An Evaluation Semantics for Classical Proofs

Chetan R. Murthy

TR 91-1213
June 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.

An Evaluation Semantics for Classical Proofs

Chetan R. Murthy*
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA
murthy@cs.cornell.edu

Abstract

We show how to interpret classical proofs as programs in a way that agrees with the well-known treatment of constructive proofs as programs and moreover extends it to give a computational meaning to proofs claiming the existence of a value satisfying a recursive predicate. Our method turns out to be equivalent to H. Friedman's proof by "A-translation" of the conservative extension of classical over constructive arithmetic for Π_2^0 sentences. We show that Friedman's result is a proof-theoretic version of a semantics-preserving CPS-translation from a nonfunctional programming language (with the "control" ($\mathcal{C}$, a relative of call/cc) operator) back to a functional programming language. We present a sound evaluation semantics for proofs in classical number theory (PA) of such sentences, as a modification of the standard semantics for proofs in constructive number theory (HA). Our results soundly extend the proofs-as-programs paradigm to classical logics and to programs with $\mathcal{C}$.

1 Introduction

It is well-known that there is an intimate relation between functional programming languages and constructive logical systems. To wit, a natural theory of typing for a functional programming language is usually constructive; equally, the natural computation system associated with a constructive mathematical system is functional. In this paper we extend this correspondence, showing that the same relationship holds between a functional programming language augmented with imperative control operators ($\lambda + \mathcal{C}$) and classical number theory. We demonstrate that one can augment the natural programming language associated with constructive (Heyting) arithmetic (HA), with the nonlocal control operator $\mathcal{C}$ [1] [1], and arrive at the natural programming language for classical (Peano) arithmetic (PA). The method of proof involves appeal to a classical result of Friedman [2] and Kreisel [3], showing the conservativity of PA over HA for Σ_1^0 sentences (and, by consideration of free integer variables, Π_2^0) [2]. We demonstrate that Friedman's A-translation can be viewed as a continuation-passing-style translation.

It is by now folklore [4,5,6] that one can view the *values* of a simple functional language as specifying *evidence* for propositions in a constructive logic, in the following manner:

- 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, $2 \times 5 = 5 + 5$, 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 $\exists x \in \mathsf{N}.R(x)$ would be an integer, n, and evidence for $R(n)$. This, again, could be represented as a pair.
- Evidence for $\forall x \in \mathsf{N}.R(x)$ would be a function which, given $n \in \mathsf{N}$, would compute evidence for $R(n)$.

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

[1] There are now two different operators with the appellation "control," both introduced by Felleisen. We will use the version introduced in [1], whose evaluation semantics can be summarized (informally) as $E[\mathcal{C}M] \rightarrow_1 M(\lambda\, x.\mathcal{A}(E[x]))$, where $\mathcal{A}M \equiv \mathcal{C}\lambda\, ().M$.

[2] $\forall x.\ \exists y.\ R(x,y)$, where R is a decidable proposition

Equally, one can view this definition as specifying a method of assigning types to program *values*. One can then extend this definition to encompass program *expressions* which are not values, such as $(\lambda\ x.b)(N)$, by defining a type system which is sound with respect to the evaluation system. That is, if M is given type T, then M evaluates to a value b, and b has type T (is evidence for T). We could call such program expressions *indirect evidence*. In Nuprl[7], as well as other constructive systems, these program expressions are purely functional.

Griffin [8] extended this work to classical propositional logic by showing that one could assign the operator "control" ($\mathcal{C}$) as the algorithmic extract of the classical rule of double-negation elimination. He gave a weak type preservation theorem which allowed one to prove that, except for nonlocal exits (due to invoking continuations), classical logic could be viewed as a typed programming language. But this interpretation excluded a semantics of evidence for classical proofs. Here we find that in fact his program extraction method works, with some simply modifications, for total-correctness logics, and that one can give a *total-correctness* typing to the nonlocal control operator $\mathcal{C}$, again, in such a way that when expression M is assigned type T (a Σ^0_1 type), M evaluates to a value b, again with type T.

Griffin's result hinged on the fact that continuation-passing-style (CPS) translation [9] on programs in the simply-typed lambda-calculus induces a double-negation translation [10] on their types. He used this to prove that, except for type errors caused by "escaping" from deep within an expression to the top-level, reduction preserves typing. But this ignores the fact that it is exactly at the top-level of a program that it is most important that reduction preserves typing, since otherwise we cannot hope to arrive at a semantics of evidence:

> The types of "classical programs" cannot be given the same operational interpretation as the types of "constructive programs." ... In the type system presented here, the distinction between a "returning expression" and a "jumping expression cannot be made by inspecting an expression's type. Thus, if M is a classical program of type $\alpha \rightarrow \beta$ and N is a classical program of type α, we know only that if the application of M to N returns to the current control context, then it will return with a (classical) value of type β.

Moreover, the "classical" conjuction is defined as $\alpha \wedge \beta \equiv \neg(\alpha \rightarrow \neg(\beta))$, and so even if a program of type $\alpha \wedge \beta$ does return to its invoking context (if it does not "goto"), the value computed bears no clear relation to evidence for A and for B, in the sense that we understand it constructively. Thus for complete programs, there is no criterion for determining whether (and how) a program actually produces evidence in the constructive sense.

Our work (specifically Theorem 5) addresses this problem, and augments double-negation translation with A-translation [2] to give the desired evidence semantics for Σ^0_1 and Π^0_2 sentences. Thus, in the same sense as HA proofs stand as evidence for the propositions the prove, so do PA proofs of Σ^0_1 and Π^0_2 sentences. Moreover, when considering a sentence ϕ of greater complexity, we can determine what sorts of types a program M will exit nonlocally with.

In addition, our work addresses the problem of total-correctness reasoning for call-by-name or partially lazy languages with explicit control; in fact, the programming language we associate with classical proofs is a call-by-name one. The amazing thing about this result is that (as will be made clear) much of the machinery we use was discovered by Friedman [2] and Kreisel [3] in the context of classical logical systems.

2 Background

Consider a functional programming language with integers (and iteration combinator), binary pairing, and binary disjoint unions. Call this language $Prog_J$. We give the syntax of this language in Figure 1. [3] It is derived from the programming language of Nuprl [7], and hence some of the notation will be unfamiliar to the reader. The notation $spread(Exp_1; Var_1, Var_2.Exp_2)$ is prefix notation for

```
let ⟨Var₁, Var₂⟩ = Exp₁ in
  Exp₂
end
```

and the notation $decide(Exp_0; Var_1.Exp_1; Var_2.Exp_2)$ similarly expands to

```
case Exp₀ of
  inl(Var₁) => Exp₁
| inr(Var₂) => Exp₂
end.
```

The form $ind(Exp_1; Exp_2; Var_1, Var_2.Exp_3)$ is an iteration form, where Exp_1 is the iteration counter,

[3] k_n is syntax for the integer n

$$
\begin{array}{rcl}
Exp & \equiv & Var_1 \\
& | & \lambda\ Var_1.Exp_1 \\
& | & Exp_1(Exp_2) \\
& | & k_n\ (n \in \mathsf{N};\ \textit{integer constants}) \\
& | & S(Exp_1) \\
& | & Exp_1 + Exp_2 \\
& | & Exp_1 \times Exp_2 \\
& | & ind(Exp_1; Exp_2; Var_1, Var_2.Exp_3) \\
& | & \langle Exp_1,\ Exp_2 \rangle \\
& | & spread(Exp_1; Var_1, Var_2.Exp_2) \\
& | & inl(Exp_1) \\
& | & inr(Exp_1) \\
& | & decide(Exp_0; Var_1.Exp_1; Var_2.Exp_2) \\
& | & axiom
\end{array}
$$

$$
\begin{array}{rcl}
ind(0; b; x, y.u) & \rightarrow_1 & b \\
ind(k_n; b; x, y.u) & \rightarrow_1 & \\
\multicolumn{3}{r}{u[k_n, ind(k_{n-1}; b; x, y.u)/x, y]} \\
& & (n > 0) \\
decide(inl(t); x.l; y.r) & \rightarrow_1 & l[t/x] \\
decide(inr(t); x.l; y.r) & \rightarrow_1 & r[t/y] \\
spread(\langle t_1,\ t_2 \rangle; u, v.t) & \rightarrow_1 & t[t_1, t_2/u, v] \\
(\lambda\ x.b)(t) & \rightarrow_1 & b[t/x] \\
S(k_n) & \rightarrow_1 & k_{n+1} \\
k_n + k_m & \rightarrow_1 & k_{n+m} \\
k_n * k_m & \rightarrow_1 & k_{n*m}
\end{array}
$$

Figure 1: The Syntax and Reduction Rules of $Prog_J$

Exp_2 is the base case, and Exp_3 computes the inductive case. At various times it will be necessary to distinguish those expressions which are integers. We do this by annotating them with $\bullet^D$, e.g., $ind(Exp_1{}^D; Exp_2; Var_1{}^D, Var_2.Exp_3)$. As an aside, one of the important properties of both HA and PA is that integer expressions have only integer subexpressions, and the reduction rules preserve this.

It is well-known that we can give a specification/correctness logic for this language by thinking of the programs as proofs of sentences in HA. In fact, we can view a sentence of constructive number theory as a specification of a program, and a proof of that sentence as a program which meets said specification. Alternatively, we can think of the specification as the type of a program in a particularly rich theory of program typing. Here, for example, we reproduce some of the rules of HA, and the corresponding program fragments. One reads each sequent of the form $\Gamma \vdash M\!:\!\Phi$ as saying that under the typing assumptions Γ, fragment M has type Φ; equivalently, under the assumptions that Γ are true, Φ is true, and has constructive witness M. The refinement calculus tells us how to inductively construct a witnessing term for a proposition as we prove it; alternatively, it allows us to construct a proof of well-typed-ness for a program.

Modus Ponens

$H \vdash M(N)\!:\!T$
  BY modus ponens A
  $\vdash N\!:\!A$
  $\vdash M\!:\!A \Rightarrow T$

Symmetry of =

$n : \mathsf{N},\ m : \mathsf{N},\ u : n = m \vdash axiom\!:\!m = n$

Transitivity of =

$a, b, c : \mathsf{N},\ u : a = b,\ v : b = c \vdash axiom\!:\!a = c$

Monotonicity of $S(\bullet)$

$n : \mathsf{N},\ u : S(n) = 0 \vdash axiom\!:\!\bot$

The meaning of the above typing rule is that *axiom* has type $\bot$ exactly when u has type $S(x) = 0$.

Injectivity of $S(\bullet)$

$x, y : \mathsf{N},\ u : x^D = y^D \vdash axiom\!:\!S(x)^D = S(y)^D$

Surjectivity of $S(\bullet)$

$x, y : \mathsf{N},\ u : S(x)^D = S(y)^D \vdash axiom\!:\!x^D = y^D$

Induction

$n^D\!:\!\mathsf{N} \vdash ind(n^D; B; i, n^D.F(n - 1^D)(i))\!:\!T$
  BY induction
  $\vdash B\!:\!T[0^D/n^D]$
  $\vdash F\!:\!\forall m^D \in \mathsf{N}.T[m^D/n^D] \Rightarrow T[S(m)^D/n^D]$

On this interpretation, we view a proof M in HA of $\forall x.\ \exists y.\ R(x, y)$ as being a program which, given X, will compute a suitable Y such that $R(X, Y)$ holds. We can think of the sentence $\forall x.\ \exists y.\ R(x, y)$ as being a proposition of which M is a proof, of being a specification which M meets, or a type, which M is a member of, and we will use these terms interchangeably. We can view program evaluation as just a proof-reduction operation. For instance, for HA and $Prog_J$, every reduction rule on well-typed $Prog_J$ programs is mirrored in proof-reduction steps on HA proofs (the so-called subject reduction property). Also, it is known [11] that *any* reduction strategy will compute well-formed, well-typed terms to the same normal forms. Hence for the natural programming language associated with HA, one reduction strategy is as good as another. When we add the $\mathcal{C}$ operator, this ceases to be true, and we are forced to either pick a particular deterministic evaluator, or restrict the reduction rules in such a manner that we can retain soundness. We will choose the former, and define

Definition 1 (Canonical Forms) Integers, *axiom*, and closed λ-terms, injections, and pairs are all canonical terms. All other closed expressions are noncanonical.

Definition 2 (Principal Arguments)
For each left-hand-side of a reduction rule, we can define the principal argument of the term. E.g., the first subterm of each of the *decide*, *spread*, *ind*, S, and application terms. For the $+$ and $\times$ terms, the principal arguments are both subterms.

Definition 3 (Evaluation to Canonical Form)
The process of evaluation to canonical form is as follows: If a term is a canonical form, do nothing. Otherwise, evaluate its principal arguments, in left-to-right order, to canonical form, and then apply one of the reduction rules. If none of the reduction rules apply, terminate the entire process with an error. Otherwise, repeat.

Fact 1 (Termination of Evaluation)
Evaluating closed, well-typed programs to canonical form terminates, successfully, in a canonical form.

Finally, we define complete evaluation:

Definition 4 (Complete Evaluation)
Complete evaluation of a closed expression consists in evaluating to canonical form, and then completely evaluating every binding-free immediate subterm (e.g. M, N in $\langle M,\ N\rangle$, but *not* b in $\lambda\ x.b$). We denote that a term M *completely evaluates* to a term N by writing $M \succ N$, and that we are working in the purely functional fragment defined so far by $M \succ_J N$.

Felleisen [12] pointed out that such a deterministic evaluator can be understood as an algorithm for dividing up a program P into an evaluation context $E[\,]$ and a redex M such that $E[M] = P$, and $E[\,]$ binds no variables in M. We assume that such has been done for our evaluator, and simply note that this is made precise in [13]. Notice that our evaluator is lazy, but that programs are evaluated completely.

While (for various technical reasons) HA is unsuitable as a programming logic, we feel there are many proper extensions of HA, Martin-Lōf type theories [14] among them, which are much more amenable to use as programming logics. The paradigm of proofs-as-programs applies to these logics also, and allows us to view a proof of a sentence Φ in Nuprl [7], for instance, as a program which meets specification Φ.

While this explanation is satisfying for functional programs and constructive reasoning systems, one wonders if one can find similar reasoning systems for almost-functional programming languages with nonlocal control operators such as call-with-current-continuation (call/cc) [15] and $\mathcal{C}$ [1]. Dually, one wonders what sort of programming language (if any) arises from viewing classical number theory, Peano Arithmetic (PA), as a specification logic in the same manner as HA. The remainder of this paper will answer these questions.

3 *PA* as a Specification Logic

To use PA as a specification logic in the same manner as HA, we need a way to extract, from a classical proof of a sentence Φ, *evidence* for Φ. Clearly, this is not always possible, since $(M_i\ \mathit{halts}) \vee \neg(M_i\ \mathit{halts})$ is one proposition for which there could be no such evidence (in general). Kreisel and Friedman identified one class of sentences whose classical proofs contained such evidence - the Π^0_2 sentences. Such a sentence is of the form $\forall x.\ \exists y.\ R(x,y)$, where $R(x,y)$ is a decidable proposition. Moreover, R can always be written as $f_R(x,y) = 0$, where f_R is a primitive recursive function. Friedman's method of proof was a purely syntactic argument, an extension of the Kolmogorov translation [16], called the A-translation. We reproduce here a version due to Leivant [17].

Definition 5 (The Kolmogorov Translation)
Given a sentence ϕ in number theory, define ϕ°, the Kolmogorov double-negation translation of ϕ, as being the simultaneous double-negation of every propositional position in ϕ:

$$\begin{array}{rcll}
(A \vee B)^\circ & \longmapsto & \neg\neg(A^\circ \vee B^\circ) & \\
(A \wedge B)^\circ & \longmapsto & \neg\neg(A^\circ \wedge B^\circ) & \\
(\exists x \in A.B)^\circ & \longmapsto & \neg\neg(\exists x \in A.B^\circ) & \\
(\forall x \in A.B)^\circ & \longmapsto & \neg\neg(\forall x \in A.B^\circ) & \\
(A \Rightarrow B)^\circ & \longmapsto & \neg\neg(A^\circ \Rightarrow B^\circ) & \\
P^\circ & \longmapsto & \neg\neg(P) & (\text{P } \mathit{prime})
\end{array}$$

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

Proof: By structural induction on the classical proof. We will do the three rules which concern $\perp$ - the classical axiom, $\perp$-elimination, and monotonicity of $S(\bullet)$, since the other rules translate relatively trivially.

Classical Axiom: The classical axiom $(P \Rightarrow \perp) \Rightarrow \perp \vdash_{PA} P$ is translated as

$$\neg\neg(\neg\neg(P^\circ \Rightarrow \neg\neg(\perp)) \Rightarrow \neg\neg(\perp)) \vdash_{HA} P^\circ.$$

We observe that $\neg\neg(\bot)$ is equivalent to $\bot$, and this allows to reduce the hypothesis to $\neg\neg(\neg\neg(\neg\neg(P^\circ)))$, which is equivalent to P°, since P° is outermost negated. We can also observe that we never use the rule of $\bot$-elimination in this derivation.

Monotocity of $S(\bullet)$: The classical rule is $S(x) = 0 \vdash_{PA} \bot$. The translation is

$$\neg\neg(S(x) = 0) \vdash_{HA} \neg\neg(\bot)$$

and is easily provable. We can note that we do not use the rule of $\bot$-elimination in this derivation.

$\bot$-elimination: The classical rule is $\bot \vdash_{PA} T$, and the translation is $\neg\neg(\bot) \vdash_{HA} T^\circ$. By observing that, again $\neg\neg(\bot)$ is equivalent to $\bot$, and T° is outermost negated, we see that this sequent is again easily provable. Again, we can observe that we never use the rule of $\bot$-elimination in the translated derivation.

■

As we have noted, the translated proof does not use the rule of $\bot$-elimination. One way of looking at A-translation is that it simply notices this fact, and replaces $\bot$ systematically throughout the proof with some new proposition, A. When we do this, we must justify that each rule of HA is still provable when we replace all instances of $\bot$ with A. For every rule except those which explicitly mention $\bot$, this is easy (since $\bot$ is just a proposition). And since $\bot$-elimination is not used in a Kolmogorov-translated proof (given as above), the only rule we must consider is monotonicity of $S(\bullet)$:

Theorem 2 (Friedman's A-Translation) If $\vdash_{PA} \phi$, then $\vdash_{HA} \phi^\circ[A/\bot]$, for some fresh propositional symbol A.

Proof: By Theorem 1, we have $\vdash_{HA} \phi^\circ$. Given a proof of $S(x) = 0 \vdash_{HA} \bot$, we can trivially show $S(x) = 0 \vdash_{HA} A$, by employing $\bot$-elimination. [4] ■

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

[4] From an arbitrary classical proof, we double-negate to arrive at a constructive proof without instances of $\bot$-elimination, and then A-translate to arrive at a constructive proof which might have instances of $\bot$-elim.

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

$$\underset{\phi\,\phi}{\neg\neg}(\exists y \in \mathsf{N}.\underset{\phi\,\phi}{\neg\neg}(f(y) = 0)).$$

To recover a proof of ϕ, it suffices to prove

$$\underset{\phi}{\neg}(\exists y \in \mathsf{N}.\underset{\phi\,\phi}{\neg\neg}(f(y) = 0)),$$

which we refer to as "Friedman's top-level trick." This is trivial, and is discussed in subsection 4.1. ■

Interestingly, the A-translation step does nothing to the computational content of a constructive proof, if *that proof was obtained via the Kolmogorov-translation* we outlined above. That is, A-translation after Kolmogorov-translation does nothing more to the computational content than just Kolmogorov-translation. The only effect of the A-translation is to "change the names," and render the proof in a form to which we can apply Friedman's top-level trick; all the real work is done by the double-negation translation. The reader might note that in A-translating the monotonicity rule, we inserted an instance of $\bot$-elimination. But going from the inference $M : S(x) = 0 \vdash_{HA} axiom : \bot$ to $M : S(x) = 0 \vdash_{HA} axiom : A$
(via $axiom : \bot \vdash_{HA} axiom : A$) does not change the computational content at all.

Friedman's original theorem is stated for Π_2^0 sentences, and is a corollary of the previous theorem, where free integer variables are allowed, and is stated:

Corollary 1 (Conservative Extension for Π_2^0) If we have a proof $\vdash_{PA} \phi$, where ϕ is Π_2^0, then we can construct a proof $\vdash_{HA} \phi$.

In the work to follow, as with Friedman's work, the result for Π_2^0 sentences follows trivially from the result for Σ_1^0 sentences, by allowing free numeric variables.

4 Directly Extracting Programs from PA Proofs

Since we can extract evidence from HA proofs, we can extract the evidence for ϕ by translating a classical proof into a constructive one and then extracting from that. Hence, by a roundabout, but completely mechanizable, translation we have succeeded in extracting evidence for a Σ_1^0 sentence from its classical proof. But this method produces a program whose structure bears little superficial resemblance to that of the original classical proof. One wonders if there is a more direct way of getting at this evidence. Since PA is a "simple" extension of HA, one wonders if perhaps one could augment (or modify) slightly the extraction procedure for HA to arrive at one for PA. This is indeed possible:

- In the same sense that HA proofs can be viewed directly as functional programs in $Prog_J$, PA proofs can be viewed directly as programs in $Prog_K \equiv Prog_J + \mathcal{C} + \lambda^v$, a functional language augmented with the nonlocal control operator $\mathcal{C}$ and by-value λ-abstractions.
- In the same sense that an HA proof M of $\exists x \in \mathsf{N}.\Phi(x)$ computes a numeral X such that $\Phi(X)$, a PA proof M of $\exists x \in \mathsf{N}.\Phi(x)$, *when Φ is primitive-recursive*, computes a numeral X such that $\Phi(X)$ holds.

Thus, classical proofs of Σ^0_1 (and by a simple extension, Π^0_2) sentences are programs. A Σ^0_1 sentence Φ is a specification, and a proof M of such a sentence is a program which meets this specification.

Let us now define the programming language $Prog_K$, as an extension of $Prog_J$, as follows: add the following four operators to the inductive definition of expressions:

$$Exp \equiv \lambda^v\ Var_1.Exp_1 \mid Exp_1({}_vExp_2) \mid \mathcal{C}Exp \mid \mathcal{A}Exp$$

The first two expressions are by-value lambda-abstraction and application, respectively. The λ-abstraction is a canonical form; the application is non-canonical when closed, and both its subterms are principal arguments (hence during evaluation, both subterms are evaluated, left-to-right, to canonical form before the application is contracted). We can define evaluation contexts as before. The latter two terms are the "control" operator and the "abort" operator, as defined in [1]. Given the evaluation context definitions metioned, one can express the operational semantics of $\mathcal{A}$ and $\mathcal{C}$ informally as

$$\begin{array}{lcl} E[\mathcal{C}M] & \succ_1 & M(\lambda\ x.\mathcal{A}(E[x])) \\ E[\mathcal{A}M] & \succ_1 & M. \end{array}$$

where $E[\ ]$ is an evaluation context. ($\mathcal{A}M$ is syntactic sugar for $\mathcal{C}(\lambda\ ().M)$. For the purposes of typing, though, we will ignore the presence of $\mathcal{A}$.) We will write $\succ_K$ for complete evaluation in the extended language, and $\succ_1$ for one step of evaluation (in the terminology of evaluation contexts). These two operators are related to call/cc [15], Landin's J-operator [18], Reynold's *escape* operator [19], PAL [20].

It should be clear that we have defined a deterministic evaluator. We now present an extraction algorithm which will directly extract the program from a classical proof. It turns out that the extraction algorithm is a slight modification and extension of the standard one for HA (part of which has already been presented). The original versions of all the modified rules have been presented earlier. The modified versions are:

Symmetry
$n : \mathsf{N},\ m : \mathsf{N},\ u : n = m$
$\quad \vdash (\lambda^v\ u'.axiom)({}_vu) : m = n$

Transitivity
$a, b, c : \mathsf{N},\ u : a = b,\ v : b = c$
$\quad \vdash (\lambda^v\ u', v'.axiom)({}_vu)({}_vv) : a = c$

Monotonicity of $S(\bullet)$
$n : \mathsf{N},\ u : S(n) = 0 \vdash (\lambda^v\ u'.axiom)({}_vu) : \bot$

Injectivity of $S(\bullet)$
$x, y : \mathsf{N},\ u : x^D = y^D$
$\quad \vdash (\lambda^v\ u'.axiom)({}_vu) : S(x)^D = S(y)^D$

Surjectivity of $S(\bullet)$
$x, y : \mathsf{N},\ u : S(x)^D = S(y)^D$
$\quad \vdash (\lambda^v\ u'.axiom)({}_vu) : x^D = y^D$

The extension consists of a single rule, that of double-negation elimination, and is taken directly from Griffin [8]:

Double Negation Elimination
$H \vdash \mathcal{C}M : P$
`BY double negation elim`
$\vdash M : \neg\neg(P)$

The idea here is simple. A single step of evaluation of well-typed PA programs does *not* always preserve well-typed-ness (the so-called "subject reduction" property). The reason is as follows: a closed term of type $a = b$ could contain unevaluated continuations. Such a term, when actually evaluated to a value, could discard its evaluation context. (It is possible to recover subject reduction properties, but this requires a pre-A-translation which would complicate the presentation, and which is technically not necessary for our result.) Thus to simply assume that u in the rule of symmetry of equality will indeed compute to *axiom* is unsound. Rather, we must evaluate u to a value, at which point we know that the value will be *axiom*. This is exactly what $(\lambda^v\ u'.axiom)({}_vu)$ does.

Now that we have defined a method of extracting programs, we must show that this method, applied to complete PA proofs of Σ^0_1 sentences, extracts terminating, well-typed programs. Let us first define the following translations on types and program fragments:

Definition 6 (CPS-Translation on Types)
Define the CPS-translation of a type T in a proof of a Σ_1^0 proposition ϕ by $(T[\phi/\bot])^\circ[\phi/\bot]$. We denote this translation by $\overline{T}$. Define also the "star-translation" of a type to be the CPS-translation with the outermost double-ϕ-ation removed. That is, $\underset{\phi}{\neg}\underset{\phi}{\neg}(T^*) = \overline{T}$.

Definition 7 (CPS-Translation on Programs)

$$
\begin{array}{rcl}
\underline{x} & \equiv & x \ (x \ \mathit{a\ variable}) \\
\underline{MN} & \equiv & \lambda\, k.\underline{M}(\lambda\, m.m\underline{N}k) \\
\underline{M(N^D)} & \equiv & \lambda\, k.\underline{M}(\lambda\, m.m(N^D)k) \\
\underline{\lambda\, x.M} & \equiv & \lambda\, k.k(\lambda\, x.\underline{M}) \\
\underline{\lambda\, v^D.M} & \equiv & \lambda\, k.k(\lambda\, v^D.\underline{M}) \\
\underline{\langle M, N\rangle} & \equiv & \lambda\, k.k(\langle \underline{M}, \underline{N}\rangle) \\
\underline{\langle M^D, N\rangle} & \equiv & \lambda\, k.k(\langle M^D, \underline{N}\rangle) \\
\underline{\mathit{inl}(M)} & \equiv & \lambda\, k.k(\mathit{inl}(\underline{M})) \\
\underline{\mathit{inr}(M)} & \equiv & \lambda\, k.k(\mathit{inr}(\underline{M})) \\
\underline{\mathit{axiom}} & \equiv & \lambda\, k.k(\mathit{axiom}) \\
\underline{M({}_vN)} & \equiv & \lambda\, k.\underline{M}(\lambda\, m.\underline{N}(\lambda\, n.m(n)(k))) \\
\underline{\lambda^v\, x.M} & \equiv & \lambda\, k.k(\lambda\, x.\underline{M}) \\
\underline{\mathcal{A}M} & \equiv & \lambda\, k.(\underline{M}(\tau_\phi)) \\
\underline{\mathcal{C}M} & \equiv & \lambda\, k.\underline{M}(\lambda\, m.m(\lambda\, g.g(\lambda\, v, h.v(k)))\tau_\phi).
\end{array}
$$

$$
\begin{array}{l}
\underline{\mathit{spread}(M; u, v.N)} \ \equiv \\
\qquad \lambda\, k.\underline{M}(\lambda\, p.\mathit{spread}(p; u, v.\underline{N}(k))) \\
\underline{\mathit{spread}(M; u^D, v.N)} \ \equiv \\
\qquad \lambda\, k.\underline{M}(\lambda\, p.\mathit{spread}(p; u^D, v.\underline{N}(k))) \\
\underline{\mathit{decide}(M; u.N; v.R)} \ \equiv \\
\qquad \lambda\, k.\underline{M}(\lambda\, d.\mathit{decide}(d; u.\underline{N}(k); v.\underline{R}(k))) \\
\underline{\mathit{ind}(M; B; n, i.I(n-1)(i))} \ \equiv \\
\qquad \mathit{ind}(M; \underline{B}; n, i.\underline{I(n-1)(i)})
\end{array}
$$

(Again, terms annotated $\bullet^D$ are integer expressions. The annotation is statically inferrable from the proof which generated the program)

The idea is to replace all instances of $\bot$ in the type T with ϕ, Kolmogorov-translate, and then again replace all instances of $\bot$ with ϕ. Since ϕ is assumed to be $\bot$-free ($\phi \equiv \exists x.\ f(x) = 0$ contains no instances of $\bot$), CPS-translating any type results in one with no instances of $\bot$. We can easily show

Theorem 4 (CPS is Well-Typed) Given a proof $\Gamma \vdash_{PA} M : T$, we can show $\overline{\Gamma} \vdash_{HA} \underline{M} : \overline{T}$, where $\overline{\Gamma}$ CPS-translates every proposition in Γ, but leaves type assumptions (e.g. $n : \mathsf{N}$) as is.

Proof: By induction on the typing derivation for M. We omit the proof, since it is entirely routine, given the CPS-translations on terms and types. ∎

Now we can map out our strategy to show that our extraction method is sound:

- Exhibit a term τ_ϕ of type $\underset{\phi}{\neg}(\exists y \in \mathsf{N}.\underset{\phi}{\neg}\underset{\phi}{\neg}(f(y) = 0))$ (the type of Friedman's top-level trick). Notice that $\underline{M}\tau_\phi$ has type ϕ. (subsection 4.1.) The reader will note that we have already used this term in our presentation of the translations of $\mathcal{C}$ and $\mathcal{A}$ above. The term is defined independently of CPS-translation, though, we are not caught in any circular definitions.
- Show that $\underline{M}\tau_\phi \succ_J b \Leftrightarrow M \succ_K b$ (Theorem 6.)

Given these facts, we can easily show:

Theorem 5 (Σ_1^0 Classical Type-Soundness) Given a PA proof of a Σ_1^0 sentence ϕ, $\vdash_{PA} M : \phi$, $M \succ_K b$ and $\vdash_{PA} b : \phi$.

Proof: Since $\underline{M}\tau_\phi$ has type ϕ in HA, evaluation of $\underline{M}\tau_\phi$ will terminate, say, in b. Then $\vdash_{PA} b : \phi$. And since $\underline{M}\tau_\phi \succ_J b \Leftrightarrow M \succ_K b$, $M \succ_K b$. ∎

It is a trivial extension to show

Corollary 2 (Π_2^0 Classical Type-Soundness) Given a PA proof of a Π_2^0 sentence $\forall x.\ \exists y.\ \psi(x, y)$, $\vdash_{PA} M : \forall x.\ \exists y.\ \psi(x, y)$, for all $n \in \mathsf{N}$, $M(n) \succ_K b$, hence $\vdash_{PA} b : \exists y.\ \psi(n, y)$. Thus PA proofs of Σ_1^0 and Π_2^0 sentences stand as evidence for the propositions they prove.

In the rest of this section, we give proofs of the two facts mentioned above, at widely varying levels of detail.

4.1 Top-Level Continuations

As we mentioned above, from a proof $\Gamma \vdash_{PA} M : T$, we can construct a proof $\overline{\Gamma} \vdash_{HA} \underline{M} : \overline{T}$. Thus, from a classical proof of a Σ_1^0 sentence $\phi \equiv \exists x \in \mathsf{N}.f(x) = 0$ (any Σ_1^0 sentence in PA can be written in this form), we get a constructive proof of $\underset{\phi}{\neg}\underset{\phi}{\neg}(\exists x \in \mathsf{N}.\underset{\phi}{\neg}\underset{\phi}{\neg}(f(x) = 0))$. To recover a proof of ϕ, we need only give a constructive proof of

$$\underset{\phi}{\neg}(\exists x \in \mathsf{N}.\underset{\phi}{\neg}\underset{\phi}{\neg}(f(x) = 0)). \tag{1}$$

ϕ then follows by modus ponens. The constructive witness for (1) is simply:

$$\tau_\phi \equiv \lambda\, p.\mathit{spread}(p; n, v.v(\lambda\, m.\langle n, m\rangle)). \tag{2}$$

It should be clear that if we CPS-translate a program M, to get a program $\underline{M}$, and if M had type ϕ, then $\underline{M}$ has type $\overline{\phi}$, and that $\underline{M}\tau_\phi$ has type ϕ.

Since every primitive-recursive relation on integers can be expressed as a primitive recursive function, restricting ourselves to the above form entails no loss of generality. But in any case, with a little care, we can recursively define the top-level-continuation for an arbitrary primitive recursive relation $\psi(x)$, rather than just $f(x) = 0$.

The topic of top-level continuations bears some discussion, since the literature has always assumed that a top-level continuation would be $\lambda\ x.x$. This assumption is predicated on the belief that the result of a computation is always computed by-value, or, alternatively, that the result of a computation is an atomic value. In fact, there is an intimate relation between top-level continuations and the semantics of complete evaluation for structured values which are evaluated in a lazy manner internally, but eagerly at top-level (as in any lazy system). Consider a noncanonical term of type $\exists x \in \mathsf{N}.f(x) = 0$. Once it is evaluated to a canonical form, $\langle N,\ G\rangle$, the type system guarantees that N is a pure integer expression. But G might contain unevaluated continuations, so we must further evaluate it to a canonical form. Thus if we think of $\underline{M}$ as a noncanonical form, then applying $\underline{M}\tau_\phi$ results in τ_ϕ being applied to the canonical form of M, say $\langle N,\ G\rangle$. Then (loosely speaking) we bind $n \mapsto N$, $v \mapsto G$, and evaluate G to a canonical form by applying to $\lambda\ m.\langle n,\ m\rangle$. Then m is bound to the canonical form of G, and the expression $\langle n,\ m\rangle$ is returned. This corresponds (in a way that can be made precise) to the process of complete evaluation described in Definition 4.

If, instead, $\phi\ =\ \exists x \in \mathsf{N}.f(x) = 0 \wedge g(x) = 0$, then we would have two choices of top-level continuation, shown in Figure 2. Both are proofs of $\phi^* \Rightarrow \phi$, or

$$\underset{\phi}{\neg}(\exists x \in \mathsf{N}.\underset{\phi\,\phi}{\neg\neg}(\Big(\underset{\phi\,\phi}{\neg\neg}(f(x) = 0) \wedge \underset{\phi\,\phi}{\neg\neg}(g(x) = 0)\Big))).$$

Both correspond to

1. evaluating a program M of type ϕ to a canonical form, $\langle N,\ G\rangle$, where N is an integer expression
2. evaluating G to a canonical form $\langle A,\ B\rangle$.
3. evaluating A,B, to canonical forms, say L,R.
4. returning $\langle N,\ \langle L,\ R\rangle\rangle$.

But (3) evaluates A to a canonical form, and *then* B, whereas (4) does these in the opposite order (note that the only difference between (3) and (4) is that a, l and b, r have been switched). All of this can be made precise with relative ease, and we can draw direct correspondences between methods of complete evaluation for structured values on the one hand, and top-level continuations on the other.

4.2 CPS-Translation Preserves Values

So far, we have shown that given a program expression M of type ϕ, a Σ_1^0 type, CPS-translation followed by application to the proper top-level continuation gives us a program expression $\underline{M}\tau_\phi$ of the same type ϕ. To finish the job, we need to show one more thing; to wit, that $\underline{M}\tau_\phi$ evaluates to a value b if and only if M, the term with instances of $\mathcal{C}$, evaluates to the same value b. The method of proof is taken from Griffin [8], who in turn adapted Plotkin [21].

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 $\underline{M}\tau_\phi$, 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 redices in $\underline{M}\tau_\phi$ (to M') and $\underline{N}\tau_\phi$ (to N'), 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 $M \succ_K N$, $M : \tau_\phi \succ_J N : \tau_\phi$. The proof is long and unilluminating, employing the colon-translation originated by Plotkin [21] and modified by Griffin [8]; hence we will eschew the proof itself, and just state the theorem. In any case, it is presented in detail in [13]. The upshot of the whole exercise is to obtain a proof of

Theorem 6. (CPS Preserves Operational Semantics) For any program M of (Σ_1^0) type ϕ, for all values $b\ \in\ \phi$,

$$M \succ_K b \Leftrightarrow \underline{M}\tau_\phi \succ_J b.$$

5 Simple Examples

We can now look at some simple examples. Consider the sentence

$$\phi \equiv \exists n \in \mathsf{N}.prime(n) \wedge n < 100.$$

Clearly there are many proofs of this sentence. We list one such proof below:

$$\tau_\phi \equiv \lambda\ p.spread(p; n, m.m(\lambda\ q.spread(q; a, b.a(\lambda\ l.b(\lambda\ r.\langle n,\ \langle l,\ r\rangle\rangle))))) \quad (3)$$

$$\tau_\phi \equiv \lambda\ p.spread(p; n, m.m(\lambda\ q.spread(q; a, b.b(\lambda\ r.a(\lambda\ l.\langle n,\ \langle l,\ r\rangle\rangle))))) \quad (4)$$

Figure 2: Top-Level Continuations for $\phi \equiv \exists x \in \mathsf{N}.f(x) = 0 \wedge g(x) = 0$

$\vdash\ \phi$ **BY double-negation elim**
 $\neg(\phi) \vdash\ \perp$ **BY function-elim**
  $\neg(\phi) \vdash\ \phi$ **BY intro** 102
   $\vdash\ prime(102) \wedge 102 < 100$ **BY and-intro**
    $\vdash\ prime(102)$
     BY double-negation elim
      THEN function elim
     $\vdash\ \phi$ **BY intro 2, etc**
    $\vdash\ 102 < 100$
     BY double-negation elim
      THEN function elim
     $\vdash\ \phi$ **BY intro 3, etc**

If we let π_2 be the proof of $prime(2) \wedge 2 < 100$, and π_3 the proof of $prime(3) \wedge 3 < 100$, then we can write the computational content of this proof as

$$\mathcal{C}\lambda\ k.k\langle 102,\ \langle k\langle 2,\ \pi_2\rangle,\ k\langle 3,\ \pi_3\rangle\rangle\rangle.$$

Here we see a simple example of the "nondeterminism" inherent in classical reasoning. There are two possible values of n specified by this proof (under a naive reading): 2 and 3. In fixing a deterministic left-to-right evaluator, we choose π_2. If we were to fix a right-to-left complete evaluation strategy for pairs (as in Figure 2, equation 4), we would choose π_3 instead).

We can also look at a simple classical proof which does not admit constructivization: assume $f : \mathsf{N} \to \{0, 1\}$ is a parameter. There is a simple classical proof of

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

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 $f : n \longmapsto 1$ is industinguishable 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 $f : n \longmapsto 1$ and determine that it attained a minimum of 1 after N 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:

$\vdash\ \psi$ **BY double-negation elim**
 $\neg(\psi) \vdash\ \perp$ **BY function-elim**
  $\vdash\ \psi$ **BY intro 0**
   $\vdash\ \forall m \in \mathsf{N}.f(0) \leq\ f(m)$ **BY function intro**
    $m : \mathsf{N} \vdash\ f(0) \leq\ f(m)$ **BY cases on** $f(0) \leq\ f(m)$
     $f(0) \leq\ f(m) \vdash\ f(0) \leq\ f(m)$ **BY hypothesis**
     $f(m)\ <\ f(0) \vdash\ f(0) \leq\ f(m)$
      BY double-negation elim
       THEN function-elim
      $f(m)\ <\ f(0) \vdash\ \psi$ **BY intro** m**, etc**

The computation in this proof is

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

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.

6 Conclusions and Related Work

We have seen that we can directly extract computational evidence from classical proofs in arithmetic. This evidence is obtained by a procedure which is a slight modification/extension of the standard extraction procedure for constructive proofs in arithmetic. The proof of soundness of extraction employed a CPS-translation on types which translated a type T in a proof of a Σ^0_1 proposition ϕ to $(T[\phi/\perp])^\circ[\phi/\perp]$. The original Friedman A-translation produced a type $T^\circ[\phi/\perp]$. We assumed that ϕ was Σ^0_1, which is $\perp$-free, so the two translations are identical in their effect on the top-level type. Within the proof, the only difference is that with Friedman's translation, every instance of $\perp$ in the original proof is translated into $\underset{\phi\,\phi}{\neg\neg}(\phi)$, whereas with our CPS-translation, every instance of $\perp$ becomes $\overline{\phi} \equiv \underset{\phi\,\phi}{\neg\neg}(\phi^*)$. This difference is inconsequential, since we could have easily imagined that Friedman used our CPS-translation rather than the one he actually used. The internal differences are slight, and affect only the translation of $\mathcal{C}$. Thus it should be obvious now that at least one (and in fact several) implementations of Friedman's results are CPS-translations. In fact, a minor variant of the Kuroda negative translation can also be used to prove these results, and the only consequence is that $\succ_K$ is call-by-value rather than call-by-name.

Moreover, we conjecture that by generalizing Reynolds' [19] notion of *trivial* and *serious* terms, one can interpret the standard translations in the literature as fixing values of classical proofs. That is, while most CPS-translations fix evaluation order completely (there is at most one closed redex (noncanonical term) in a CPS-translated term at any point in its evaluation), the translation is this paper fixes the order of non-integer expressions only. Since integer expressions are "trivial" (can have no control side-effects), they are left untranslated. We conjecture that the standard translations can also be interpreted in a like manner, by generalizing further the notion of "trivial" to include terms of non-integer type which can be shown to have no control side-effects.

Although implicit in the results of Friedman and in the translations due to Gödel, Kolmogorov, and others, no one in the computer science community had exploited the computational significance of the fact that classical proofs of Π_2^0 sentences (i.e. $\forall x.\ \exists y.\ R(x, y)$ where R is decidable) can be translated into constructive proofs of the same sentences.

Fortune, Leivant, and O'Donnell [22] found that a typed CPS-translation effected a double-negation-like translation. This was rediscovered (accidentally) by Meyer and Wand [23], who also found that typed CPS-translation effected a double-negation translation. Likewise, Krivine [24] found that a double-negation translation on proofs in the polymorphic lambda-calculus effected a CPS-like translation on the implicit programs. Independently, Gabbay and Reyle [25,26] found that an extension of PROLOG with a "restart rule" much like the $\mathcal{A}$ operator was sound and complete with respect to classical propositional logic, and an analogous result for a subtheory of classical predicate logic. In completely independent work, Adriaan Rezus [27] discovered a variant of Griffin's result, working from the angle of understanding equational proof behaviours. He discovered a relative of the $\mathcal{C}$ operator by examining classical proofs and characterizing which proofs ought to be equal as programs. However, he was not concerned at that stage with arriving at evidence semantics, and indeed, his logic contained as primitives only $\vee, \rightarrow, \forall$.

Griffin [8] found an algorithmic interpretation of classical proofs, or, alternatively, a logical characterization of the control operator $\mathcal{C}$. He showed that one could view $\mathcal{C}$ as the algorithmic content (program extraction) of the rule of double-negation elimination $(\neg\neg(P) \vdash P)$ in classical propositional logic. Griffin went on to show that the process of CPS-translation was a translation from a classical propositional logic into a constructive propositional logic. Griffin demonstrated that such a CPS-translation preserved the operational semantics of the term it was translating. But his typing was only a weak typing, and did not guarantee that a program of type ϕ would compute a value of type ϕ. In addition, as we have mentioned, he defined conjunction (and disjunction) in terms of implication and negation (e.g. $\alpha \wedge \beta \equiv \neg(\alpha \rightarrow \neg(\beta))$) and as such, the extent to which a proof $A \wedge B$ actually computed evidence for $A \wedge B$ was not determined.

He had (in effect) reconstructed a part of Friedman's conservative extension result [2] from HA to PA for Π_2^0 sentences. He did not apply the last step, that of A-translation, nor make the connection between top-level continuations, Friedman's top-level trick, and complete evaluation strategies. He also restricted his attention to propositional logic (or, equivalently, simply-typed lambda-calculus). To quote Griffin:

> This paper has shown that a formulae-as-types correspondence can be defined between classical propositional logic and a typed Idealized Scheme containing a control operator similar to Scheme's **call/cc**. It should be noted, however, that the paper merely presents a *formal* correspondence ... At this point there still remains the question: *Why should there be any correspondence at all?* Whether or not there is a deeper reason underlying the correspondence is unclear at this time.

Griffin's work pointed towards a deeper logical and computational content behind Friedman's result and also the classic results concerning CPS-translation.

We feel that our results explain the correspondence. We feel we have shown that classical proofs are as real a programming language as constructive proofs. From every classical proof we can extract a program, by the rules we described earlier. However, only for Σ_1^0 and Π_2^0 sentences are we guaranteed that every classical proof computes evidence. To put it simply, *every classical proof is a program, but only proofs of Σ_1^0 and Π_2^0 sentences are always correct.* Our results show that classical logic is a programming language much like constructive logic, but with explicit control operations.

We have shown that CPS-translation is not just a double-negation translation, but a double-negation/A-translation (a "Friedman" translation). We have shown that Friedman's conservative extension result is simply (and beautifully) the application of a CPS-translation soundness theorem to an almost-functional program extracted from a classical proof, demonstrating that CPS-translation and "Friedman" translation are one and the same, and that from classical proofs we may directly extract correct, well-typed programs. By employing A-translation, we can not only show that a classical proof computes evidence, but also determine what sort of pathological behaviour a nonconstructivizable proof can exhibit - that is, what types of values a nonconstructivizable proof can abort with.

In summary, every classical proof can be assigned a program, but only for Π_2^0 sentences ϕ do these programs always compute *evidence* in the constructive sense [28] for ϕ. From a programming languages/specification viewpoint, our work provides a typed foundation for total-correctness reasoning about programs with $\mathcal{C}$, much as constructive type theories [14,29] provide a typed foundation for reasoning about functional programs.

The full consequences of this run deep: Not only can we regard classical proofs in PA (or other suitable theories) as a programming language that makes *logical* use of jumps and continuations, but we are able, in the process, to extend the Curry-Howard interpretation of functional computation to include such programming constructs as $\mathcal{C}$ and call/cc, and reason about correctness of programs using them in a semantically clear manner.

7 Future Work

The possibilities for future work are numerous. We list only a few here:

- Discover the algorithmic ideas inherent in the many constructivizable classical proofs which exist.

- Extend these results to richer type theories, such as Nuprl. The logic we presented here is only the core of a rich programming logic. While we can in theory extract computations from Classical Nuprl proofs (via translation), our results show us how to extract computations directly from the classical proofs. One could also apply work of Felleisen and Hieb [30] to yield a logic for reasoning about equalities between programs with $\mathcal{C}$, much as Nuprl allows one to reason about equalities between functional programs.

- There are many double-negation translations in the literature, and it would be interesting to learn exactly what sorts of evaluation semantics these translations correspond to.

- Many people are working on new and powerful control operators [31,32,33], and one could give reasoning systems (both ML-style and total-correctness reasoning systems) for languages with some of these operators in a like manner to what has been done herein. We conjecture that the shift/reset system of Danvy & Filinski [31] can be reasoned about with extensions of the techniques presented in this paper.

Acknowledgements

I thank Matthias Felleisen, who had a great impact on the form of this paper, both by reading it in a very rough version, and by improving the entire presentation of the results. I thank Tim Griffin for his thorough and careful reading of this paper, and also for many long conversations, electronic and verbal, about this work. I also thank Bill Aitken, Olivier Danvy, and John Reppy for their careful reading and corrections, and Doug Howe for countless hours of conversations and sound advice. Thanks are also due to the referees, whose comments tightened up the paper considerably. Finally, I thank Robert Constable, whose guidance made this work possible.

References

[1] M. Felleisen, D. Friedman, E.Kohlbecker, and B. Duba, "Reasoning with continuations," in *Proceedings of the First Annual Symposium on Logic in Computer Science*, pp. 131–141, 1986.

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

[3] G. Kreisel, "Mathematical significance of consistency proofs," *Journal Of Symbolic Logic*, vol. 23, pp. 155–182, 1958.

[4] W. Howard, "The formulas-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism* (J. P.

Seldin and J. R. Hindley, eds.), pp. 479–490, NY: Academic Press, 1980.

[5] R. L. Constable, "Constructive mathematics and automatic program writers," in *Proc. IFP Congr.*, (Ljubljana), pp. 229–33, 1971.

[6] E. Bishop, "Mathematics as a numerical language," in *Intuitionism and Proof Theory* (e. a. J. Myhill, ed.), pp. 53–71, North-Holland, 1970.

[7] R. L. Constable, et al., *Implementing Mathematics with the Nuprl Proof Development System.* Englewood Cliffs, New Jersey: Prentice-Hall, 1986.

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

[9] M. J. Fischer, "Lambda-calculus schemata," in *Proceedings of the ACM Conference on Proving Assertions about Programs*, vol. 7 of *Sigplan Notices*, pp. 104–109, 1972.

[10] K. Gödel, "On intuitionistic arithmetic and number theory," in *The Undecidable* (Davis, M., ed.), pp. 75–81, Raven Press, 1965.

[11] D. Prawitz, "Ideas and results in proof theory," in *Studies in Logic and the Foundations of Mathematics*, pp. 235–307, Amsterdam: North-Holland, 1971.

[12] M. Felleisen and D. Friedman, "Control operators, the SECD machine and the λ-calculus," in *Formal Description of Programming Concepts III*, pp. 131–141, North-Holland, 1986.

[13] C. Murthy, *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Department of Computer Science, 1990.

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

[15] W. Clinger and J. Rees, "The *revised*^{3} report on the algorithmic language scheme," *SIGPLAN Notices*, vol. 21, no. 12, pp. 37–79, 1986.

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

[17] D. Leivant, "Syntactic translations and provably recursive functions," *Journal Of Symbolic Logic*, vol. 50, no. 3, pp. 682–688, 1985.

[18] P. J. Landin, "A correspondence between Algol-60 and Church's lambda notation," *Communications of the ACM*, vol. 8, pp. 89–101, 1965.

[19] J. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the 25th ACM National Conference*, pp. 717–740, ACM, 1972.

[20] A. Evans, "PAL - A language designed for teaching programming linguistics," in *Proceedings of the ACM 23rd National Conference*, pp. 395–403, 1968.

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

[22] S. Fortune, D. Leivant, and M. O'Donnell, "The expressiveness of simple and second-order type structures," *J. ACM*, vol. 30, pp. 151–185, Jan. 1983.

[23] A. R. Meyer and M. Wand, "Continuation semantics in typed lambda-calculi (summary)," in *Logics of Programs* (R. Parikh, ed.), vol. 193 of *Lecture Notes in Computer Science*, pp. 219–224, Springer-Verlag, 1985.

[24] J. L. Krivine, "Opérateurs de mise en mémoire et traduction de gödel," *Archiv. for Math. Logic*, no. 30, pp. 241–267, 1990.

[25] D. Gabbay and U. Reyle, "N-Prolog: An extension of Prolog with hypothetical implications. I.," *Journal of Logic Programming*, vol. 4, pp. 319–355, 1984.

[26] D. Gabbay, "N-Prolog: An extension of Prolog with hypothetical implications. II. Logical foundations, and negation as failure," *Journal of Logic Programming*, vol. 4, pp. 251–283, 1985.

[27] A. Rezus, "Classical proofs (lambda calculus methods in elementary proof theory)," 1990. Unpublished draft.

[28] R. Constable, "The semantics of evidence," Tech. Rep. TR 85-684, Cornell University, Department of Computer Science, Ithaca, New York, May 1985.

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

[30] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," Tech. Rep. 100, Rice, Houston, 1989.

[31] O. Danvy and A. Filinski, "Abstracting control," in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160, 1990.

[32] M. Felleisen, "The theory and practice of first-class prompts," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 180–190, 1988.

[33] M. Felleisen and D. Sitaram, "Control delimiters and their hierarchies," *Lisp and Symbolic Computation*, vol. 3, pp. 67–99, 1990.