\mathcal{C}(\lambda (). M)$, and intuitively, $\mathcal{C}$ applies its argument to an abstraction of the evaluation context ($E[]$) in which it is evaluated. Thus, $\mathcal{C}$ allows a term to “goto” another evaluation context, discarding the context in which it is currently evaluating. We can repeat this process for every rule of our classical logic. For each rule, we examine

the translated rule fragment, and attempt to discover the proper “classical extract” which, when translated, will give the translated extract. This defines a programming language, $Prog_K$, and a translation $\bullet$ back to its functional core, $Prog_J$. Here are a few of the clauses of the translation:

$$\begin{aligned} \underline{x} &\equiv x \text{ (} x \text{ a variable)} \\ \underline{MN} &\equiv \lambda k. \underline{M}(\lambda m. m \underline{N} k) \\ \underline{\lambda x. M} &\equiv \lambda k. k(\lambda x. \underline{M}) \\ \underline{CM} &\equiv \lambda k. \underline{M}(\lambda m. m(\lambda g. g(\lambda v. h.v(k)))\tau_\phi). \end{aligned}$$

But, having arrived at a candidate set of rules (complete with extractions), we still must show that these rules are sound. This comes in three stages:

- Show that reduction to a *weak-head-normal-form* [Bar84] terminates. Apply “colon-translation”, discovered by Plotkin [Plo75], and extended by Griffin [Gri90].
- Show that reduction is sound. We consider each reduction rule in turn, and show that $(M \rightarrow_1 M') \Rightarrow (\underline{M} = \underline{M}')$. The evaluation rule for $\mathcal{C}$ is difficult, but Felleisen’s methods [FFED86] suffice.
- Show that reduction preserves typing, i.e., $(\Gamma \vdash M : T) \wedge (M \rightarrow_1 N) \Rightarrow (\Gamma \vdash N : T)$.

The last obligation - to show that reduction preserves typing - is in some sense the real “meat” behind any typing system. While it is easy to show that all the original reduction rules of the functional core language preserve typing, we cannot show that if $E[CM] : \phi$, where M has type $\neg\neg(T)$, then $M(\lambda x. \mathcal{A}E[x])$ has the same type ϕ . Let us demonstrate this:

$$\begin{aligned} M &: \neg\neg(T) \\ x &: T \\ E[x] &: \phi \\ \mathcal{A}E[x] \equiv \mathcal{C}\lambda ().M &: \text{untypable} \\ \lambda x. \mathcal{A}E[x] &: \text{untypable} \end{aligned}$$

Thus, we see that $M(\lambda x. \mathcal{A}E[x])$ is ill-typed. We can remedy this problem in one of two ways. Griffin [Gri90] chose to assume that the type of the entire program was, instead of some arbitrary ϕ , the type $\perp$. Then, the typing above goes thru trivially. His method corresponded to wrapping a complete program M in a term $\mathcal{C}\lambda k. k(M)$.

Besides being on a somewhat logically insecure footing, this choice does not give meanings directly to the programs we wish to write – which do not have these “wrappers” around them. Moreover, it obscures the fact that in reality programs do not have type $\perp$, but rather concrete

types. We choose instead to replace every $\perp$ in the entire proof (and in every rule) with ϕ (a pre-A-translation), without changing the type of the program. This can happen when ϕ is $\perp$ -free (e.g. when ϕ is Σ_1^0). The reader may verify that this typing works. So what have we wrought with our modification? Suppose we start with a classical proof of a $\perp$ -free Σ_1^0 sentence, ϕ . If we replace every instance of $\perp$ in every sequent and every rule (including the classical axiom) with ϕ , then the “classical extract” is unchanged, just as the constructive extract is unchanged by such a modification. Moreover, in this modified logic, called $PA(\phi)$, *reduction preserves typing*.

Theorem 4 (Reduction Preserves Typing) Given a proof $\vdash_{PA} M : \phi$, we can show $\vdash_{PA(\phi)} M : \phi$, and if $M \rightarrow M'$, then

$$\vdash_{PA(\phi)} M' : \phi.$$

As we noted before, we can prove that reduction to a weak-head-normal-form [Bar84] will terminate (via a colon-translation argument). Consider now the case $\phi \equiv \exists x \in \mathbb{N}. f(x) = 0$. We know that the only *whnfs* of this type are pairs $\langle N, axiom \rangle$, where $N \in \mathbb{N}$ and *axiom* is a proof of $f(N) = 0$ (that is, $f(N) = 0$ is true). But such a value is *also* of type ϕ in PA , or even in HA . Thus, even though we had to move to $PA(\phi)$ to prove that reduction preserves typing, when we finish reduction, we are left with a well-typed value in PA .

Theorem 5 (Classical Evaluation Semantics) Given a proof $\vdash_{PA} M : \phi$, if ϕ is Σ_1^0 , then $M \rightarrow V$, where V is a *value*, and $\vdash_{PA} V : \phi$.

So we have found an evaluation semantics for (nearly) classical proofs which is sound, and gives evidence for Σ_1^0 sentences. But what import does this have for real classical proofs?

6.1 Why Π_2^0 ?

So finally we come to the question: *Why* does Friedman’s translation only work for Π_2^0 sentences? Let us look at the Σ_1^0 sentences. A Σ_1^0 type can always be written as $\exists x \in \mathbb{N}. f(x) = 0$. Members of this type are pairs $\langle X, Y \rangle$ where $X \in \mathbb{N}$, and Y is a witness of $f(X) = 0$ - that is, *axiom*. Σ_1^0 sentences have the special property that their *whnfs* are concrete data values, and specifically contain no embedded function closures, which also means they contain no unevaluated continuations. Quite simply, when evaluating a program of Σ_1^0 type ϕ , we always end up with a value devoid of function closures. And this value has type ϕ in PA exactly because it has that type in $PA(\phi)$.

Now a Π_2^0 type can be written $\forall x \in \mathbb{N}. \exists y \in \mathbb{N}. f(x, y) = 0$. An expression of this type is a function F with domain $\mathbb{N}$, such that $F(X)$ computes evidence of type $\exists y \in \mathbb{N}. f(X, y) = 0$. W.l.o.g

we can assume that F is of the form $\lambda x.M$. When F is applied to $X \in \mathbf{N}$, $F(X)$ has type $\exists y \in \mathbf{N}.f(X, y) = 0$ ($\in \Sigma_1^0$), and so it will compute evidence for this type. But the intuitive reason why F has type $\forall x \in \mathbf{N}.\exists y \in \mathbf{N}.f(x, y) = 0$ is that when fed a concrete datum, it produces a concrete datum, again, without any embedded function closures or instances of $\mathcal{C}$.

Consider now a Σ_2^0 type, written $\phi \equiv \exists x \in \mathbf{N}.\forall y \in \mathbf{N}.f(x, y) = 0$. A proof of such a type is a program which computes to a pair $\langle X, G \rangle$, where $X \in \mathbf{N}$, and, in $PA(\phi)$, G has type $\forall y \in \mathbf{N}.f(X, y) = 0$. G is a function and when G is applied to a value Y , $G(Y)$ can reduce to evidence for $f(X, Y) = 0$. Alternatively, $G(Y)$ can “unwind” evaluation to a previous context, in which case $G(Y)$ is *not* evidence for $f(X, Y) = 0$. So if we had an oracle which could tell us whether, for any Y , $G(Y)$ would apply an unevaluated continuation, thus unwinding its evaluation context, we could decide whether G was evidence for $\forall y \in \mathbf{N}.f(X, y) = 0$. This is not decidable in general, so we cannot determine if G is evidence for $\forall y \in \mathbf{N}.f(X, y) = 0$.

For example, there is a simple classical proof of

$$\psi \equiv \exists n \in \mathbf{N}.\forall m \in \mathbf{N}.f(n) \leq f(m).$$

This sentence expresses the fact that every boolean function attains a minimum (where $0 < 1$). Suppose we had a constructive proof of this fact, from which we extracted a program. Intuitively, this cannot be the case because the constant function $f : n \mapsto 1$ is indistinguishable within n steps from a function which becomes zero after $n + 1$ (or some suitably large number) of values. So if our program could examine the constant function $f : n \mapsto 1$ and determine that it attained a minimum of 1 after N steps of computation, then we could “spoof” it by giving it as input a function which attained zero only after a suitably long time (say a stack of N 2’s). And our program would report an incorrect minimum on this input. We give a classical proof of ψ below:

```

⊢ ψ BY double-negation elim
¬(ψ) ⊢ ⊥ BY function-elim
⊢ ψ BY intro 0
  ⊢ ∀m ∈ N. f(0) ≤ f(m) BY function intro
    m : N ⊢ f(0) ≤ f(m) BY cases on f(0) ≤ f(m)
      f(0) ≤ f(m) ⊢ f(0) ≤ f(m) BY hypothesis
      f(m) < f(0) ⊢ f(0) ≤ f(m)
        BY double-negation elim
          THEN function-elim
            f(m) < f(0) ⊢ ψ BY intro m, etc

```

The computation in this proof is

$C\lambda k.k\langle 0,$
 $\quad \lambda m.\text{if } f(0) \leq f(m) \text{ then axiom}$
 $\quad \text{else } k\langle m, \lambda m'.\text{axiom} \rangle \rangle$

Intuitively, the program given by the proof will make a “guess” that 0 is the desired n . Then, given m , it will check if $f(0) \leq f(m)$. If so, then it will simply report success. If not, then $f(m) < f(0)$, which means that $f(m) = 0$. So the program will unwind the context back to before it chose 0, and instead choose m . As a result, our program does not really provide evidence for the truth of the proposition it purports to be a proof of, but rather, provides a program which, given a counterexample, will “throw” back to a place in the computation where it can change the “answer” to disqualify the counterexample.

There is one special case. If G contains no expressions $\mathcal{AM}$, then it is easy to show that G is evidence for $\forall y \in \mathbb{N}.f(X, y) = 0$, because every unevaluated continuation is an expression $\mathcal{AM}$. Since G has type $\forall y \in \mathbb{N}.f(X, y) = 0$ in $PA(\phi)$, and $\mathcal{AM}$ does not occur in G , then G has the same type in PA . Thus G has the same type in $PA(\forall y \in \mathbb{N}.f(X, y) = 0)$. And from this, we can show that G computes evidence for $\forall y \in \mathbb{N}.f(X, y) = 0$.

As one can see, the criterion for deciding that a proof of a Σ_2^0 (or stronger) sentence actually computes evidence for that sentence is nontrivial, and for any realistic implementation, intractable.

7 Other Translations as Variant Evaluation Semantics

There are other double-negation translations which, in concert with A-translation, suffice to translate classical proofs of Π_2^0 sentences into constructive proofs. One such translation is a simple modification of the Kuroda negative translation. This translation is very simple - to translate a term, double-negate the *body* of every universal quantification and r.h.s. of every implication, and the outside of the entire term ². We can again “discover” a $Prog_K$, and a corresponding translation $\underline{\bullet}$. This time, it turns out that $Prog_K$ has call-by-value semantics. Here is a fragment of the translation:

$$\begin{aligned}
\underline{x} &\equiv \lambda k.k(x) \text{ (} x \text{ a variable)} \\
\underline{MN} &\equiv \lambda k.\underline{M}(\lambda m.\underline{N}(\lambda n.m(n)k)) \\
\underline{\lambda x.M} &\equiv \lambda k.k(\lambda x.\underline{M}) \\
\underline{CM} &\equiv \lambda k.\underline{M}(\lambda m.m(\lambda v.h.k(v))(\lambda x.x))
\end{aligned}$$

²The original Kuroda translation did not double-negate the r.h.s. of implications

While the left-hand-sides look just like the call-by-name language already proposed, it is *not*, since the lambda-abstraction and application here, as well as the $\mathcal{C}$, are call-by-value. In fact, if one consults a standard work on CPS-translation [Plo75,FFED86], one finds that this translation is indeed a call-by-value translation. By a process analogous to that for the Kolmogorov translation, and using techniques essentially identical to those found in [Plo75,FFED86], we can show that a by-value evaluation semantics is sound for this programming language and type system, again for Π_2^0 sentences only.

We can go further, though, and consider “mixed-mode” translations. Consider the following Kolmogorov translation of the rule of modus ponens:

$$\begin{array}{l} \overline{\Gamma} \vdash \lambda k. \underline{M}(\lambda m. \underline{N}(\lambda n. m(\lambda k. k(n))k)) : \overline{B} \\ \text{BY translated modus ponens} \\ \vdash \underline{M} : \overline{A} \Rightarrow \overline{B} \quad (= \neg\neg(\overline{A} \Rightarrow \overline{B})) \\ \vdash \underline{N} : \overline{A}. \end{array}$$

We can add a second rule of modus ponens to our classical logic, “by-value modus ponens”, which does the obvious thing - the application term extracted will function in a by-value manner. In a like manner, we can add eager pairing, or even half-eager pairing (which would evaluate, say, the first component of a pair when constructing the pair). This translation has the undesirable quality that eager features are not reflected in the translated types. It is a simple (though tedious) matter to give another translation which does reflect the eagerness of expressions in the forms of types.

7.1 An *A Priori* Classical Programming Language

The fact that we can invent new translations which enforce different evaluation orders is not surprising. In fact, we could imagine that, rather than starting with a classical logic, and discovering a programming language by observing that double-negation translation looked like CPS-translation, we had instead just discovered, from first principles, a programming language in classical proofs. We could have taken the lambda-calculus, and added the operator $\mathcal{C}$, to arrive at the programming language $\lambda + \mathcal{C}$. One usually writes the evaluation semantics for such a language as:

$$\begin{array}{ll} E[(\lambda x. b)N] & \triangleright_1 E[b[N/x]] \\ E[\mathcal{C}M] & \triangleright_1 M(\lambda x. \mathcal{A}E[x]) \\ E[\mathcal{A}M] & \triangleright_1 M. \end{array}$$

But this definition has the undesirable quality that we have not specified what, if anything, is meant by “ $E[\]$ ”. In deterministic evaluation, $E[\]$ specifies an *evaluation context* [FF86], which,

loosely speaking, identifies unambiguously in a program the next redex for contraction. To remedy this fault, Felleisen [Fel87] introduced the notion of local and global reduction rules. Local reduction rules were those which were valid in any program context, and global rules were those which were only valid on entire programs. Speaking in these terms, one can rephrase the evaluation rules for a programming language in two classes, using $\rightarrow_1$ for local rules, and $\triangleright_1$ for global rules:

$$\begin{array}{lll}
(\lambda x.b)N & \rightarrow_1 & b[N/x] \quad (\beta) \\
(\mathcal{C}M)N & \rightarrow_1 & \mathcal{C}\lambda k.M(\lambda f.k(fN)) \quad (\mathcal{C}_L) \\
M(\mathcal{C}N) & \rightarrow_1 & \mathcal{C}\lambda k.N(\lambda a.k(Ma)) \quad (\mathcal{C}_R) \\
(\mathcal{A}M)N & \rightarrow_1 & M \quad (\mathcal{A}_L) \\
M(\mathcal{A}N) & \rightarrow_1 & N \quad (\mathcal{A}_R) \\
\mathcal{C}M & \triangleright_1 & M(\lambda x.\mathcal{A}x) \quad (\mathcal{C}_{\lambda_1}) \\
\mathcal{A}M & \triangleright_1 & M \quad (\mathcal{A}_{\lambda_1}).
\end{array}$$

Again, the $\triangleright_1$ rules are only applicable to entire programs. We can show that all these rules can be given proper typings in the same manner as was done for the other, global rules. That is, we can show that each of the local rules can be given a typing in classical logic, but that the global rules require us to “pre-A-translate” the logic. However, as has been observed numerous times, the reduction system specified above is *not* Church-Rosser. To make it Church-Rosser, one must further restrict the applicability of some of the rules. For instance, when the redex is $(\mathcal{C}M)(\mathcal{C}N)$, we are faced with a choice of $\mathcal{C}_L$ or $\mathcal{C}_R$. Likewise, given a redex $(\lambda x.b)(\mathcal{C}N)$, we must choose whether to apply β or $\mathcal{C}_R$.

The choice of a translation, in a very real sense, is equivalent to deciding on the precedence of the various rules when they conflict. So, for instance, choosing the call-by-name CPS-translation, and the accompanying Kolmogorov-translation, is equivalent to choosing to disallow $\mathcal{C}_R$ completely. And choosing the call-by-value CPS-translation outlined previously is equivalent to choosing to allow $\mathcal{C}_L$ everywhere, call-by-value β (β_v), and $\mathcal{C}_R$ only when M is a value, i.e. a λ -abstraction.

To demonstrate this, the reader may amuse himself by CPS-translating, with both the call-by-name and call-by-value translations, each of β , β_v , $\mathcal{C}_L$, $\mathcal{C}_R$, and the restricted version of $\mathcal{C}_R$ (where M is a λ -abstraction), and verifying the facts stated above.

8 Extending up the Arithmetic Hierarchy

In a certain sense, the results we have presented are all that can be hoped for. It is a simple matter to give classically provable Σ_2^0 sentences $(\exists n \in \mathbb{N}. \forall m \in \mathbb{N}. f(n, m) = 0)$ from which we have no *hope* of extracting constructions. The reason, as we have discussed, lies in our use of classical reasoning to prove the proposition $\forall m \in \mathbb{N}. f(n, m) = 0$. If we could somehow prevent this, then the value G from our previous example would be purely functional, and we would have a proof of the Σ_2^0 sentence. To achieve this, we must selectively disallow classical reasoning upon particular propositions. We do this with a new term-forming construct, the $J(\bullet)$ ³ operator, and new rules which govern when we may introduce or eliminate the J operator. Syntactically, the J operator can be applied to any proposition, and results in a proposition. It introduces no new free variables, and binds no variables. The Kolmogorov-translation of $J(T)$ is $\neg\neg(T)$. Note that the type T is *not* internally translated.

There is only one rule for the J operator:

$$\begin{array}{c} \bar{n} : \mathbb{N}, \bar{x} : J(\Gamma) \vdash_K (\lambda^v \bar{x}. M)(\bar{x}) : J(T) \\ \text{BY J-protect} \\ \bar{n} : \mathbb{N}, \bar{x} : \Gamma \vdash_J M : T. \end{array}$$

The meaning of $\vdash_J$ (resp. $\vdash_K$) is that the proof from this point down (leaf-ward) in the proof tree must be given constructively (resp. classically). Intuitively, $J(\bullet)$ is a marker which delineates where classical reasoning may not be used. The remainder of the rules of PA treat J-protected terms as atomic propositions. The translation of the J-protect rule is basically the by-value translation of $(\lambda^v \bar{x}. M)(\bar{x})$, but we protect from translation the term M . For the case where $\bar{x} : J(\Gamma) \equiv x : J(\psi_1), y : J(\psi_2)$, the translated sequent would be:

$$\begin{array}{c} \bar{n} : \mathbb{N}, x : \neg\neg(\psi_1), y : \neg\neg(\psi_2) \vdash_J \lambda k. x(\lambda x. y(\lambda y. k(M))) : \neg\neg(T) \\ \text{BY translated J-protect} \\ \bar{n} : \mathbb{N}, x : \psi_1, y : \psi_2 \vdash_J M : T. \end{array}$$

and it is a relatively simple matter to show:

Theorem 6 (PA+J Conservative Extension)

If $\vdash_{PA+J} \forall x \in \mathbb{N}. J(\psi_1(x)) \Rightarrow \exists y \in \mathbb{N}. J(\psi_2(x, y))$, then $\vdash_{HA} \forall x \in \mathbb{N}. \psi_1(x) \Rightarrow \exists y \in \mathbb{N}. \psi_2(x, y)$

³The name is chosen to remind the reader of “intuitionistic”

Proof: Along the same lines as Friedman’s conservative extension proof. The translation of the J-protect rule has already been shown. ■

Moreover, the constructive extract we obtain from this proof is as outlined above, and we can give sound evidence semantics for the original classical proof as before. However, it is easy to show that we cannot, in this classical logic, prove

$$\exists n \in \mathbb{N}. J(\forall m \in \mathbb{N}. f(n) \leq f(m)).$$

One simply notes that the proof given, and in fact any proof, of this fact, re-uses the fact that $\neg(\exists n \in \mathbb{N}. J(\forall m \in \mathbb{N}. f(n) \leq f(m)))$ while proving $\forall m \in \mathbb{N}. f(n) \leq f(m)$. This re-use is prohibited, because before proving this universal, one must invoke J-protect, which guarantees that the negative hypothesis is not available.

9 Conclusions and Related Work

This work is based on the principle that we can understand the algorithmic content of a nonconstructive reasoning system by translating proofs into a constructive system, and then using the constructive content there to infer what the direct algorithmic content should be for the nonconstructive system. Our work is based on the fundamental logical results of Friedman [Fri78], who showed that classical arithmetic (and other classical theories) conservatively extends its constructive counterpart, and the work of Griffin [Gri90], who discovered that one could give the control operator $\mathcal{C}$ as the algorithmic extract of the classical axiom. Griffin did not employ A-translation, though, and did not explore the implications of A-translation results for total-correctness of programs written using $\mathcal{C}$. Our results elsewhere [Mur91] showed that such programs correspond to classical proofs, and that their total-correctness corresponds to the constructivizability of these proofs. We have seen that, based upon the observation that Friedman’s translation is a CPS-translation, we can “pull back” a reasoning system for an almost-functional language from a reasoning system for its functional core. We used this idea to show why certain classical proofs were always constructivizable, and why other classes could not in general be so.

We showed how different translations allowed us to pull back radically different reasoning systems for our programming language, corresponding to different operational semantics. We have seen that in fact we can start with a non-Church-Rosser “classical” programming language, with a classical type system, and, by specifying the translation into a constructive logic/functional programming language, we restrict reduction in the classical language to be confluent. Finally, we

showed how to extract evidence systematically from proofs of non- Π_2^0 sentences.

In the future, we wish to explore more fully the “design space” of translations. There are many, many double-negation translations which suffice for Friedman’s results, and it would be quite interesting to see what sorts of logics arise out of “pulling back” reasoning systems with these. We also would like to begin using classical theories with the J operator as a way of giving natural correctness proofs for nontrivial programs using $\mathcal{C}$. The availability of J allows us to reason about nontrivial postconditions without “coding them up” as integer equalities, e.g. “ σ is a most general unifier of $\langle X, Y \rangle$ ” is awkward to express as a predicate $f(\sigma, X, Y) = 0$. Finally, there is the task of integrating these results with the hierarchies of control operators proposed by, among others, Danvy and Filinski [DF90].

References

- [Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.
- [Con85] R. Constable. The semantics of evidence. Technical Report TR 85-684, Cornell University, Department of Computer Science, Ithaca, New York, May 1985.
- [CR86] W. Clinger and J. Rees. The revised³ report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.
- [DF90] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.
- [DT89] D. A. Van Dalen and A. S. Troelstra. *Constructivism in Mathematics*. North-Holland, 1989.
- [Fel87] Matthias Felleisen. *The Calculi of λ_v – CS conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.
- [FF86] M. Felleisen and D. Friedman. Control operators, the SECD machine and the λ -calculus. In *Formal Description of Programming Concepts III*, pages 131–141. North-Holland, 1986.
- [FFED86] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 131–141, 1986.
- [Fis72] M. J. Fischer. Lambda-calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, volume 7 of *Sigplan Notices*, pages 104–109, 1972.
- [Fri78] Harvey Friedman. Classically and intuitionistically provably recursive functions. In Scott, D. S. and Muller, G. H., editor, *Higher Set Theory*, volume 699 of *Lecture Notes in Mathematics*, pages 21–28. Springer-Verlag, 1978.
- [Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.
- [Kol67] A. N. Kolmogorov. On the principle of the excluded middle. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 414–437. Harvard University Press, Cambridge, Massachusetts, 1967.
- [Mur91] Chet Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 1991.

- [Plo75] G. Plotkin. Call-by-name, call-by-value, and the λ -calculus. *Theoretical Computer Science*, pages 125–159, 1975.