Non-Constructive Computational Mathematics

Kenneth Kunen [1]
University of Wisconsin, Madison, WI 53706, U.S.A.
kunen@cs.wisc.edu

August 28, 1995

ABSTRACT

We describe a non-constructive extension to Primitive Recursive Arithmetic, both abstractly, and as implemented on the Boyer-Moore prover. Abstractly, this extension is obtained by adding the unbounded μ operator applied to primitive recursive functions; doing so, one can define the Ackermann function and prove the consistency of Primitive Recursive Arithmetic. The implementation does not mention the μ operator explicitly, but has the strength to define the μ operator through the built-in functions `EVAL$` and `V&C$`.

§1. INTRODUCTION

This paper is a mixture of theory and practice.

The **theory** begins with the notions of *constructivism* and *finitism* in the philosophy of mathematics. As with all philosophical notions, these cannot appear directly in a mathematical theorem or a computer program, but they have been useful guides over the past hundred years to discovering mathematical results, and more recently, to designing computer implementations.

Informally, a *constructivist* only believes in objects for which there is an explicit construction; in particular, a function definition is meaningful only if it is accompanied by a procedure for computing values of the function. Axiom systems such as ZF (Zermelo-Fraenkel set theory) are obviously non-constructive, but even the system PA (Peano Arithmetic), which deals only with natural numbers, is non-constructive. For example, if $\phi(x, y)$ is some property of x, y, constructivists would not assert that they could prove

$$\forall x \big(\exists y \phi(x, y) \vee \forall y \neg \phi(x, y)\big) \tag{1}$$

unless they could define a function f and prove that

$$\forall x \big(\phi(x, f(x)) \vee \forall y(\neg \phi(x, y))\big) \tag{2}$$

Note that (2) is equivalent to

$$\forall x, y \big(\phi(x, y) \Rightarrow \phi(x, f(x))\big) \tag{2'}$$

[1] Author supported by NSF Grant CCR-9503445. The author is grateful to R. S. Boyer for his explanations of the Boyer-Moore prover.

and this equivalence is constructive if ϕ is decidable, so we may view f as a Skolem function for ϕ.

Now, for a constructivist, "function" presumably means "recursive function" (by Church's Thesis). However, it is easy to find even primitive recursive ϕ such that no total recursive f satisfies $(2')$; for these ϕ, (1) would be false to a constructivist. However, (1) is immediate from the law of the excluded middle in classical logic, so it is trivially provable in PA. Furthermore, within PA, it is easy to define a function f and prove $(2')$; just let $f(x)$ be $\mu y\, \phi(x,y)$ if $\exists y \phi(x,y)$ and 0 otherwise ($\mu y\, \phi(x,y)$ denotes the least y such that $\phi(x,y)$). A constructivist would simply not accept such a definition of a function. For further discussion of constructivism, see Beeson [2] or Troelstra [17].

Kleene [12] turned the above philosophical argument into a mathematical theorem. The system HA (Heyting Arithmetic) is an attempt to formalize constructive number theory. HA has exactly the same axioms as does PA, but allows only intuitionistic logic in its proofs. By [12], (1) is not provable within HA unless one can produce a recursive f and prove $(2')$; in particular, one can write down specific primitive recursive ϕ for which (1) is not provable in HA.

Now, *finitism* is an extreme version of constructivism. HA and PA have the same induction axiom:

$$\psi(0) \wedge \forall x(\psi(x) \Rightarrow \psi(x+1)) \quad \Rightarrow \quad \forall x \psi(x) \tag{3}$$

Here, ψ is an arbitrary logical formula. A true *finitist* would reject such an axiom, saying that a formula which quantifies over the infinite set of natural numbers does not in general have any meaning.

The logical system PRA (Primitive Recursive Arithmetic) is an attempt to formalize finitistic mathematics. PRA has a name for every primitive recursive function, g, and includes the recursive definition of g as an axiom. The logic of PRA does not explicitly write quantifiers, but universal quantification is understood, so that the theorems of PRA are all universally quantified validities about primitive recursive functions. Without quantifiers, induction in PRA can be formalized as a proof rule; for each quantifier-free ψ:

$$\text{if} \quad \vdash \psi(0) \quad \text{and} \quad \vdash \psi(x) \Rightarrow \psi(x+1) \quad \text{then} \quad \vdash \psi(x) \tag{4}$$

The book by Goodstein [9] develops PRA in quite a bit of detail, proving basic facts about number theory, through the uniqueness of prime factorization. Also, via Gödel numbering, one may prove within PRA most of the known mathematical theorems about finite structures. For example, one may prove Ramsey's Theorem, along with all the basic theorems on the structure of finite groups and fields. The papers by Parsons [15] and Sieg [16] say more about the proof-theoretic strength of PRA.

Still, PRA is weaker than HA. For example, the Ackermann function is a well-known recursive function which is not primitive recursive. So, this function cannot even be mentioned in PRA, whereas it is easy to define it and derive its basic properties in HA. Furthermore, the statement $CON(PRA)$ (that PRA is consistent) is easy to *express* within PRA, and is provable in HA but not in PRA. That $PRA \nvdash CON(PRA)$ follows by Gödel's Second Incompleteness Theorem, which applies to any theory such as PRA, HA, or PA. But $HA \vdash CON(PRA)$, since within HA, one may define a recursive function

universal for primitive recursive functions, and then formalize the proof that all sentences provable from PRA are true. Actually, this proof just uses a small piece of HA. If we form PRA' by adding to PRA a name, A, for the Ackermann function, and the axioms that A satisfies its usual definition, then $PRA' \vdash CON(PRA)$ (see Lemma 3.1.4).

To show that HA is weaker than PA, one must refer to Kleene's result, mentioned above, that instances of (1) are not provable in HA. By the Incompleteness Theorem, $PA \nvdash CON(HA)$, because $PA \nvdash CON(PA)$ and, by another result of Gödel [8], $CON(HA) \Leftrightarrow CON(PA)$ is provable in PA (and, in fact, in PRA).

Now, as already mentioned, "constructive" is a philosophical notion, not a mathematical one, and hence need not be constrained by the specific formal rules of HA or any other formal system. In particular, although $HA \nvdash CON(HA)$, there are two well-known proofs of $CON(HA)$ (equivalently, by [8], of $CON(PA)$), which many constructivists would accept. Kleene's [12] proof by recursive realizability formalizes the intuition that every statement provable from HA is constructively true. The other is due to Gentzen [7]. Let PRA'' be obtained by adding to PRA induction and recursion on the ordinal ϵ_0. This is clearly stronger than PRA', since the Ackermann function can be defined by recursion on ω^2. Gentzen showed that $PRA'' \vdash CON(PA)$. Although PA proves statements (e.g., versions of (1) above) which no constructivist would accept, many constructivists would accept the use of ϵ_0, and hence Gentzen's proof of $CON(PA)$. In fact, we see no reason why there could not be a constructive proof of $CON(ZF)$.

Aside from "logic" results, Paris and Harrington [14] describe a strengthening of Ramsey's Theorem which is provably equivalent to $CON(PA)$. This is a simple combinatorial statement which is provable in PRA'', but not in PA. A direct proof from PRA'' was given by Ketonen and Solovay [11]; see [13] for a simpler proof.

We turn now to **practice**. There are many computer systems available for verifying proofs in mathematical logic. One of the most well known among these is the system Nqthm, developed by Boyer and Moore [5], and described by them as a system of "computational logic". To first approximation, it is an implementation of PRA. It has been used to verify statements about the correctness of hardware and software design, as well as constructive theorems of pure mathematics.

Actually, Nqthm extends PRA in three important ways. First, it can verify theorems about finite symbolic expressions (Lisp S-expressions), as well as theorems about natural numbers. Second, it allows the definition of functions by recursion on the ordinal ϵ_0, and proofs by induction on ϵ_0. Third, it includes a self-referential feature, so that one may talk about the semantics of Nqthm within Nqthm itself.

Of these three extensions, the first is both the least important in theory and the most important in practice. Allowing symbolic entities in the formalism is clearly inessential, since one may simply code these entities by Gödel numbers, but it is also clear that a practical verification system could not rely on Gödel numbering. The other two extensions are definitely essential, in that each of them separately allows Nqthm to prove statements which are not provable in PRA. However, the applications of these extensions seem to be primarily in pure mathematics; we know of no useful (to engineers) statements about algorithms or digital circuits which go beyond PRA.

By the second extension, Nqthm really contains PRA''. The use of ordinals in Nqthm

itself was described in some detail in [13], including a verification on Nqthm of the Paris-Harrington Ramsey theorem. Still, as pointed out above, this extension remains "constructive" in some sense.

Now, the third extension to Nqthm is not constructive at all. In this paper, we show that this extension alone (without using the second extension) is stronger than had hitherto been suspected. This extension is implemented through axioms about the non-recursive functions `EVAL$` and `V&C$`. Although, Boyer and Moore [4,5] pointed out that the definition of these functions is non-constructive, it it was not clear from [4,5] that the introduction of these functions was in fact non-conservative over PRA – that is, using the axioms about these functions, one could prove a universal validity about primitive recursive functions which was not already provable in PRA.

The theoretical content of the third extension (without the second) is the system we shall call PRA^*. This is obtained by adding a function symbol f and equation $(2')$ above for each primitive recursive predicate ϕ. In PRA^*, one can define the Ackermann function and prove $CON(PRA)$. In particular, PRA^* is not conservative over PRA.

In §2, we make some preliminary remarks on recursion theory, and in §3, we discuss our theoretical results about PRA and related systems. Actually, the relationship between the theory (PRA^*) and practice (`EVAL$` and `V&C$` in Nqthm) is not completely straightforward; we discuss this further in §4.

§2. REMARKS ON RECURSION THEORY

§2.1. Names. The *partial recursive functions* form the least class containing some simple basic functions and closed under primitive recursion and the μ operator. Following Kleene [12] (roughly), each such function can be denoted by a *name*, which describes how it was constructed. Formally, the names are those symbolic entities built by applying the following rules; j, n, m all denote natural numbers:

Base names (Identity, Zero, Successor function, Wild cards):

1. I_j^n $(j < n)$ is a name of arity n.
2. Z^n $(0 \leq n)$ is a name of arity n.
3. S^1 has arity 1.
4. W_j^n $(0 < n)$ is a name of arity n.

Compound names (Composition, Primitive recursion, Mu operator):

5. $C_m^n(\sigma, \tau_1, \ldots, \tau_m)$ is a name of arity n whenever $n \geq 0$, $m \geq 1$, each τ_i is a name of arity n, and σ is a name of arity m.
6. $P^n(\sigma, \tau)$ is a name of arity n whenever $n \geq 1$, σ is a name of arity $n-1$, and τ is a name of arity $n+1$.
7. $M^n(\tau)$ is a name of arity n whenever $n \geq 0$ and τ is a name of arity $n+1$.

Note that the superscript denotes the *arity* of a name, which is always a natural number. We shall sometimes need some special kinds of names; specifically:

- *Primitive recursive names* are those built just using rules $1-6$.
- *Pure names* are those built using just rules $1-3$ and $5-7$.
- *Simple names* are those pure names built using rules $1-7$, but containing at most one symbol of the type M^n.

The primitive recursive names are used to name the primitive recursive functions, which can be defined without the use of the μ operator. The Kleene Normal Form Theorem (see Theorem 2.9.1 below) says that every partial recursive function can be named by a simple name. The wild cards are used primarily to describe relative computability; that is, the computation of one function, using another as an oracle. Actually, the class of pure names is just a minor convenience, since whenever we are not discussing relative computability, we could just as well allow the wild cards, but interpret them all to be the identically zero function.

§**2.2. Partial functions.** If $n > 0$, $(\omega^n \longrightarrow \omega)$ is the set of all functions from ω^n into ω, and $(\omega^n \xrightarrow{P} \omega)$ is the set of all functions from ω^n into $\omega \cup \{\perp\}$. So, $(\omega^n \longrightarrow \omega) \subset (\omega^n \xrightarrow{P} \omega)$. If $f \in (\omega^n \xrightarrow{P} \omega)$, we *think* of f as a *partial function*, and let $dom(f) = \{\vec{x} \in \omega^n : f(\vec{x}) \neq \perp\}$; we also extend f to a function on $(\omega \cup \{\perp\})^n$ by saying that $f(\vec{x}) = \perp$ whenever some component of $\vec{x}$ is $\perp$. To motivate this convention, think of f as being obtained by a computer program which expects n input values and attempts to output a value. Then, $f(\vec{x}) = \perp$ means that the computation doesn't halt. An input of $\perp$ means that the input never arrives, so that f certainly doesn't halt. These conventions are useful when discussing compositions of functions, where the input to f is the output of another function.

If $n = 0$, we think of a program which expects no input, so it just computes a number, if it halts. That motivates the definitions: $(\omega^0 \longrightarrow \omega) = \omega$ and $(\omega^0 \xrightarrow{P} \omega) = \omega \cup \{\perp\}$.

2.3. What the names name. An *oracle* is an indexed family, $\mathcal{F} = \{f_j^n : j < \omega, 0 < n < \omega\}$, of total functions, with $f_j^n \in (\omega^n \longrightarrow \omega)$, such that all but finitely many of the f_j^n are identically zero. $\Phi_\lambda^{\mathcal{F}} \in (\omega^n \xrightarrow{P} \omega)$ is defined as follows whenever λ is a name with arity n and $\mathcal{F}$ is an oracle.:

$\lambda = I_j^n$: $\Phi_\lambda^{\mathcal{F}}(\vec{x}) = x_j$.
$\lambda = Z^n$: $\Phi_\lambda^{\mathcal{F}}(\vec{x}) = 0$.
$\lambda = S^1$: $\Phi_\lambda^{\mathcal{F}}(x) = x + 1$.
$\lambda = W_j^n$: $\Phi_\lambda^{\mathcal{F}}(\vec{x}) = f_j^n(\vec{x})$
$\lambda = C_m^n(\sigma, \tau_1, \ldots, \tau_m)$: $\Phi_\lambda^{\mathcal{F}}(\vec{x}) = \Phi_\sigma^{\mathcal{F}}(\Phi_{\tau_1}^{\mathcal{F}}(\vec{x}), \ldots, \Phi_{\tau_m}^{\mathcal{F}}(\vec{x}))$.
$\lambda = P^n(\sigma, \tau)$: $\Phi_\lambda^{\mathcal{F}}(\vec{x}, 0) = \Phi_\sigma^{\mathcal{F}}(\vec{x})$. $\Phi_\lambda^{\mathcal{F}}(\vec{x}, y+1) = \Phi_\tau^{\mathcal{F}}(\vec{x}, y, \Phi_\lambda^{\mathcal{F}}(\vec{x}, y))$.
$\lambda = M^n(\tau)$: $\Phi_\lambda^{\mathcal{F}}(\vec{x}) = y \in \omega$ iff $\Phi_\tau^{\mathcal{F}}(\vec{x}, y) \in \omega \backslash \{0\}$ and for all $z < y$, $\Phi_\tau^{\mathcal{F}}(\vec{x}, z) = 0$.
$\Phi_\lambda^{\mathcal{F}}(\vec{x}) = \perp$ if there is no such y.

If each f_j^n is the identically zero function, we write Φ_λ for $\Phi_\lambda^{\mathcal{F}}$. If $f \in (\omega^m \longrightarrow \omega)$, we write Φ_λ^f for $\Phi_\lambda^{\mathcal{F}}$, where $f_0^m = f$, and the other f_j^n are identically zero. Likewise define $\Phi_\lambda^{f,g}$ when $f \in (\omega^m \longrightarrow \omega)$ and $g \in (\omega^k \longrightarrow \omega)$.

$g \in (\omega^n \xrightarrow{P} \omega)$ is *partial recursive* in $\mathcal{F}$ iff $g = \Phi_\lambda^{\mathcal{F}}$ for some n-ary name λ. g is *total recursive* (or just *recursive*) in $\mathcal{F}$ iff g is partial recursive in $\mathcal{F}$ and g is total (that is, $g \in (\omega^n \longrightarrow \omega)$). g is *primitive recursive* in $\mathcal{F}$ iff $g = \Phi_\lambda^{\mathcal{F}}$ for some n-ary primitive recursive name λ.

2.4. Specific examples. We show here that some standard functions are primitive recursive by our definition.

The constant function 2 of three variables is named by $C_1^3(S^1, C_1^3(S^1, Z^3))$.

Addition is defined by

$$x + 0 = h(x) \ ; \ x + (y + 1) = g(x, y, (x + y)) \quad ,$$

where $h(x) = x$ and $g(x, y, z) = z + 1$ so $+ = \Phi_{PLUS}$, where $PLUS$ is the name $P^2(I_0^1, C_1^3(S^1, I_2^3))$.

Multiplication is defined by

$$x * 0 = h(x) \ ; \ x * (y + 1) = g(x, y, (x * y)) \quad ,$$

where $h(x) = 0$ and $g(x, y, z) = x + z$. So, $* = \Phi_{TIMES}$, where $TIMES$ is $P^2(Z^1, C_2^3(PLUS, I_0^3, I_2^3))$, or $P^2(Z^1, C_2^3(P^2(I_0^1, C_1^3(S^1, I_2^3)), I_0^3, I_2^3))$. Likewise, if $FACT$ is the name $P^1(C_1^0(S^1, Z^0), C_2^2(TIMES, C_1^2(S^1, I_0^2), I_1^2))$, then the factorial function is Φ_{FACT}.

Other useful functions are $sgn(x)$ (the smaller of x and 1), given by the name $P^1(Z^0, C_1^2(S^1, Z^2))$, and $x \dot{-} y$ (the larger of 0 and $x - y$), given by the name $P^2(I_0^1, C_1^3(P^1(Z^0, I_0^2), I_2^3))$. One can then define $max(x, y) = x + (y \dot{-} x)$ and $min(x, y) = y \dot{-} (y \dot{-} x)$.

When we consider *predicates*, we follow the usual convention in computer programming and identify the number 0 with *false* and all positive numbers with *true*. Note that the μ operator was defined with this convention in mind.

It is also conventional that 1 is the "primary" true value, to be returned by "intended" booleans. For example, the $<$ predicate (actually, function) is defined by $LT(x, y) = sgn(y \dot{-} x)$. The propositional connectives are defined by: $or(x, y) = sgn(x + y)$ and $not(x) = 1 \dot{-} x$; other ones are combinations of these.

Using the μ operator, it is easy to write names for non-total functions. For example, let $LOOP$ be the 0-ary name $M^0(Z^1)$, and let $SLOOP$ be is its "successor", $C_1^0(S^1, M^0(Z^1))$. Then $\Phi_{LOOP} = \Phi_{SLOOP} = \perp$.

§**2.5. Computations.** Informally, if $\Phi_\lambda^{\mathcal{F}}(\vec{x}) = y \in \omega$, then there is a finite computation which establishes this fact. The computation is the search tree you would construct in an attempt to compute the value of $\Phi_\lambda^{\mathcal{F}}(\vec{x})$. If defined in the natural way, such a computation is unique. For example, say you wanted to apply the definition and compute $7 + 2$ by evaluating the name $P^2(I_0^1, C_1^3(S^1, I_2^3))$. You could organize the search by unwinding the intended meaning of P^2 and C_1^3, eventually getting down to the base names, whose values you would write down at sight. The result of the computation might look like:

$P^2(I^1_0, C^3_1(S^1, I^3_2))[7,2] = 9$
  $P^2(I^1_0, C^3_1(S^1, I^3_2))[7,1] = 8$
    $P^2(I^1_0, C^3_1(S^1, I^3_2))[7,0] = 7$
      $I^1_0[7] = 7$
    $C^3_1(S^1, I^3_2)[7,0,7] = 7$
      $I^3_2[7,0,7] = 7$
      $S^1[7] = 8$
  $C^3_1(S^1, I^3_2)[7,1,8] = 9$
    $I^3_2[7,1,8] = 8$
    $S^1[8] = 9$

We can think of this as a tree, displayed in preorder traversal.

To make this formal, call a *search node* any triple of the form $(\lambda, \vec{x}, v)$, where for some $n \in \omega$, λ is an n-ary name, $\vec{x} \in \omega^n$, and $v \in \omega$. An *$\mathcal{F}$-computation* is a finite rooted ordered tree, where each node $N = (\lambda, \vec{x}, v)$ in the tree is a search node, for which one of the following holds:

λ is a base name, N is a leaf, and $\Phi^{\mathcal{F}}_\lambda(\vec{x}) = v$.

λ is $C^n_m(\sigma, \tau_1, \ldots, \tau_m)$, and for some $y_1, \ldots, y_m$, the children of N are
$(\tau_1, \vec{x}, y_1), \ldots, (\tau_m, \vec{x}, y_m), (\sigma, (y_1, \ldots, y_m), v)$, in that order.

$\lambda = P^n(\sigma, \tau)$, $\vec{x} = (\vec{u}, 0)$, and N has one child, $(\sigma, \vec{u}, v)$.

$\lambda = P^n(\sigma, \tau)$, $\vec{x} = (\vec{u}, y+1)$, and for some z, N has two children
$(\lambda, (\vec{u}, y), z)$, and $(\tau, (\vec{u}, y, z), v)$, in that order.

$\lambda = M^n(\tau)$, and for some $y > 0$, N has $v+1$ children, in order:
$(\tau, (\vec{x}, 0), 0) \ldots, (\tau, (\vec{x}, v-1), 0), (\tau, (\vec{x}, v), y)$.

2.5.1. Lemma. Given $\mathcal{F}$, an n-ary λ, and $\vec{x} \in \omega^n$,

a. If $\Phi^{\mathcal{F}}_\lambda(\vec{x}) = y \in \omega$, there is exactly one $\mathcal{F}$-computation with root node $(\lambda, \vec{x}, y)$.
b. If there is any $\mathcal{F}$-computation with root node $(\lambda, \vec{x}, y)$, then $\Phi^{\mathcal{F}}_\lambda(\vec{x}) = y$.

§**2.6. Recursive Functionals.** If we fix λ and let $\mathcal{F}$ vary, we get recursive second order functionals. If $m, n > 0$, and $p \in (\omega^1 \longrightarrow \omega)$, we say that a map $\Gamma : (\omega^m \longrightarrow \omega) \to (\omega^n \longrightarrow \omega)$ is *recursive in parameter p* iff for some n-ary name λ, $\Gamma(f) = \Phi^{p,f}_\lambda$ for all $f \in (\omega^m \longrightarrow \omega)$. We may view $(\omega^m \longrightarrow \omega)$ as a topological space (in fact, a separable complete metric space) if we consider ω^m to be discrete and take the usual product topology. Since every computation is finite, we have:

2.6.1. Lemma. A map $\Gamma : (\omega^m \longrightarrow \omega) \to (\omega^n \longrightarrow \omega)$ is continuous iff Γ is recursive in some parameter.

These second-order recursive functionals are echoed in the proof theory of PRA; see §3.1. A version of this lemma is also true for maps on other separable complete metric spaces, but it involves some coding of the points in the space by sequences of natural numbers.

§**2.7. Gödel Numbering.** As usual, $\omega^{<\omega} = \bigcup_{n \in \omega} \omega^n$ is the set of all finite sequences of natural numbers. Let p_i be the ith prime; so $p_0 = 2$, $p_1 = 3$, etc. If $s \in \omega^n$, let $\#(s) = \prod_{i<n} p_i^{s_i+1}$, and let $lh(\#(s)) = n$. Denote the empty sequence by $\emptyset$ or $()$; $\#(\emptyset) = 1$

and $lh(1) = 0$. Let $Seq = \{\#(s) : s \in \omega^{<\omega}\}$. Let $lh(x) = 0$ for $x \notin Seq$. Let $\pi(i, \#s) = s_i$ whenever s is a sequence of length greater than i, and let $\pi(i, x) = 0$ whenever $i \geq lh(x)$.

2.7.1. Lemma. *Seq*, *lh*, and π are primitive recursive. For each n, the map $(x_0, \ldots, x_{n-1}) \mapsto \#(x_0, \ldots, x_{n-1})$ is primitive recursive.

If τ is a name, we define $\#(\tau)$ as follows:

$$\#(I^n_j) = \#(1, n, j); \quad \#(Z^n) = \#(2, n); \quad \#(S^1) = \#(3, 1); \quad \#(W^n_j) = \#(4, n, j)$$
$$\#(C^n_m(\sigma, \tau_1, \ldots, \tau_m)) = \#(5, n, m, \#(\sigma), \#(\tau_1), \ldots, \#(\tau_m))$$
$$\#(P^n(\sigma, \tau)) = \#(6, n, \#(\sigma), \#(\tau))$$
$$\#(M^n(\tau)) = \#(7, n, \#(\tau))$$

Using this numbering, we obtain a universal function. For each $n > 0$, $\vec{x} \in \omega^n$, and n-ary name λ, define $\varphi^{\mathcal{F},n}(\#\lambda, \vec{x}) = \Phi^{\mathcal{F}}_{\lambda}(\vec{x})$; set $\varphi^{\mathcal{F},n}(e, \vec{x}) = \perp$ if e is not of the form $\#\lambda$ for an n-ary name. Let $\varphi^{\mathcal{F},n}_e(\vec{x}) = \varphi^{\mathcal{F},n}(e, \vec{x})$. Then, as e varies, the $\varphi^{\mathcal{F},n}_e$ enumerate all functions in n variables partial recursive in $\mathcal{F}$. As usual, delete the superscript $\mathcal{F}$ when $\mathcal{F}$ is the zero oracle; also, delete the superscript n when it is clear from context.

Since the numbering is effective, $\varphi^n(e, \vec{x})$ is partial recursive in $e, \vec{x}$, but to prove this rigorously, we must Gödel number computations, as we do in the next section.

§2.8. The Kleene Normal Form Theorem. If $\mathcal{C}$ is a computation whose root node $N = (\lambda, \vec{x}, y)$ has r children, heading subtrees $\mathcal{C}_1, \ldots, \mathcal{C}_r$ ($r \geq 0$), then let $\#\mathcal{C} = \#(\#\lambda, \#\vec{x}, y, \#\mathcal{C}_1, \ldots, \#\mathcal{C}_r)$. Thus, if $C = \#\mathcal{C}$ is the Gödel number of a computation, then $\pi(0, C)$ is the Gödel number of the function being computed, $\pi(1, C)$ is the Gödel number of the input, and $\pi(2, C)$ is the output, or *upshot*, of the computation.

Definition. $U(C) = \pi(2, C)$. The *Kleene T predicate* (i.e., function with range $\{0, 1\}$) $T^{\mathcal{F}} \in (\omega^3 \longrightarrow \omega)$ is defined by: $T^{\mathcal{F}}(e, s, C) = 1$ iff for some $n, \lambda, \vec{x}, \mathcal{C}$, λ is an n-ary name, $e = \#\lambda$, $\vec{x} \in \omega^n$, $s = \#\vec{x}$, $\mathcal{C}$ is an $\mathcal{F}$-computation whose root node is of the form $(\lambda, \vec{x}, y)$, and $C = \#\mathcal{C}$. For each n, let $T^{\mathcal{F},n}$ be the $n + 2$ – ary predicate defined by: $T^{\mathcal{F},n}(e, \vec{x}, C) = T^{\mathcal{F}}(e, \#\vec{x}, C)$.

2.8.1. Theorem. U is primitive recursive, and $T^{\mathcal{F}}$ is primitive recursive in $\mathcal{F}$. If $f \in (\omega^n \xrightarrow{P} \omega)$ is partial recursive in $\mathcal{F}$, then for some $e \in \omega$, f is of the form:

$$f(\vec{x}) = \varphi^{\mathcal{F},n}(\vec{x}) = U(\mu C\, T^{\mathcal{F},n}(e, \vec{x}, C)) \quad .$$

This Theorem arises in every formalization of recursion theory, and is embedded in the following philosophical argument for Church's Thesis: Suppose we have *any* precisely defined algorithm for computing a function. Then, *presumably*, it is primitive recursive to decide whether or not some scribbling is a correct computation and to retrieve the value computed by that scribbling. We could then re-define $T(\vec{x}, C)$ to mean that C is the Gödel number of a successful computation by this algorithm, and re-define $U(C)$ to be the value computed by this computation. It follows that the function $f(\vec{x}) = U(\mu C\, T(\vec{x}, C))$ computed by this algorithm is partial recursive in our technical sense.

Note that Church's Thesis does not rule out the computability of a non-recursive function by, e.g., some biological (= physical) process.

The technical interest in the Theorem is that it shows that every partial recursive function is obtained by only one use of the μ operator (i.e., it is named by a *simple* name; see §2.1). It also yields a partial recursive enumeration of the partial recursive functions, since it is now clear that $\varphi^{\mathcal{F},n}$ is partial recursive in $\mathcal{F}$. By the standard Cantor diagonal argument, the *halting problem*, $K = \{e : \varphi^1_e(e) \neq \perp\}$, is undecidable.

§2.9. The Recursion Theorem. Informally, in computing, a *recursion* is a definition of $f(y)$ by some procedure which calls f itself. For example,

D1: $f(y) = 3$ if $y = 10$; $f(y) = f(y+1)$ otherwise.

The *First Recursion Theorem* is the fairly obvious statement that any such recursion always defines f as a partial recursive function – namely, what you get by trying to compute f in the natural way; in the given example, $dom(f) = \{y : y \leq 10\}$. For $y \notin dom(f)$, the natural C or Lisp program for computing $f(y)$ will not return a value. The *Second Recursion Theorem* (Kleene [12]) is the less obvious fact that one can actually define f from a Gödel number for f. For example, there is an e such that

D2: $\varphi^1_e(0) = e$; $\varphi^1_e(y+1) = (\varphi^1_e(y))^2$.

In general, such a definition will again define a partial function, although in this particular example, φ^1_e is total, since we easily prove $y \in dom(\varphi^1_e)$ by induction on y.

2.9.1. Theorem. Given a partial recursive $g \in (\omega^{n+1} \xrightarrow{P} \omega)$, there is an $e \in \omega$ such that for all $\vec{y} \in \omega^n$, $\varphi^n_e(\vec{y}) = g(e, \vec{y})$.

Proof. Let SUB be a primitive recursive function such that for each $a, x, \vec{y}$, we have $\varphi^n_{SUB(a,x)}(\vec{y}) = \varphi^{n+1}_a(x, \vec{y})$. Then, fix any a such that $\varphi^{n+1}_a(x, \vec{y}) = g(SUB(x, x), \vec{y})$ for all $x, \vec{y}$. Let $e = SUB(a, a)$. Then for all $\vec{y}$: $\varphi^n_e(\vec{y}) = \varphi^n_{SUB(a,a)}(\vec{y}) = \varphi^{n+1}_a(a, \vec{y}) = g(SUB(a, a), \vec{y}) = g(e, \vec{y})$. ∎

The diagonal argument in this proof is exactly the same as the one used in the proof of Gödel's Second Incompleteness Theorem to produce a formula which asserts its own non-provability.

In example D2 above, $g(x, 0) = x$ and $g(x, y+1) = (g(x, y))^2$. The First Recursion Theorem can be considered a special case of the Second; for example, for D1 above, we can let $g(x, y)$ be 3 if $y = 10$ and $\varphi^1_x(y+1)$ otherwise.

Although example D1 is trivial, this method is actually useful in proving the computability of functions which are defined in a way which is recursive but not *primitive* recursive. A simple example of this is the Ackermann function, A, defined by:

$A(0, y) = 1$

$A(1, 0) = 2$

$x \geq 2 \Rightarrow A(x, 0) = x + 2$

$(x > 0 \wedge y > 0) \Rightarrow A(x, y) = A(A(x \dot{-} 1, y), y \dot{-} 1)$

The details of the definition vary in different references (this one is from [1]), but the key point is the double recursion on the last line of definition, which is not primitive recursive. The other three lines were chosen so that for small y, the map $x \mapsto A(x, y)$ is a familiar function; $A(x, 1) = 2x$ (for $x > 1$) and $A(x, 2) = 2^x$; $A(x, 3)$ is the "stack-of-twos" function,

$A(x+1,3) = 2^{A(x,3)}$. The diagonal, $x \mapsto A(x,x)$ eventually dominates every primitive recursive function. To prove that A is a total recursive function, first apply the Recursion Theorem, as with D1, to prove the existence of a partial recursive A satisfying the above definition, and then prove by double induction on (y,x) that $A(x,y) \neq \perp$.

§**2.10. Universal functions.** The map $(e,x) \mapsto \varphi_e(x)$ is a universal partial recursive function. In general, any attempt to get a universal function for a class of *total* functions leads out of the class by Cantor's diagonal argument. Specifically, if f is a total function of two variables, and we let $f_e(x) = f(e,x)$, then the function $e \mapsto f(e,e)+1$ is not one of the f_e. One can effectively enumerate all the primitive recursive functions. For example, if we let $f(e,x)$ be $\Phi_\tau(x)$ whenever $e = \#\tau$, where τ is a pure primitive recursive name (see §2.1) of arity one, and set $f(e,x) = 0$ otherwise, then f is total recursive, and the f_e enumerate all the primitive recursive functions. But, then, f is not *primitive* recursive. Since there are only countably many recursive functions, it is easy to find a total f such that the f_e enumerate all total recursive functions, but then f will not be recursive.

Let $\mathcal{PRK}$ be the class of total functions which are primitive recursive in K (see §2.8) (equivalently, primitive recursive in some r.e. set). The following (failed) attempt to get a $\mathcal{PRK}$ function which is universal for $\mathcal{PRK}$ can be motivated either by the Kleene Normal Form Theorem, or by the desire to adjoin to the recursive functions just enough functions to satisfy equations $(2),(2')$ of the Introduction for each primitive recursive ϕ. It is an abstract version of what is implemented on Nqthm.

We concentrate on the *simple* names (see §2.1), and evaluate them by re-interpreting the μ operator to return 0 rather than $\perp$ when it is undefined, so that the whole theory deals only with total functions. Formally, define $\Psi_\lambda \in (\omega^n \longrightarrow \omega)$ whenever λ is a simple name with arity n. The cases in the definition of Ψ_λ are exactly the same as in the definition of Φ_λ, with the exception of the case

$\lambda = M^n(\tau)$: $\Psi_\lambda(\vec{x}) = y \in \omega$ iff $\Psi_\tau(\vec{x},y) \in \omega\backslash\{0\}$ and for all $z < y$, $\Psi_\tau(\vec{x},z) = 0$.
$\Psi_\lambda(\vec{x}) = 0$ if there is no such y.

Since we are applying this scheme only for simple names λ, the τ here is a primitive recursive name, so $\Psi_\tau = \Phi_\tau$. In general for simple λ, $\Psi_\lambda(\vec{x}) = \Phi_\lambda(\vec{x})$ whenever $\Phi_\lambda(\vec{x}) \neq \perp$, so $\Psi_\lambda = \Phi_\lambda$ whenever Φ_λ is total. As λ varies over the simple names, the Φ_λ enumerate all the partial recursive functions (by the Kleene Normal Form Theorem), and the Ψ_λ enumerate $\mathcal{PRK}$.

Note that if τ is a primitive recursive name and λ is $M^n(\tau)$, then Ψ_λ is a Skolem function for Φ_τ:

$$\forall x,y\big(\Phi_\tau(\vec{x},y) \neq 0 \;\Rightarrow\; \Phi_\tau(\vec{x},\Psi_\lambda(\vec{x})) \neq 0\big)$$

This is precisely equation $(2')$ of the Introduction, where $\phi(\vec{x},y)$ says "$\Phi_\tau(\vec{x},y) \neq 0$" and $f(\vec{x})$ is $\Psi_\lambda(\vec{x})$.

Let $\hat{\mu}x\psi(x)$ be the least x such that $\psi(x)$ if $\exists x\psi(x)$, and 0 otherwise. So, the partial recursive functions are all obtained by zero or one application of the μ operator, but any expression with multiple applications of the μ operator is still partial recursive. The $\mathcal{PRK}$ functions are those functions obtained by zero or one application of the $\hat{\mu}$ operator. If we allow multiple applications of $\hat{\mu}$ (use all names, rather than just simple ones), we would obtain all arithmetical functions.

For each n-ary name λ, with $e = \#\lambda$, and n-tuple $\vec{x}$, define $\psi_e(\vec{x}) = \Psi_\lambda(\vec{x})$; set $\psi_e(\vec{x}) = 0$ if e is not of the form $\#\lambda$ for an n-ary simple name. The map $(e, x) \mapsto \psi_e(x)$ is universal for $\mathcal{PRK}$ and hence not in $\mathcal{PRK}$; it is recursive in K, but not *primitive* recursive in K; this map is the abstract analogue of the Nqthm `EVAL$`.

Following Boyer and Moore [4,5], define the function vc in $\mathcal{PRK}$. Here, vc stands for *value and computation*, and is the abstract analogue of the Nqthm `V&C$`. Let

$$vc(e, s) = \hat{\mu} C \, T(e, s, C) \quad .$$

That is, when $e = \#\lambda$ for some n-ary name λ, and $s = \#\vec{x}$ for some $\vec{x} \in \omega^n$, then $vc(e, s)$ is the Gödel number of the computation which computes the value of $\Phi_\lambda(\vec{x})$; $vc(e, s) = 0$ when there is no such computation. If we just want the value, without the computation, we can just use the upshot function, U, and define $apply(e, s) = U(vc(e, s))$. Then $apply$, like vc, is in $\mathcal{PRK}$, and is the analogue of the Nqthm `APPLY$`. Since $U(0) = 0$ under our Gödel numbering, $apply(e, \#\vec{x})$ is $\varphi_e(\vec{x})$ when $\varphi_e(\vec{x}) \neq \perp$, and 0 otherwise. Now, suppose $e = \#\lambda$ for some simple n-ary name λ. Then $apply(e, \#\vec{x}) = \varphi_e^n(\vec{x}) = \psi_e^n(\vec{x})$ whenever $\vec{x} \in dom(\varphi_e^n)$; in particular this holds whenever λ is a primitive recursive name. One should not confuse the two total functions ψ and $apply$. Since $apply \in \mathcal{PRK}$, it cannot be universal for $\mathcal{PRK}$, whereas $(e, x) \mapsto \psi_e(x)$ is universal for $\mathcal{PRK}$. To see a specific example of the difference, let $e = \#SLOOP$, where $SLOOP$ (defined in §2.4) "computes" the successor of the function which doesn't halt. Then $\psi_e^0() = 1$, while $apply(e, \#()) = 0$. An Nqthm example like this is discussed in §2.10 of [5], but is not quite so transparent there because the μ operator is not built in on the surface, but must be constructed using other primitives; see §4 and below.

Roughly, PRA axiomatizes the $\Phi_\tau = \Psi_\tau$ for τ a primitive recursive name. One can extend PRA in the natural way to axiomatize the Ψ_λ for all simple names λ, obtaining the system PRA^*. In PRA^*, one can define vc and $apply$, obtaining a function universal for the primitive recursive functions and thereby proving $CON(PRA)$. These matters are taken up more formally the next section. However, the recursion-theoretic strength of a class of functions is a good informal guide to the proof-theoretic strength of the natural axiomatization of the class. In this vein, we take up one more matter, which mirrors the fact that using vc, that one can get by in Nqthm without an explicit μ operator.

There is an apparent self-referentiality in the definition of vc. Let KTP be a primitive recursive name for the Kleene T predicate $T(e, s, C)$, so that if VC is the simple name $M^2(KTP)$, then VC names vc in the sense that vc is Ψ_{VC}. Now in $vc(e, s)$, we may have $e = \#\lambda$, where λ is VC or the name of some function constructed from VC. Of course, this self-referentiality is only apparent, since if vc calls itself in this way, it is really Φ_{VC} being called.

It turns out that if we keep the self-referentiality and delete the μ operator, we will still get the full strength of vc. To see this, use just the primitive recursive names. Our intent is that the wild card W_0^2 will denote vc, and the other wild cards denote the identically zero function. To do this formally, define a *Vcomputation* as follows. A Vcomputation is still a finite rooted ordered tree, where each node $N = (\lambda, \vec{x}, v)$ in the tree is a search node, but now we demand that λ be only a primitive recursive name. The requirements on N are then exactly as in §2.5, except in the case where λ is of the form W_j^n. Then,

unless $n = 2$ and $j = 0$, we require that N be a leaf and $v = 0$. If $n = 2$ and $j = 0$, we have $N = (W_0^2, (e, s), v)$. We then require that for some $m, \vec{y}, \mathcal{C}, \tau$: $e = \#\tau$, τ is an m-ary primitive recursive name, $\vec{y}$ is an m-tuple of numbers, $s = \#\vec{y}$, $\mathcal{C}$ is a Vcomputation, and $v = \#\mathcal{C}$. Note that this definition is not circular. In fact, if we define the predicate $\widetilde{T}(e, s, C)$ to be true iff for some n, $e = \#\lambda$ for some n-ary primitive recursive name, $s = \#\vec{x}$ for some $\vec{x} \in \omega^n$, and $C = \#\mathcal{C}$, where where $\mathcal{C}$ is a Vcomputation whose root node is of the form $(\lambda, \vec{x}, y)$, then $\widetilde{T}$ is primitive recursive, since it can be defined by recursion on C. For each n, define $\widetilde{T}^n$ by: $\widetilde{T}^n(e, \vec{x}, C) = \widetilde{T}(e, \#\vec{x}, C)$. Let $\widetilde{\varphi}_e^n(\vec{x}) = U(\mu C\, \widetilde{T}^n(e, \vec{x}, C))$; so these $\widetilde{\varphi}_e^n$ are all partial recursive.

In particular, we have the partial recursive function $\mu C\, \widetilde{T}(e, s, C)$, and the corresponding (non-recursive) total function, $\widetilde{vc}(e, s) = \hat{\mu} C\, \widetilde{T}(e, s, C)$, which is an abstract version of Nqthm `V&C$`. The following theorem says that $\widetilde{vc}$ has the full non-constructivity of vc.

2.10.1. Theorem.

a. If $f \in (\omega^n \xrightarrow{P} \omega)$ is partial recursive, then for some $e \in \omega$, $f = \widetilde{\varphi}_e^n$.

b. Every function in $\mathcal{PRK}$ is primitive recursive in $\widetilde{vc}$.

Proof. To prove (a), use the fact that the $\widetilde{\phi}_e$ satisfy the Recursion Theorem (by the same proof as in §2.9), and are closed under primitive recursion. Then, in particular, every r.e. set is of the form $\{s : \widetilde{vc}(e, s) \neq 0\}$ for some e, from which (b) follows easily. ∎

§3. PRIMITIVE RECURSIVE ARITHMETIC

§3.1 PRA. As already noted in the Introduction, PRA does not use quantifiers, although the semantics and proof rules are as if all variables are universally quantified. Thus, the theorems of PRA will all be universally valid statements about primitive recursive functions, such as

$$x \neq 0 \ \wedge\ x * y = x * z \ \Rightarrow\ y = z$$

We now spell out a precise definition of PRA.

Basic Syntax: There is a constant symbol 0, and for each primitive recursive name τ of arity n, we have a function symbol, F_τ, of arity n. There is also a countably infinite set of variable symbols. Using 0, variables, and function symbols, we form terms. An *atomic formula* is an expression of the form $\alpha = \beta$, where α, β are terms. We build *formulas* from atomic formulas and propositional connectives. There are no quantifiers. We use S to abbreviate the function symbol for the successor function, F_{S^1}. We are using $\alpha, \beta, \gamma, \delta$ for *terms* of PRA, and, as before, λ, τ, σ for the *names* for primitive recursive functions. We use ϕ, ψ for formulas. Letters like x, y, z will be used for the variable symbols in the logic. $\phi(\alpha/x)$ denotes the formula obtained by replacing all occurrences of x in ϕ by α.

We call a formula ϕ *absolutely valid* iff it is universally valid in all models; informally, this means that its validity can be established by reasoning about $=$, without regard to the intended meaning of 0 or the F_τ. For example, any formula of the form $(\alpha = \beta) \Rightarrow (\phi(\alpha/x) \Leftrightarrow \phi(\beta/x))$ is absolutely valid, as is any propositional tautology. Note that absolute validity is a primitive recursively decidable property of ϕ, since it can be tested by seeing if ϕ is valid in all models of a finite size bounded by an exponential function of the size of ϕ.

Axioms: ϕ is a *logical axiom* whenever ϕ is absolutely valid. For each primitive recursive name λ, we have *definitional axioms* corresponding to the definition of Φ_λ:

$$\begin{aligned}
\lambda = S^1 &: \ S(x) \neq 0 \ ; \ S(x) = S(y) \Rightarrow x = y \\
\lambda = I^n_j &: \ F_\lambda(x_0, \ldots, x_{n-1}) = x_j \\
\lambda = Z^n &: \ F_\lambda(\vec{x}) = 0 \\
\lambda = C^n_m(\sigma, \tau_1, \ldots, \tau_m) &: \ F_\lambda(\vec{x}) = F_\sigma(F_{\tau_1}(\vec{x}) \ldots F_{\tau_m}(\vec{x})) \\
\lambda = P^n(\sigma, \tau) &: \ F_\lambda(\vec{x}, 0) = F_\sigma(\vec{x}) \ ; \ F_\lambda(\vec{x}, S(y)) = F_\tau(\vec{x}, y, F_\lambda(\vec{x}, y))
\end{aligned}$$

Proof Rules: $\vdash \phi$ means that ϕ is provable in PRA. The provable formulas are the least set of formulas containing all the axioms and closed under the following rules of inference:

1. Modus Ponens: If $\vdash \phi \Rightarrow \psi$ and $\vdash \phi$ then $\vdash \psi$.
2. Substitution: If $\vdash \phi$ then $\vdash \phi(\alpha/x)$.
3. Induction: If $\vdash \phi(0/x)$ and $\vdash (\phi \Rightarrow \phi(S(x)/x))$ then $\vdash \phi$.

Equivalently, we may say $\vdash \phi$ iff there is a *formal proof* of ϕ, where a formal proof is a finite list of formulas $\phi_0, \ldots, \phi_n$, where ϕ_n is ϕ, and each ϕ_i is either an axiom or follows from earlier formulas on the list by one of the proof rules.

Note that the F_τ, for various τ, are just distinct function symbols. PRA does not contain any mechanism for quantifying over the τ.

Most formalizations of PRA in the literature ([9]) do not take all the absolutely valid formulas as axioms, but rather use some more low-level axiomatization, from which all the absolutely valid formulas can be derived. In their "official" explanation of their basic logic ([5], Chapter 4), Boyer and Moore describe such a low-level axiomatization, but this is irrelevant to the actual implementation of Nqthm, which employs an algorithm to detect something like our absolutely valid formulas and rewrite them to $true$; in addition, it also rewrites to $true$ statements about $+$ which are valid by "linear arithmetic", such as $x = f(y) \ \Rightarrow \ 2 + (x + 3) = (1 + f(y)) + 4$ (see [5], §11.1.4).

We may introduce some abbreviations, so that the theory looks more like conventional mathematics. We have already introduced S. If $PLUS$ and $TIMES$ are the specific names described in §2.4, we let $\alpha + \beta$ abbreviate $F_{PLUS}(\alpha, \beta)$ and let $\alpha * \beta$ abbreviate $F_{TIMES}(\alpha, \beta)$. Applying the axioms for the identity and zero functions, we have the usual recursive axioms for $+$ and $*$:

$$\begin{aligned}
x + 0 = x \ ; \ x + S(y) &= S(x + y) \\
x * 0 = 0 \ ; \ x * S(y) &= x * y + x
\end{aligned}$$

For example, since $PLUS$ is really $P^2(I^1_0, C^3_1(S^1, I^3_2))$. we derive $x + S(y) = S(x + y)$ as follows. First derive:

$$x + S(y) = F_{C^3_1(S^1, I^3_2)}(x, y, x + y) \qquad (\phi_1)$$

$$F_{C^3_1(S^1, I^3_2)}(x, y, x + y) = S(F_{I^3_2}(x, y, x + y)) \qquad (\phi_2)$$

$$F_{I^3_2}(x, y, x + y) = x + y \qquad (\phi_3)$$

ϕ_1 is a definitional axiom, and ϕ_2, ϕ_3 follows from definitional axioms by substitution. We now apply modus ponens three times with the (absolutely valid) logical axiom $(\phi_1 \Rightarrow (\phi_2 \Rightarrow (\phi_3 \Rightarrow (x + S(y) = S(x + y)))))$.

Using the axioms about $+, *$, plus induction, all the basic facts about $+, *$ (such as the associative, commutative, and distributive laws) may be derived; see [9, 5]. Likewise, we may let $x < y$ abbreviate $F_\lambda(x, y) = S(0)$, where λ is some standard name for the definition of the $<$ predicate (actually, function – see §2.4). Likewise, introduce abbreviations $x \geq y$, $x > y$, $x \dot{-} y$, etc.

Call a formula *pure* iff it is built using only the F_λ for λ a pure primitive recursive name (see §2.1). Let PRA_0 be PRA restricted to pure formulas. The usual definition of PRA [9] is actually our PRA_0. However, our PRA is a conservative extension of PRA_0, since our axioms say nothing at all about the value of the wild cards. Formally,

3.1.1. Lemma. If ϕ is a pure formula and is provable in PRA, then it is provable in PRA_0.

Allowing wild cards is useful because it enables us to prove, within PRA, general statements which apply to every function. In fact, under the natural interpretation, these statements may be thought of as applying to *all* functions from ω to ω, and hence, by suitable encoding, all real numbers. Thus, for example, some theorems of real analysis, such as $\forall x \geq 0[sin(x) \leq x]$ are theorems of PRA, and hence essentially finitistic. We are really proving results about primitive recursive functionals (see §2.6).

A generalization of Lemma 3.1.1 is also useful for proving *theorem schemas*, which then may be specialized to specific functions. To formalize this idea, call a *retraction* any map ρ which assigns to each wild card W_j^n a pure primitive recursive name, $(W_j^n)\rho$. Then, in the natural way, we define $(\phi)\rho$ whenever ϕ is a formula. The following is easily proved by induction on derivations:

3.1.2. Lemma. Let $\phi, \psi_1, \ldots, \psi_n$ be formulas, and suppose one can derive ϕ in PRA by using $\psi_1, \ldots, \psi_n$ as additional axioms. Let ρ any retraction such that each of $(\psi_1)\rho, \ldots, (\psi_n)\rho$ is provable in PRA_0. Then $(\phi)\rho$ is provable in PRA_0.

As an example of Lemma 3.1.2, we may formalize the following general argument, which at first sight seems second order: Given any f any function on ω, we may let $g(n)$ be the largest $x \leq n$ such that $g(x) \leq n$; set $g(n) = 0$ if there is no such n. Assume now that f is increasing $(x < y \Rightarrow f(x) < f(y))$. Then we can prove some theorems about g; for example, g is a left inverse of f $(g(f(x)) = x)$. In particular, these theorems now apply to any specific f we choose. Common examples are obtained by letting $f(x) = 2^x$, whence $g(n) = \lfloor \log_2(n) \rfloor$ when $n > 0$, or by letting $f(x) = x^2$, whence $g(n)$ is the integer part of $\sqrt{n}$.

A version of Lemma 3.1.2 as a proof rule is implemented in Nqthm, through the commands CONSTRAIN and FUNCTIONALLY-INSTANTIATE. See [3] for a detailed discussion.

As a special case of Lemma 3.1.2, where ϕ is a pure formula:

3.1.3. Lemma. Let $\psi_1, \ldots, \psi_n$ be formulas, and suppose there is *some* retraction ρ such that each $(\psi_1)\rho, \ldots, (\psi_n)\rho$ is provable in PRA_0. Then $PRA \cup \{\psi_1, \ldots, \psi_n\}$ is a conservative extension of PRA_0.

Now, using the same formalism, we may write natural axioms about a function which are *not* satisfied by any primitive recursive function. For example (see below), one can axiomatize the Ackermann function. This extension is not conservative, since it proves $CON(PRA)$.

First, some remarks on $CON(PRA)$.

Each specific natural number k is denoted by a numeral $\ulcorner k \urcorner$, where $\ulcorner 0 \urcorner$ is the symbol 0 and $\ulcorner k+1 \urcorner$ abbreviates the term $S(\ulcorner k \urcorner)$. Thus, $\ulcorner 4 \urcorner$ abbreviates the term $S(S(S(S(0))))$.

By using Gödel numbering, it is easy to formalize the *syntax* of PRA (including the notion of "proof") within PRA. In particular, one may express $CON(PRA)$. We let PF be a name for the characteristic function of the set of all (x, y) such that y is the Gödel number of a formula and x is the Gödel number of a proof of y. If k is the Gödel number of the formula $S(0) = 0$, we can let $CON(PRA)$ be the formula $F_{PF}(\ulcorner k \urcorner, y) = 0$. Then $CON(PRA)$ is not provable in PRA; see [9] for an explanation of the Incompleteness Theorem in the PRA_0 setting. Note that $CON(PRA_0) \Leftrightarrow CON(PRA)$ is provable in PRA.

Every true *ground* pure statement (such as $\ulcorner 3 \urcorner + \ulcorner 4 \urcorner = \ulcorner 7 \urcorner$) is provable in PRA. This does not hold for non-ground statements, such as $CON(PRA)$. However, every provable statement is true, whether or not it is ground; in particular, PRA is clearly consistent. This remark can obviously be formalized in ZF set theory, but in fact can be formalized in much weaker theories. All we really need is for the theory to contain some enumerating function of the primitive recursive functions. In particular, it is easy to prove $CON(PRA)$ in HA, or even in some extension of PRA which axiomatizes such an enumerating function.

In fact, it is enough to add the Ackermann function, since that will give us an enumerating function. Let PRA' be formed from PRA by adding a definition for the Ackermann function. Formally, we shall use the first binary wild card to denote this function. So, use A to abbreviate $F_{W_0^2}$, and let PRA' be PRA plus the axioms

$A(0, y) = \ulcorner 1 \urcorner$

$A(\ulcorner 1 \urcorner, 0) = \ulcorner 2 \urcorner$

$x \geq \ulcorner 2 \urcorner \Rightarrow A(x, 0) = x + \ulcorner 2 \urcorner$

$(x > 0 \wedge y > 0) \Rightarrow A(x, y) = A(A(x \dot{-} \ulcorner 1 \urcorner, y), y \dot{-} \ulcorner 1 \urcorner)$

Now, one may define an enumerating function for the primitive recursive functions which is primitive recursive in the Ackermann function, proving:

3.1.4. Lemma. $PRA' \vdash CON(PRA)$.

Further details of the proof are given below. Now, PRA' is a rather ad hoc extension of PRA, since there is no particular reason for singling out the Ackermann function. Clearly this is a part of something more general, but there are various candidates for what that "something" is.

One possibility would be to formalize double recursions in general. These may be viewed as recursion on the ordinal ω^2, and one could even go up to recursions on ϵ_0. Many constructivists would accept these extensions as constructive (see the Introduction), but we do not dwell on these here, as we have discussed them elsewhere [13].

Another candidate is to formalize the μ operator. This also seems natural, but it is definitely not constructive. We take this up in the next section.

We remark here that *induction* on ω^2 or even ω^n is completely within PRA. That is,

3.1.5. Lemma. Within PRA or PRA', suppose $\phi(x,y)$ is a formula, and $f(x,y)$, $g(x,y)$ are function, and we can prove

$$\neg\phi(x,y) \quad \Rightarrow \quad (f(x,y),g(x,y)) <_{lex} (x,y) \ \wedge \ \neg\phi(f(x,y),g(x,y)) \quad ,$$

where $<_{lex}$ denotes lexical order on pairs. Then we can prove $\phi(x,y)$.

Proof. First, by ordinary primitive recursion, define $h(x,y)$ to be the least $y' \leq y$ such that $\neg\phi(x,y')$ if there is such a y' (and, say, 0 otherwise). Then $\neg\phi(x,y)$ implies $\neg\phi(f(x,h(x,y)),g(x,h(x,y)))$ and $f(x,h(x,y)) < x$, so that $\phi(x,y)$ can be proved by ordinary induction. ∎

Proof of Lemma 3.1.4. First, arguing within PRA, define $T^-(e,s,C)$ to be equal to 1 (i.e., $true$) unless e is a Gödel number of some n-ary pure primitive recursive name and s is the Gödel number of an n-tuple of numbers, in which case $T^-(e,s,C)$ is equal to the Kleene T predicate, $T(e,s,C)$. Informally, it is true that $\forall es \exists C T^-(e,s,C)$, but we cannot prove (or even say) this in PRA.

Now, within PRA', define a diagonal function,

$$D(e,s) = \mu C < A(s+5, e+5)\,(T^-(e,s,C)) \quad .$$

Here, the μ is the *bounded* μ operator – that is, $\mu C < b\,\phi(C)$ returns b if there is no $C < b$ such that $\phi(C)$. This operator is primitive recursive, so that we indeed can define D in PRA'. But now, we argue by double induction on e,s (see Lemma 3.1.5) and prove that indeed

$$\forall es(D(e,s) < A(s+5, e+5)) \quad .$$

We can then define $diag(e,s) = U(D(e,s))$, use this to define a truth predicate for PRA formulas, and then carry out the argument indicated above that every statement provable from PRA is true. ∎

§**3.2. Formalizing the μ operator.** For each primitive recursive function $f(x,y)$, there is a natural way to axiomatize the behavior of the function $\mu y f(x,y)$. Since this is a partial function, the logic will be simpler if we replace the $\perp$ by 0, so we are really axiomatizing $\hat{\mu} y f(x,y)$. This way, we stay within the simple logical framework of PRA, proving universal validities about natural numbers, but we create a stronger theory, which shall call PRA^*. Now, among these primitive recursive functions is the Kleene T function (predicate); applying $\hat{\mu}$ to that will give us the power to prove theorems about computations from arbitrary names for recursive functions. As a special case, one can define the Ackermann function, so that PRA^* will be an extension of PRA'.

One way to formalize $\hat{\mu}$, in the spirit of §3.1, would be to choose a wild card W_j^n for each n-ary pure primitive recursive name and then to add axioms about W_j^n. However, the choice of j would depend on some specific enumeration of these names. Since we have already embedded the μ operator in names anyway, it is equivalent, but slightly more natural, to just use the simple names, as in §2.10.

So, PRA^* has a function symbol F_λ for each simple name λ. The proof rules are the same as for PRA, with, of course, a slightly different set of terms. The logical axioms are the same as those for PRA (with our new notion of term), plus the axiom for the $\hat{\mu}$ operator; that is, whenever λ is $M^n(\tau)$ (so τ is then a primitive recursive name), we formalize the definition of Ψ_λ by taking as axioms:

$$F_\lambda(\vec{x}) > 0 \Rightarrow F_\tau(\vec{x}, y) > 0$$
$$F_\lambda(\vec{x}) > 0 \wedge y > z \Rightarrow F_\tau(\vec{x}, z) = 0$$
$$F_\lambda(\vec{x}) = 0 \wedge F_\tau(\vec{x}, 0) = 0 \Rightarrow F_\tau(\vec{x}, z) = 0$$

That is, if we say $g(x) = \hat{\mu} x f(x, y)$, then $g(x) = 0$ if there is no y such that $f(x, y) \neq 0$. Note that it is also possible for $g(x)$ to be 0 "honestly" – that is, $f(x, 0) \neq 0$. This will not cause a problem with the proof theory, since formal proofs can always branch on the two cases: whether or not $f(x, 0) = 0$.

Observe that in PRA^*, we have added Skolem functions for PRA formulas, making equation $(2')$ of the Introduction into a theorem. More precisely, call a formula ϕ in the language of PRA^* *primitive recursive* iff ϕ is built using only the F_λ for λ a primitive recursive name. Then

3.2.1. Lemma. Suppose $\phi(\vec{x}, y)$ is a primitive recursive formula. Then there is a function symbol f such that

$$\phi(\vec{x}, y) \quad \Rightarrow \quad \phi(\vec{x}, f(\vec{x}))$$

is provable in PRA^*.

We remark that the definition of Ψ_λ would have made sense for all names, not just simple ones, and we could have then postulated the above axioms for all names. However, then the Ψ_λ would have enumerated all arithmetical sets, and the corresponding axiomatic theory would have been equivalent to PA; in fact, it would be what you would get by Skolemizing PA in the obvious way. Our system PRA^* is strictly weaker than PA, since $PA \vdash CON(PRA^*)$.

Next, we show how to formalize double recursions in PRA^*. We focus specifically on the Ackermann function (which will show that PRA' is contained in PRA^*), but we do not use any specific features about the Ackermann function (such as the fact that its graph is primitive recursive).

3.2.2. Lemma. There is a 2-ary simple name λ such that if we use A to abbreviate F_λ, the PRA' axioms about A (see §3.1) are provable in PRA^*.

Proof. First note that the Recursion Theorem has a non-vacuous content even within PRA. That is, within PRA, one may talk about Gödel numbers for arbitrary partial recursive functions, and define the function SUB. We may thus trace out the argument in Theorem 2.9.1 to come up with a Gödel number e for a simple name λ such that φ^2_e "should" be the Ackermann function.

Within PRA, we cannot prove (*or even say*) that $\forall xy \exists C\, T^2(e, x, y, C)$. However, in PRA^*, we can first of all state this (using our Skolem functions, by Lemma 3.2.1), and

then prove it by double induction on (y, x), which works in PRA^* as it does in PRA (see Lemma 3.1.5). ■

Besides proving functions such as A are defined, we can also prove functions are not defined. For example, if $LOOP$ and $SLOOP$ are as in §2.4, then in PRA^* one may prove $F_{LOOP}() = 0$ and $F_{SLOOP}() = \ulcorner 1 \urcorner$.

Arguing in ZF (or even in PA), PRA^* is obviously consistent, since it has a natural model with domain of discourse ω. Observe that an attempt to prove $CON(PRA^*)$ within PRA^* along the lines of Lemma 3.1.4 fails. Within PRA^*, we may formalize $vc(e, s)$ and $apply(e, s)$ (see the discussion in §2.10). These appear to be self-referential, since e may be any Gödel number of a simple name. However, this is misleading, since if the name involves the μ operator, $apply(e, \#\vec{x})$ can differ from the actual value, $\psi_e(\vec{x})$. One cannot define $\psi_e(\vec{x})$ within PRA^*.

§4. NQTHM

§4.1. Basics. Proof theories such as PRA and its variants could obviously be implemented on the computer, but the naive implementations of these would be very awkward to use. There are a number of complexities in Nqthm which make it easy to use in practice, but which also make it difficult to pin down its proof-theoretic strength. We mention a few of these here, in an attempt to connect the theory with the practice. In some cases, this is fairly straightforward, but the semantics of `V&C$` and `EVAL$` is a bit more obscure.

We begin with the more straightforward complexities.

Nqthm does not actually have a fixed pre-existing symbol for each possible primitive recursive function. Rather, it allows one to use primitive recursion to introduce new symbolic names as they are needed. As a result, it does not include all the axioms of PRA when it is booted up, as we seemed to indicate in §3. Rather, the axioms grow as the user adds primitive recursive definitions. For example, a typical numeric definition is the primitive recursive function $euclid(x, y)$, which implements the Euclidean Algorithm. It returns $gcd(x, y)$ when $0 < x < y$, and 0 otherwise

```
(defn euclid (x y)
     (if (not (and (lessp 0 x) (lessp x y)))     0
     (if (equal (remainder y x) 0)    x
     (euclid (remainder y x) x))))
```

When the user enters this expression, the axioms for this primitive recursive definition of `euclid` are added to the database of axioms. More precisely, Nqthm maintains a symbol table, on which `EUCLID` is stored with `FORMALS` equal to the list `(X Y)` and with `BODY` equal to `(IF (NOT (AND (LESSP 0 X) ... )))`.

Note that this is not a "standard" basic primitive recursion as described in §2. Nqthm accepts this definition because it can verify that in the computation of $euclid(x, y)$, all recursive calls will be of the form of $euclid(x', y')$ with $y' < y$ (actually, $y' = x < y$, since otherwise there is no recursive call). This is what is usually called a *course-of-values*

recursion, but, as is well-known [12], such recursions can be reduced to standard primitive recursions.

Furthermore, Nqthm functions operate not only on natural numbers but also on Lisp S-expressions. There is a built-in function, *count*, which measures the complexity of S-expressions. If y is a natural number, then $count(y) = y$. In particular, Nqthm views $euclid(x, y)$ as being defined for all x, y, not just numbers. Thus, in accepting the function *euclid*, it actually verified that in the computation of $euclid(x, y)$, all recursive calls would be of the form of $euclid(x', y')$ with $count(y') < count(y)$. For this particular function, this adds nothing new, since the condition `(lessp x y)` implies that `y` is a number. However, all the standard Lisp-style recursions on complexity of S-expressions are also justified in Nqthm as recursions on *count*. It is easy to see that if one Gödel numbers S-expressions in some standard way, then any recursion on *count* can be justified as a course-of-values recursion, so that such recursions are still all part of PRA.

So far, the features described give us a system equivalent to PRA. As mentioned in the Introduction, the two features which go beyond PRA are the facility for recursion on the ordinal ϵ_0 and the facility for self-referential definitions.

Ordinal recursion on Nqthm has been discussed in detail elsewhere [13], and in any case, by Gentzen [7], there was never any doubt that this feature went beyond PRA. The use or non-use of this feature is easy to document; if the source file does not contain the word `ORD-LESSP`, then ordinal recursion is never invoked.

The self-referential feature is implemented through a number of functions: `V&C$`, `V&C-APPLY$`, `APPLY$`, `EVAL$`, and `FOR`. These functions are described in [5], and in more detail in [4]. There is no discussion there of the μ operator; rather the informal motivation of `V&C$` is an attempt to formalize an interpreter for the logic. Formally, this motivation is circular, since the interpreter evaluates expressions built from `V&C$`, and of course, by Tarski's theorem, a system which can define its own truth predicate is inconsistent. However, in the formal analysis, one can ignore the motivation and work from the explicit axiomatization for `V&C$` and related functions given in [5]. In the notation of §2.10, Nqthm appears to avoid an inconsistency because it formalizes the theory of something like Ψ_{VC}, which can talk about the evaluation of Φ_{VC}, not Ψ_{VC}.

Ideally, one should do two things now:

1. Show that every theorem of PRA^* can indeed be proved on Nqthm without using ordinal recursion.
2. Show that every theorem proved on Nqthm without using ordinal recursion can also be proved in PRA^*.

In fact, we shall do neither, formally, although we shall present an informal argument for both of these. The deductive mechanism in Nqthm is fairly complex, even without the use of `V&C$`. Nqthm does not actually output a proof in some standard formal logic, but rather it simply announces that a theorem has been proved; this announcement is accompanied by an English explanation, but sometimes the explanation is simply that some complex term has been rewritten to `T` (*true*). Furthermore, verifications for statements involving `V&C$` often are obtained by rewriting via internal routines such as `REWRITE-V&C-APPLY$`, which are not even documented in [5]. Thus, lacking a formal statement of the actual reduction mechanism, it seems hopeless to attempt to prove (1) or (2) formally here, or even to

prove that Nqthm is consistent. Because of its complexities, even given a formalization of Nqthm, any formal analysis would probably require the assistance of automated reasoning techniques.

Even informally, our argument involves a few tricks. Nqthm does not have any construct directly analogous to the μ operator, so we cannot make a direct translation to PRA^*. However, Nqthm has `V&C$`, which is *roughly* analogous to our vc (see §2.10), and this should be sufficient to capture all the non-constructivity in PRA^* by Theorem 2.10.1. We consider first the similarities between vc and `V&C$`, and then the differences.

In our $vc(e, s)$, e encodes a name of a function and s encodes the arguments to the function. The Nqthm `V&C$` also inputs a name and arguments. The axioms for `V&C$` spell out how an expression is to be evaluated, and `V&C$` is axiomatized to return the atom `F` if the evaluation fails to terminate. A user-defined function, such as the `euclid` above, is evaluated by evaluating the `BODY` of the expression; this guarantees that purely primitive recursive expressions get evaluated correctly.

A minor difference is in the form of the output. Our vc returns an actual computation, which encodes in particular the value returned. In Nqthm, there is no formal notion of "computation", and `V&C$` returns just a `(VALUE . COST)` pair, where `COST` is a number which measures in some way the number of steps such a computation would take. However, under any reasonable formal definition, a computation would be obtained in a primitive recursive way from the cost and the arguments.

A more important difference is in the form of the input. Since there is no μ operator, one cannot simply input a name for a partial recursive function. However, one can name partial recursive functions via self-referential definitions. As a trivial example, in Nqthm we cannot simply write a name for a 0-ary function $g()$ which never halts, as we did with $LOOP$ in §2.4; there, we said, essentially, $g() = \mu y\,(false)$. However, in Nqthm, we can formalize the Liar Paradox ("this statement is false") by defining $g() = 1 \dot{-} g()$, which guarantees that $g()$ is undefined. More generally, if ϕ is a given predicate and we wanted to define $f(x) = \mu y\,\phi(x, y)$, we could instead define f recursively by saying that $f(x) = h(x, 0)$, where $h(x, y) = 0$ if $\phi(x, y)$, and $h(x, y) = h(x, y + 1) + 1$ otherwise. As described in Kleene [12], the use of such recursive schemas is an alternate way of obtaining all the partial recursive functions, so that we can in fact in this way name every partial recursive function.

Now, Nqthm will not accept a self-referential definition, such as

```
(defn liar () (not (liar)))
```

since the recursion is not well-founded, but the `BODY` of a function can call `V&C$` or `EVAL$`, which in turn can be fed some construct using names of functions; this is not even viewed as a recursion at all. Since these names may include `V&C$`, it appears that Theorem 2.10.1 applies to show that one can define every partial recursive function in this way. More simply, however, as described in [5], there is a built-in hack with `EVAL$` which enables one to define a function whose `BODY` explicitly calls that function. This does not come out in the axioms for `EVAL$` (which appear to be a standard primitive recursion from `V&C$`), but rather in the axioms for the computation of `BODY`. Thus, one can define:

```
(defn liar () (eval$ T '(not (liar)) NIL ))
```

Now, `LIAR` is stored in Nqthm's symbol table with `BODY` equal to `(NOT (LIAR))`; and from

this one may prove in Nqthm that this function never halts. This function is discussed in [5], but for some reason is called `RUSSELL` there. Actually, the non-halting is due just to the self-referentiality, not the liar paradox. With:

```
(defn selfref () (eval$ T '(selfref) NIL ))
```

one can prove in Nqthm that this function never halts either, since by the axioms for `V&C$`, the cost of evaluating `(selfref)` must be one more than the cost of evaluating `(selfref)`. For further examples, see [5], along with §4.2, where we show explicitly how to encode the μ operator and define the Ackermann function.

Returning to items (1) (2) above: (1) holds (presumably) because one can, within Nqthm, use `V&C$` to derive the axioms for the $\hat{\mu}$ operator applied to any primitive recursive function. (2) holds (presumably) because one can justify any self-referential definition obtained with `EVAL$` and `V&C$` by the First Recursion Theorem, as we described in §2.9 and §3.2.

§4.2. A Script. We constructed a short (31KB) Nqthm script, available by email from the author, which illustrates some of the ideas in §4.1 with three items.

Item 1: A simple abstract example, corresponding to Lemma 3.1.5, that one may formalize double inductions (that is, inductions on ω^2) in PRA. Rather than use the built-in Nqthm definition of ordinal, which is a little more complex, we defined our own notion of "ordinal below ω^2" to be just an ordered pair of numbers (x, y) (representing $\omega \cdot x + y$) and defined the order, which is just lexorder:

```
(defn ordp (p) (and
    (listp p)
    (numberp (car p))
    (numberp (cdr p)) ))
(defn lexp (p1 p2) (or
    (lessp (car p1) (car p2))
    (and (equal (car p1) (car p2)) (lessp (cdr p1) (cdr p2))) ))
```

We now considered an abstract property, Q on our ω^2, and assumed it was inductive, in the sense that $\forall\beta(\neg Q(\beta) \Rightarrow \exists\alpha < \beta\, \neg Q(\alpha))$. Of course, to even say this in Nqthm, we need to postulate a function, g, which returns the α as $g(\beta)$. So, we had:

```
(dcl Q (p))
(dcl g (p))
(add-axiom Q-induction (rewrite) (implies
    (and (ordp p) (not (Q p)))
    (and (ordp (g p)) (lexp (g p) p) (not (Q (g p)))) ))
```

Then, after about 100 lines of intermediate lemmas, corresponding to the proof of our Lemma 3.1.5, we got:

```
(prove-lemma Q-is-true (rewrite) (implies (ordp p) (Q p)))
```

Of course, the point here is that we did this by using only those features of Nqthm which stay within PRA; we did not use the self-referential features of Nqthm (`EVAL$`, `V&C$`, etc.), or `ORD-LESSP` (which enables ordinal recursion and induction).

Item2: We used EVAL$ and V&C$, but not ORD-LESSP, and defined the Ackermann function (using the definition in §2.9). We feel our method is general enough to implement other such double recursions.

Of course, the goal here is not just to define a function named "Ackermann", but prove that it works – that is, that it satisfies the definition given above. So, we wound up proving:

```
(prove-lemma ackermann-works (rewrite) (equal (ackermann x y)
    (if (zerop x) 1
    (if (zerop y) (if (equal x 1) 2 (plus x 2))
    (ackermann (ackermann (sub1 x) y) (sub1 y)) ))))
```

To do this, we used three main tricks.

First trick: We found it a bit awkward to deal with the built-in V&C$, which is axiomatized to return a (value . cost) pair (or F if the computation fails to halt). In our intended application, the cost is irrelevant, so we defined a function qval (quick valuation) which reset the cost to 0:

```
(defn reset (x) (if x (cons (car x) 0) F))
(defn qval (term va) (reset (v&c$ T term va)))
```

We then proved a sequence of lemmas showing that qval satisfies the axioms corresponding to the built-in axioms about V&C$ (see [5], §4.10.2). For example, the following lemma explains how to evaluate an if statement. It is somewhat simpler for qval than for V&C$, since we do not have to compute the cost:

```
(prove-lemma qval-if (rewrite) (equal
      (qval  (list 'if test thencase elsecase)  va )
      (if (qval test va)   ; if evaluation of test halts
          (if (car (qval test va))   ; if test is true
              (qval thencase va)     ; then evaluate the thencase
              (qval elsecase va) )   ; else evaluate the elsecase
          F) )  )
```

This lemma, and a few others in our script, were greatly aided by the built-in Nqthm routines REWRITE-V&C-APPLY$ and REWRITE-CAR-V&C-APPLY$. These are not documented in [5], but have the effect of letting Nqthm see unaided that V&C$ does the "right thing" when applied to ordinary primitive recursive functions.

Second trick: As described in §4.1, we used the hack with EVAL$ to define ack, a first approximation to the Ackermann function, as follows:

```
(defn base (x y) (or (zerop x) (zerop y)))
(defn base-fn (x y) (if (zerop y) (row0 x) 1))
(defn ack (x y) (eval$ T
        '(if (base x y) (base-fn x y)
               (ack (ack (sub1 x) y) (sub1 y)  ))
        (list (cons 'x x) (cons 'y y))    ))
```

Now, this definition doesn't guarantee that ack "works". It only guarantees that ack is stored in Nqthm's symbol table with BODY equal to the quoted expression,

```
'(if (base x y) (base-fn x y)
       (ack (ack (sub1 x) y) (sub1 y)  ))
```

This in turn guarantees that `(qval '(ack x y) va)` will attempt to evaluate this `BODY`. We still have to prove something about how this body is evaluated, which requires analyzing its syntactic form. We used the functions `base` and `base-fn` to make this form as short as possible, and to isolate the base case, which is the non-problematic part of the definition. Our official definition of the function `ackermann` is:

```
(defn assn (valx valy)  (list (cons 'x valx) (cons 'y valy)))
(defn A (valx valy) (qval '(ack x y) (assn valx valy)))
(defn ackermann (valx valy) (car (A valx valy)))
```

That is, the function `A` should return a `(value . 0)` pair, and then `ackermann` is the `car` of this pair.

Third trick: Of course, the second trick could be applied to *any* self-referential definition. There are other such self-referential definitions which in fact define total functions, but for which this totality is not provable in PA (or, even in ZF). In the case at hand, the lemma `ackermann-works` requires a double induction on the pair (y, x), and we use the method of Item 1 for formalizing such inductions without the use of ordinals.

Item 3. This illustrate the use of `EVAL$` and `V&C$` to implement the non-constructive unbounded μ operator. As of this writing, the *twin prime conjecture* is still open; this is the statement that there are infinitely many numbers y such that y and $y+2$ are both primes. To keep the notation short, we defined `primep` in the obvious way, and then defined $fpp(y)$ to say that y is the first element of a twin prime pair:

```
(defn fpp (y)   (and
        (primep y)
        (primep (add1 (add1 y)))))
```

Then, we defined a function `next-twin-prime` which, for each x, returns a $y > x$ such that $fpp(y)$ if there exists such a y, and returns the atom `F` if there is no such y. We called those x for which such a y exists "*good*". For good x, we proved that `next-twin-prime` returns what it should:

```
(prove-lemma properties-of-good (rewrite) (implies
        (good x)
        (and
           (fpp (next-twin-prime x))
           (lessp x (next-twin-prime x))))  )
```

If x is not good, we proved that `next-twin-prime` returns `F` and there is no larger twin prime pair:

```
(prove-lemma properties-of-bad-1 (rewrite) (implies
        (not (good x))
        (equal (next-twin-prime x) F)))
(prove-lemma properties-of-bad-2 (rewrite) (implies
        (and (not (good x))  (lessp x y))
        (not (fpp y))) )
```

Since we did not settle the twin prime conjecture, this set of lemmas is non-constructive. The actual definition of `next-twin-prime` was:

```
(defn ntp (x) (eval$ T
        '(if (fpp (add1 x))  (add1 x)  (ntp (add1 x)))
        (list (cons 'x x)) ) )
(defn N (valx) (qval '(ntp x) (list (cons 'x valx))))
(defn next-twin-prime (x) (if (good x) (car (N x)) F))
(defn good (x) (and
        (fpp (car (N x)))
        (lessp x (car (N x)))))
```

Thus, we first used the `EVAL$` hack, as with the Ackermann function, but here, as described at the end of §4.1, to explicitly name a use of the μ operator; that is, $ntp(x) = \mu y\,(y > x \wedge fpp(y))$. Then, we used `qval` to get `next-twin-prime` from `ntp` in the same way that we got `Ackermann` from `ack`.

§5. CONCLUSION

In this paper, we have analyzed the non-constructive self-referential features in the Boyer-Moore system of "computational logic". The question now arises as to why they are there at all. Perhaps they should be deleted. Some readers may consider it to be merely a philosophical quibble whether the proof theory is constructive. However, one of the main uses of Nqthm has been to prove correctness assertions about hardware and software. It seems unlikely that a practical statement about physical reality could require non-constructive means for its proof. Now, the practical uses of the self-referential features are all constructive. One use of `EVAL$`, as described in [5], allows one to define functions which take (a name for) a function as input (as one does in Lisp), *and* to prove general theorems about these functions. However, these uses can be obtained just as well through the constructive commands `CONSTRAIN` and `FUNCTIONALLY-INSTANTIATE` (see [3], or Lemma 3.1.2 and the following discussion). Another use in verification is to implement an embedded interpreter (see, e.g., [6]). However, these uses would be obtained if `EVAL$` were restricted to apply to primitive recursive functions. Thus, Nqthm could be modified to be purely constructive without diminishing its practical usefulness.

Of course, there is a mathematical interest in non-constructive systems, and in particular in talking about unbounded searches through the natural numbers. In fact, the original motivation of Boyer and Moore, as described in [5], was not to formalize PRA at all ([5] never mentions PRA). However, one might then argue for a more straightforward implementation of PRA^* or a stronger theory, where the unbounded μ operator is explicit; something of this nature has already been done as a modification of Nqthm by Kaufmann [10].

The full strength of Nqthm has not really been clear even to the community of users of the system, and the script described in §4.2 managed to hack Nqthm to produce results beyond what was commonly expected to be possible. Although hacking can be fun, users will tend to have more confidence in a system if its semantics is clearly visible on the surface.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] M. L. Beeson, *Foundations of Constructive Mathematics*, Springer-Verlag, 1985

[3] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore, Functional Instantiation in First Order Logic, in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, ed., Academic Press, 1991, pp. 7 – 26.

[4] R. S. Boyer and J S. Moore, The Addition of Bounded Quantification and Partial Functions to a Computational Logic and its Theorem Prover *J. Automated Reasoning*, 4 (1988) 117 – 172.

[5] R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press, 1988.

[6] B. C. Brock and W. A. Hunt, An Overview of the Formal Specification and Verification of the FM9001 Microprocessor, *preprint*, currently available on WWW at `http://www.cli.com/hardware/fm9001.html`.

[7] G. Gentzen, Die Widerspruchsfreiheit der reinen Zahlentheorie, *Mathematische Annalen* 112 (493 – 565) 1936.

[8] K. Gödel, Zur intuitionistischen Arithmetik und Zahlentheorie, *Ergebnisse eines Mathematischen Kolloquiums* 4 (1933) 34 – 38, reprinted in Feferman, Dawson, Kleene, Moore, Solovay, and van Heijenoort, *Kurt Gödel Collected Works, Volume 1*, Oxford University Press, 1986.

[9] R. L. Goodstein, *Recursive Number Theory*, North-Holland 1964.

[10] M. Kaufmann, An Extension of the Boyer-Moore Theorem Prover to Support First-Order Quantification, *J. Automated Reasoning*, 9 (1992) 355 – 372.

[11] J. Ketonen and R. Solovay, Rapidly Growing Ramsey Functions, *Annals of Math* 113 (1981) 267 – 314.

[12] S. C. Kleene, *Introduction to Metamathematics*, Van Nostrand, 1952.

[13] K. Kunen, A Ramsey Theorem in Boyer-Moore Logic, *J. Automated Reasoning*, to appear.

[14] J. Paris and L. Harrington, A Mathematical Incompleteness in Peano Arithmetic, in *Handbook of Mathematical Logic*, J. Barwise, ed., North-Holland, 1978, pp. 1133 – 1142.

[15] C. Parsons, On a Number-theoretic Choice Scheme and its Relation to Induction, in *Intuitionism and Proof Theory*, Kino. Myhill, and Vessley, eds., North-Holland, pp. 459-473. See also JSL 37 (1972) 466 – 482.

[16] W. Sieg, Fragments of Arithmetic, *APAL* 28 (1985) 33 - 71.

[17] A. S. Troelstra, *Constructivism in Mathematics, Volume 1*, North-Holland, 1988.