We investigate a logical framework for programming languages which treats requirements on computational resources as part of the formal program specification. Resource bounds are explicit in the syntax of all programs. In a programming language based on this approach, compliance of a program with imposed resource bounds would be assured by verifying the syntactic correctness using a compiler with a static type checking feature. The principal innovation of this paper is the introduction of systems of logical inference, called polynomially graded logics. These logics make resource bounds part of every proposition and every deduction. The sample calculus presented in this paper is a restriction of Godel's system T to polynomial time resources. We prove that the numerical functions representable in this calculus are exactly the PTIME functions.

Introduction. Our framework for this preliminary report is a resource-bounded refinement of the "propositions as types" paradigm of Curry, Howard, Girard, and Martin-Lof. This paradigm has been exploited for constructing very high level applicative programming languages by de Bruijn, Reynolds, Constable, Huet, Coquand, and others. Our principal innovation is the introduction of systems of logical inference, which we call polynomially graded logics, which make resource bounds part of every proposition and every deduction. Using the Curry-Howard isomorphism this can be equivalently expressed by saying we are introducing typed \( \lambda \)-calculi, called polynomially graded typed \( \lambda \)-calculi, which make resource bounds part of every type and every term.

Polynomial time versions of intuitionistic systems and their realizabilities have been investigated by Cook [5] for propositional logic, by Buss [1] for arithmetic with sharply bounded quantifiers, and more recently by Cook and Urquhart [6] and by J. Crossley and P.J. Scott [7].

Our graded calculi stem from analysis of the PER interpretation of typed \( \lambda \)-calculi (see Mitchell [11]). The PER interpretation is based on a realizability for intuitionistic logic and arithmetic using indices of all recursive functions. It is natural to ask for logic, after the work of Nerode—Remmel [12, 13] on PTIME algebra, whether there are realizabilities for intuitionistic logic which use only indices of PTIME functions, instead of indices of all recursive functions. For the realizability interpretation of intuitionistic logic, and using the obvious interpretation of the question in PER's, this is essentially answered in the negative by Freyd—Scedrov [8] (see Carboni et al. [2]). Freyd—Scedrov showed that if the usual properties of the PER interpretation for intuitionistic logic are to hold, and the indices used are indices of recursive functions, then one of the indices used is the index of a universal Turing machine. It is not possible to get along with less than all partial recursive functions as realizers and have a realizability interpretation with the usual properties for intuitionistic logic. In the presence of the rest of the apparatus, the rules of implication introduction and elimination are essentially equivalent to the presence among the realizers of an index of a Turing machine universal over all the functions whose indices are used. Because there is no PTIME Turing machine which is universal for all PTIME functions, it does not seem to be possible to introduce a single implication with the usual implication introduction and elimination rules that would cover all PTIME functions as realizers. Some refinement in the intuitionistic treatment of implication introduction and elimination seems to be required to develop a logic with a PTIME analogue of the realizability semantics. In this paper one such refinement is discussed. For simplicity of exposition partial equivalence relations are not dealt with here, the refinement we give is limited to the HRO...
interpretation (Troelstra [14]). Other possible refinements worthy of study are outlined briefly in the concluding remarks at the end of the paper.

Fix a Turing model of computation (such as multi-tape machines). Then there exists a sequence of polynomials $p_k$ with increasing degrees (e.g. $2^k$) such that for any $k$, all functions computable within a time bound given by $p_k$ are computable by a single Turing machine within time bound given by $p_{k+1}$ from a code of a program for the function and its input.

With this in mind, say that a Turing machine $T$ is $k$-TIME if $T(x)$ is computable in $\leq p_k(\max\{|x|, 2\})$ steps. It can be shown that there is a $(k+1)$-TIME universal Turing machine for all $k$-TIME computable functions (see below). We introduce one graded implication $\Rightarrow_k$ for each $k$-TIME universal machine, so that $A \Rightarrow_k B$ is realized by the codes of $k$-TIME computable functions with value in $B$ for any argument in $A$. This restores the implication introduction and elimination rules "locally", for one statement at a time, using the appropriate graded implication $\Rightarrow_k$. We note that modus ponens for the $k$th implication is computed by a universal machine for the $k$-TIME functions, which is itself only $(k+1)$-time. Care is needed in the setting up of the system, for the composition of a $k$-TIME function and an $n$-TIME function may only be a $(k+n)$-TIME function. (It is $k+n$ rather than $k\cdot n$ because the degree of the polynomial $p_k$ is $2^k$, not $i$).

Thus the existence of this hierarchy of universal PTIME Turing machines leads to graded implication and implication introduction. This suggests that we must grade all introduction and elimination rules, that is, all deduction rules. Since deductions are merely a mirror of introduction and elimination rules, this suggests that we must grade all deductions as well. The exact grades attached to the deduction rules of our calculus are the result of thinking through how PTIME Turing machines for hypotheses lead to PTIME Turing machines for conclusions. That is, the exact grades result from examining exactly how we computationally carry out logical deductions. For pedagogical reasons, that is, in order to make the syntactic deduction rules of the calculus understandable, we follow the statement of each syntactical deduction rule of polynomial time logic by the corresponding Turing machine intended as a semantic interpretation of that rule. We do this because the resource bounds which are incorporated in the grades of the deduction rules are not understandable unless one knows the associated Turing machine interpretation.

Those who prefer an uninterrupted inductive definition of syntactical deduction should ignore the interpolated machine interpretations.

For clarity, we pinpoint in advance differences between the polynomially graded logic introduced in this paper and the conventional typed $\lambda$-calculi (Mitchell [11]).

- Deductions are expressed as "typing judgements" in the style of Martin-Löf and Girard intuitionistic logics.
- All typing judgements are graded using a graded refinement of the usual notation for typing judgements.
- The usual typed $\lambda$-abstraction $\lambda x:A.t$ is replaced by a sequence of graded typed lambda abstractions $\lambda^k x:A.t$.
- The usual application $(xy)$ is replaced by a sequence of graded applications $(x^k y)$.
- The grade of an abstraction term $\lambda^k x:A.t$ does not depend on the grade $k$ of $t$, but does depend on the size of (a code for) $t$.
- A version of bounded recursion equations as a term formation rule (see Rule 9 below) is given, and is also given as a corresponding term reduction rule. Of course, both will be graded as well. This is in order to express one of the inductive definitions of PTIME functions by means of bounded recursion as described in Wagner and Wechsung [15], ch. 4. This is a way of representing in our resource bounded system part of the primitive recursion scheme present in Gödel's system T of functionals of finite type (see Troelstra [14]).

**Notation for typing judgements.** Types (that is, propositional formulas) are defined inductively as follows:

i) We assume a collection of basic types, including $\mathbb{N}$.
ii) If $A$ and $B$ are types and $k$ is a natural number, then $A\times B$, $A\rightarrow B$, and $A\Rightarrow_k B$ are types.

We want an inductive definition of the notion of a graded
Deduction \( \vdash t \) of a propositional formula \( A \), that is, a definition of:

"\( t \) is a deduction of \( A \) from \( \Gamma \) with grade \( k \) and size \( p \)".

It is equivalent, except for notation, to give an inductive definition of the notion of a graded typing judgement that term \( t \) is of type \( A \):

"term \( t \) is of type \( A \) with grade \( k \) and size \( p \) in context \( \Gamma \)".

We write \( \Gamma \vdash_{k,p} t : A \), where \( k \) and \( p \) are natural numbers. We shall also use fixed numerical constants \( b_1 \) and \( c_j \) whose sizes are \( \leq 200 \).

We assume a countably infinite collection of variables. A context \( \Gamma \) is a finite list of expressions, each of the form \( x : A \), where \( x \) is a variable and \( A \) is a type, and no variable occurs twice. In writing \( \Gamma, x : A \) we will assume that it is a context.

**Notation for P–TIME Turing Machines.** A convenient way to visualize a sequence of Turing machines with the properties mentioned in the introduction is to consider a setting in which an outside observer with a clock monitors the actions of a single Turing machine \( M \) which is universal for all partial computable functions (of a fixed arity). We let \( p_k(x) = 2^{2^k} \). Then the \( k \)th machine in the sequence is obtained when the observer always turns \( M \) off in \( p_k(\max\{2, |x|\}) \) steps; here \(|x|\) is the binary length of the input \( x \). Actually we start with a universal Turing machine \( M \) that operates on triples \( <e, x, k> \), where \( e \) is a code of a Turing machine \( T \), \( x \) is a string, and \( k \) is a code of an integer. \( M \) will first read \( x \) and \( k \) and then mark off \( p_k(\max\{2, |x|\}) \) squares on a special tape. Then \( M \) will read \( e \) and \( x \) and start a simulation of the operation of \( T \) on input \( x \). As \( M \) executes one simulated step of the operation of \( T \), \( M \) will move the head on the special tape one square to the right. Once \( M \) has moved the head on the special tape past the marked area of \( p_k(\max\{2, |x|\}) \) squares and hence \( M \) has simulated \( p_k(\max\{2, |x|\}) \) steps of the operation of \( T \) on input \( x \), then \( M \) automatically halts. Then our \( k \)th universal Turing machine can be obtained by writing a front end for \( M \) which, when given the code of a pair \( <e, x> \), will produce a code of the triple \( <e, x, k> \) which is then given to \( M \).

Here is the notation for our PTIME analogue of the IRO interpretation in Troelstra [14]. (A PTIME analogue of the HE0 interpretation, or of PER semantics, can be developed similarly if we wish to keep track of the partial equivalence relations needed.) We fix a 1–TIME computable pairing function and a 1–TIME left and right pair component functions. We assume that

\( 0 \) is a code of a 0–TIME algorithm for constant function \( 0 \)

\( <0, 0> = 0 \).

Each type is interpreted as denoting a set \( A \) of natural numbers containing 0. \( A \) is then called the value of \( A \).

The value of \( N \) will be the set of all natural numbers.

The value of \( A \ast B \) will be the set of pairs of codes \( <a, b> \), where \( a \in A \) and \( b \in B \).

The value of \( A \ast B \) will be the set of all \( <e, i> \) such that if \( i = 0 \), then \( e \in A \), and if \( i > 0 \), then \( e \in B \).

The value of \( A =_k B \) will be the set of codes for \( k \)–TIME Turing machines with output in \( B \) for any input in \( A \).

Contexts are interpreted as consisting of finite lists (not numerical codes of lists).

Our convention is that for Turing machines, input is read from right to left.

A derived typing judgement

\[ \Gamma \vdash_{k,p} t : A \]

will be interpreted by a PTIME Turing machine \( T \) such that its program is of length \( \leq p \) and such that for each \( n \in \Gamma \), \( T(n) \) is computed in \( \leq p_k(\max\{2, |n|\}) \) steps and \( T(n) \in A \). In fact, \( T \) will have the additional feature that it reads through \( \Gamma \) just far enough to the left so as to reach all the slots that correspond to those variables in \( \Gamma \) which are free in \( t \).

The definition of deduction (or equivalently, typing judgement) and Turing machine interpretation is a

377
definition by induction given by clauses 1 – 10 below. We note that in our description of grades and programs we will use loose bounds on the size and the run time of the programs. More precise estimates are easy to obtain, but depend on the details of the particular model of computation that one chooses. Our interest here is only in the general theory of such calculi. So our estimates of program size and run time have been chosen to be independent of additional detail. They include some extra constant factors.

The inductive definition.

**DEDUCTION RULE 1.**

For any $1 \leq h \leq d$, with $m = d \cdot b_0$,

$$x_1 : A_1, \ldots, x_d : A_d \vdash \text{true},$$

$$x_h : A_h.$$

**PTIME MACHINE FOR RULE 1.**

The program for this rule simply outputs the $h^{th}$ component of the $n$-tuple on the input tape. Thus the grade is 0 because the program is clearly linear time. The size of the program depends on the number of inputs $n$. Since we have decided to read inputs from right to left, the machine for the $h^{th}$ component has to read the input only from $x_d$ to $x_h$, including $x_h$.

**DEDUCTION RULE 2.**

$$\Gamma \vdash m, p : A \quad \Gamma \vdash n, q : B$$

$$\Gamma \vdash k, r <u, v> : A \times B,$$

where

$r = 2\max\{p, q\} + b_1,$

$k = 1 + \max\{m, n\}.$

**PTIME MACHINE FOR RULE 2.**

This program takes the integer outputs $a$ and $b$ of the programs for $u$ and $v$ respectively and produces the integer code for the ordered pair $(a, b)$. Here we assume that we are using the standard pairing function where $(a, b)$ is coded by $((a + b)^2 + 3a + b)$. The length of the program clearly is just a constant plus the length of the programs for $u$ and $v$. As for the grade, we must allow the possibility that the run times for either $u$ or $v$ uses exactly $p_1(\max\{2, |x|\})$ steps for input $x$ where $r = \max\{m, n\}$. Thus, even though producing the code for the ordered pair is very efficient, we cannot assume that this extra computation will not cause us to use more than $p_1(\{2, |x|\})$ steps on input $x$. Thus the grade is forced to be at least $1 + \max\{m, n\}$ in such circumstances no matter how efficient this extra computation may be. Of course this is a general phenomenon so that any extra computation added on after a program or programs can cause the grade to jump at least 1.

**DEDUCTION RULE 3.**

(For the left and right projections from a product.)

3a) $$\Gamma \vdash k, p : A \times B$$

$$\Gamma \vdash m, q : A,$$

where

$q = p + b_2,$

$m = k + 1.$

3b) $$\Gamma \vdash k, p : A \times B$$

$$\Gamma \vdash m, q : A,$$

where

$q = p + b_3,$

$m = k + 1.$

**DEDUCTION RULE 4.**

(For the left and right inclusions into a disjoint union.)

4a) $$\Gamma \vdash k, p : A$$

$$\Gamma \vdash m, q : A \oplus B,$$

where

$q = p + b_4,$

$m = k + 1.$

4b) $$\Gamma \vdash k, p : A$$

$$\Gamma \vdash m, q : A \oplus B,$$

where

$q = p + b_5,$

$m = k + 1.$

**PTIME MACHINES FOR RULES 3 AND 4.**

The explanation for the sizes of the program and their grades is similar to that of Rule 2.
DEDUCTION RULE 5. (For definition by cases.)

The assumptions
\[ \Gamma \vdash_{k,p} t : A \rightarrow B, \]
\[ \Gamma, x : A \vdash_{m_1} u : C, \]
\[ \Gamma, y : B \vdash_{n,r_2} v : C \]
yield
\[ \Gamma \vdash_{r,q} \theta[x : A, y : B, u, v, t] : C, \]
where
\[ q = 3 \max\{p, r_1, r_2\} + b_6, \]
\[ r = 1 + 2\max\{k, m, n\}. \]

PTIME MACHINE FOR RULE 5.
The program for the conclusion of this rule does the following things on a given input. First, it computes the output of \( t \), which is an integer code for an ordered pair \((e, j)\) of integers. Next, it checks if \( j = 0 \). If \( j = 0 \), it runs \( U \) on the given input together with \( e \). If \( j > 0 \), it runs \( V \) on the given input together with \( e \). Note that in computing the grade we are basically composing two functions. The number of steps required for this composition on an input of length \( x \) is roughly bounded by
\[ p_{\max\{l,n\}}(p_k(x)) = \left( \frac{2^k}{p_{\max\{l,n\}}} \right)^{x^{2k}} \]
\[ = x^{2k+\max\{l,n\}} \]
However our computation is not simply a composition of two functions since we must do some additional checking. Thus the grade becomes \( 1 + 2\cdot\max\{k, i, n\} \). Note that the program can be uniformly produced from the programs of \( t, U, \) and \( V \) so the size of the program is \( 3\cdot\max\{p, r_1, r_2\} + b_6 \) for some constant \( b_6 \).

DEDUCTION RULE 6. (Provided there are no free variables in \( t \) other than \( x \).)

\[ \Gamma, x : A \vdash_{k,p} t : B, \]
\[ \Gamma \vdash_{m,q} \lambda x : A. \ t : (A \rightarrow B), \]
where
\[ q = p + b_7, \]
m = \( \log(p) + b_8 \).

PTIME MACHINE FOR RULE 6.
Let \( T \) be the Turing machine for the assumption of Rule 6. Let \( \omega \) be the Turing table for \( T \). Recall that \( T \) has to read only the value for \( x \). The program \( \tau \) for the conclusion of Rule 6 simply writes \( \omega \). The size \( q \) of program \( \tau \) depends on the size \( p \) of \( \omega \). More precisely, \( q = p + b_7 \) for some appropriate constant \( b_7 \). Similarly the run time of program \( \tau \) also depends on \( p \) because \( \tau \) must write the output of size \( p \). More precisely, the grade will be bounded by \( \log(p) + b_8 \) for some appropriate constant \( b_8 \).

DEDUCTION RULE 7.

\[ \Gamma, x : A \vdash_{k,p} t : (A \rightarrow B), \]
\[ \Gamma \vdash_{m,q} \lambda x : A. \ t : (A \rightarrow B), \]
where
\[ q = 2\max\{p, r\} + (j + 1)b_9, \]
\[ j = i + 1, \]
\[ m = j + 1 + 2\max\{k, n\}. \]

PTIME MACHINE FOR RULE 7.
The program for the conclusion of Rule 7 takes the output of the program for \( U \), which is a code of a program \( U \) for an \( i \)-TIME computable function which maps elements of type value \( A \) to elements of type value \( B \), and the output \( V \) of the program for \( V \), which is an element of type value \( A \), and runs the program \( U \) on input \( V \). More precisely, the program for the conclusion of Rule 7 uses the programs for \( U \) and \( V \) to compute the ordered pair \(<U, V>\) and then runs the \( j\)th universal Turing machine described above on the ordered pair \(<U, V>\). Because \( U \) is a program for an \( i \)-TIME computable function, we know that the \((i+1)\)st universal Turing machine can simulate the computation of \( U \) so that we let \( j = i + 1 \). Thus the size of the program depends on the sizes \( p \) and \( q \) of the programs for \( U \) and \( V \) respectively plus the size of the program for the \( j\)th universal Turing machine. From our description of the \( j\)th universal Turing machine, it is easy to see that the size of the program for the \( j\)th universal Turing machines depends on the size of the front end which we add to the fixed
program for the universal Turing machine. The size of this front end depends on \( j \), i.e., is \((j+1)\cdot c\) for some appropriate constant \( c \). Thus the size of the program for the conclusion of Rule 7 is \( 2^{\max\{p, r\}} + (j + 1)\cdot b_g \) for some appropriate constant \( b_g \). Similarly the grade of our program depends on the grades \( k \) and \( n \) of the programs for \( u \) and \( v \) respectively plus the grade \( j \) of the \( j \)th universal Turing machine. We can then use the same analysis as we used in Rule 5 for analyzing the grade of a composition to conclude that the grade of the program for the conclusion of Rule 7 should be \( 1 + j + 2^{\max\{k, n\}} \).

We shall use several primitive term constructors. Some of them are redundant and are assumed only for convenience. Most of the functions are self explanatory and will be described in detail in the next section on semantics. In rule 8k, \( A \) is any type.

**DEDUCTION RULE 8.**

8a. \( \Gamma \vdash 0, q_0 \quad 0 : N \)

8b. \( \Gamma \vdash m, q \quad s : N \vdash_1 N \),
where
\( q = c_1 \),
\( m = \log(c_1) \).

8c. \( \Gamma \vdash m, q \quad s_1 : N \vdash_1 N \),
where
\( q = c_2 \),
\( m = \log(c_2) \).

8d. \( \Gamma \vdash m, q \quad s_2 : N \vdash_1 N \),
where
\( q = c_3 \),
\( m = \log(c_3) \).

8e. \( \Gamma \vdash m, q \quad \text{add} : N \times N \vdash_2 N \),
where
\( q = c_4 \),
\( m = \log(c_4) \).

8f. \( \Gamma \vdash m, q \quad \text{mult} : N \times N \vdash_2 N \),
where
\( q = c_5 \),
\( m = \log(c_5) \).

8g. \( \Gamma \vdash m, q \quad : N \times N \vdash_2 N \),
where
\( q = c_6 \),
\( m = \log(c_6) \).

8h. \( \Gamma \vdash m, q \quad \text{explh} : N \vdash_2 N \),
where
\( q = c_7 \),
\( m = \log(c_7) \).

8i. \( \Gamma \vdash m, q \quad s_{g : N} : N \vdash_1 N \),
where
\( q = c_8 \),
\( m = \log(c_8) \).

8j. \( \Gamma \vdash m, q \quad \text{lh} : N \vdash_1 N \),
where
\( q = c_9 \),
\( m = \log(c_9) \).

8k. \( \Gamma \vdash m, q \quad n : A \),
where
\( q = n \cdot c_{10} \),
\( m = |n| \).

**PTIME MACHINE FOR RULE 8.**
First, we note that for each rule 8x, the machine for rule 8x always outputs the same program. We describe each output in the order of presentation.

- For rule 8a, \( 0 \) is the constant function which takes the value zero.
For rule 8b, \( s \) is the successor function.
For rule 8c, \( s_1 \) adds a 0 at the end of a number in binary notation.
For rule 8d, \( s_2 \) adds a 1 at the end of a number in binary notation.
For rule 8e, \( \text{add} \) is addition.
For rule 8f, \( \text{mult} \) is multiplication.
For rule 8g, \( \div \) is limited subtraction. That is, for any natural numbers \( t \) and \( u \), \( t \div u = t - u \) if \( t \geq u \) and \( t \div u = 0 \) if \( t < u \).
For rule 8h, \( \text{exp} \) raises \( x \) to the binary length of \( x \).
For rule 8i, \( \text{sg} \) is Kleene's signum function. That is, \( \text{sg}(x) = 1 \) if \( x > 0 \) and \( \text{sg}(0) = 0 \).
For rule 8j, \( \text{lh} \) computes the binary length of an input.
For rule 8k, \( n \) represents the constant function whose value is \( n \).

**DEDUCTION RULE 9.** Let \( A \) be \( N^w \) (associate to the right). Then the assumptions:
\[
\Gamma \vdash_{k,s} g : (A \Rightarrow k', N),
\Gamma \vdash_{m,r_1} h_1 : (A \times N \times N \Rightarrow m', N),
\Gamma \vdash_{n,r_2} h_2 : (A \times N \times N \Rightarrow n', N),
\Gamma \vdash_{p,r_3} \rho : (A \times N \Rightarrow \rho' N),
\]
yield
\[
\Gamma \vdash_{j,q} t : (A \times N \Rightarrow j', N),
\]
where \( t \) is
\[
\text{LIMREC}^{[k',m',n',p']}(g, h_1, h_2, \rho)
\]
and where
\[
j = \max\{k, m, n, p\} + b_9(2 + \max\{k', m', n', p'\}) + c_5 + c_7 + c_8 + c_{12},
\]
\[
j' = 1 + 2\max\{k', m', n', p'\},
\]
\[
q = 4\max\{s, r_1, r_2, r_3\} + b_9(2 + \max\{k', m', n', p'\}) + (2^{w+1})b_0 + c_5 + c_7 + c_8 + c_{11}.
\]

**PTIME MACHINE FOR RULE 9.**
Primitive recursion is built into Gödel's system \( T \) of functionals of finite type ([Troelstra [14]]). Similarly, the intent of rule 9 is to build a version of Bounded Syntactic Recursion (BSR) as described in Wagner and Wechsung ([15], ch. 4) into our calculus. Wagner and Wechsung say that a function \( H: N^{w+1} \rightarrow N \) is obtained by BSR from \( G: N^w \rightarrow N, H_1, H_2: N^{w+2} \rightarrow N, \) and \( P : N^{w+1} \rightarrow N \) if the function \( H \) given by
\[
(1) \quad H(y_1, \ldots, y_w, e) = G(y_1, \ldots, y_w), \text{ where } e \text{ is the empty string, and}
\]
\[
(2) \quad (a) \quad H(y_1, \ldots, y_w, x0) = H_1(y_1, \ldots, y_w, x, H(y_1, \ldots, y_w, x)),
\]
\[
(b) \quad H(y_1, \ldots, y_w, x1) = H_2(y_1, \ldots, y_w, x, H(y_1, \ldots, y_w, x)),
\]
satisfies
\[
(3) \quad H(y_1, \ldots, y_w, x) \leq P(y_1, \ldots, y_w, x) \text{ for all } y_1, \ldots, y_w, x.
\]
We shall actually modify the definition of BSR slightly and say that a function \( H \) is obtained by limited recursion from \( G, H_1, H_2, \) and \( P \) as above if \( H \) is given by
\[
(i) \quad H(\tilde{y}, e) = \text{sg}[P(\tilde{y}, e) \cdot G(\tilde{y})] \cdot G(\tilde{y}),
\]
\[
(ii) \quad (a) \quad H(\tilde{y}, x0) = \text{sg}[P(\tilde{y}, x0) \cdot H_1(\tilde{y}, x, H(\tilde{y}, x))] \cdot H_1(\tilde{y}, x, H(\tilde{y}, x)),
\]
\[
(b) \quad H(\tilde{y}, x1) = \text{sg}[P(\tilde{y}, x1) \cdot H_2(\tilde{y}, x, H(\tilde{y}, x))] \cdot H_2(\tilde{y}, x, H(\tilde{y}, x)).
\]
Note that if an \( H \) defined by clauses (1) and (2) actually satisfies clause (3), then equations (i) and (ii) will give the same function. Thus limited recursion has the same power as BSR as far as functions are concerned. Our LIMREC rule (Rule 9) actually operates in the following way. We start with programs \( g, h_1, h_2, \) and \( \rho \) which, on a given input, produce instructions for computing functions \( G, H_1, H_2, \) and \( P \) respectively, as given by the assumptions of Rule 9. The program for the conclusion of Rule 9 is a result of a certain modular transformation on the given programs \( g, h_1, h_2, \) and \( \rho \). This resulting program takes the same given input and produces instructions for
computing the function $H$ defined by limited recursion according to clauses (i) and (ii) from the functions $G$, $H_1$, $H_2$, and $P$. Our program for the conclusion of Rule 9 actually produces the set of instructions as follows.

Basically the program calls a module for a computation according to clauses (i) and (ii), and this module is directed to use the output of the submodules $g$, $h_1$, $h_2$, and $\rho$ as its input. That is, this module has slots for the outputs of $g$, $h_1$, $h_2$, and $\rho$. Of course the submodules for $g$, $h_1$, $h_2$, and $\rho$ have slots for the variables of $\Gamma$. The module then asks the submodules to compute the set of instructions which compute $G$, $H_1$, $H_2$, and $P$. Then these sets of instructions are placed in the appropriate slots in the module; thus producing a set of instructions which can be used to compute the function $H$ according to equations (i) and (ii). Now consider the size of the program for the conclusion of Rule 9. Clearly it depends on the size of the programs for the submodules $g$, $h_1$, $h_2$, and $\rho$. Thus we include $4\max\{s, r_1, r_2, r_3\}$ in $q$. The module itself must also contain instructions to run the submodules. The set of instructions we use is the $z$th universal Turing machine, where 

$$z = (1 + \max\{k', m', n', p'\}).$$

By our analysis in Rule 7, the set of instructions for the $z$th universal Turing machine is of size $(2 + \max\{k', m', n', p'\}) \cdot b_9$. Our module also contains subroutines for computing the functions $sg(x), x \cdot y$, and $x^1y$ so that we also have to include a factor of $c_5 + c_7 + c_8$ in $q$. Similarly we need a subroutine for projections which, based on our analysis of Rule 1, contributes at most a factor $(w+1) \cdot b_9$ to $q$.

Also there is a constant factor $c_{11}$ due to the various extra computations that are required to assemble the various parts of the final program. The important point to note is that this factor is just a constant, because the operation of the module is uniform not only with respect to the inputs but also with respect to the submodules once we are given the grades $k'$, $m'$, $n'$, and $p'$. Thus the size $q$ of the program is given by

$$q = 4\max\{s, r_1, r_2, r_3\} + (2 + \max\{k', m', n', p'\}) \cdot b_9 + c_5 + c_7 + c_8 + c_{11}.$$ 

Next we consider the grade $j$ of the program. Suppose that the inputs to the program are $\gamma_1, \ldots, \gamma_d$, and that $t = |\gamma_1| + \ldots + |\gamma_d|$. First, the program has to run the submodules. The submodules run in time

$$(1)^{2\max\{k, m, n, p\}}.$$ 

Note that the set of instructions produced by the program for the conclusion of Rule 9 includes instructions for the $z$th universal Turing machine which is used to run the set of instructions produced by the submodules. This means that the final set of instructions produced will contain instructions of size $(2 + \max\{k', m', n', p'\}) \cdot b_9$ to account for the instructions for the $z$th universal Turing machine.

Similarly the set of instructions produced by the program for the conclusion of Rule 9 must also contain instructions for the projection functions, $sg(x), x \cdot y, x^1y$, and other instructions to connect all the subroutines. Thus the size of the output will be

$$4 \cdot (t)^{2\max\{k, m, n, p\}} + (2 + \max\{k', m', n', p'\}) \cdot b_9 + c_5 + c_7 + c_8 + c_{12}$$

for some appropriate constant $c_{12}$. Certainly a gross overestimate of the run time for the program is $(t)^{2j}$, where

$$j = \max\{k, m, n, p\} + (2 + \max\{k', m', n', p'\}) \cdot b_9 + c_5 + c_7 + c_8 + c_{12}.$$ 

Finally we must consider the grade $j'$ of the set of instructions produced by the program for the conclusion of Rule 9. This is basically the run time of the computation by limited recursion. Since the run times of the functions $G$, $H_1$, $H_2$, and $P$ are bounded by

$$(|y_1| + \ldots + |y_w| + |x|)^{\nu},$$

where

$$\nu = 2\max\{k', m', n', p'\},$$

the run time of the computation by limited recursion is bounded by

$$|x| \sum_{i=0}^{\nu} \gamma_i^{\nu},$$

where

$$\gamma_i = |x| - i - 1 + |y_1| + \ldots + |y_w| + |\rho(x)|_{|x|-i-1}, y_1, \ldots, y_w|.$$ 

382
and \( (x)_j \) denotes the initial segment of \( x \) of length \( j \) if \( j \geq 0 \), and \( |p(x)_{j-1} - y| \) is by convention set equal to 0. But each \( z_i \) is then obviously bounded by \( (|x| + |y|)^{v+1} \).
Thus the entire computation runs in no more than \( \sum_{i=0}^{j} (|x| + |y|)^{2+v} \leq (|x| + |y|)^{2+v+1} \) steps.
Thus the grade \( j' \) of the set of instructions produced by the program for the conclusion of Rule 9 is \( 1 + 2 \cdot \max\{k', m', n', p'\} \).

**DEDUCTION RULE 10.**

10a. If \( \Gamma \vdash k,p t : A \) and \( m \geq k \),
then \( \Gamma \vdash m,p t : A \).

10b. If \( \Gamma \vdash k,p t : A \) and \( q \geq p \),
then \( \Gamma \vdash k,q t : A \).

**PTIME MACHINE FOR RULE 10.**
Keep the same machine.

**Substitution and Reduction.** Substitution of terms may be defined by following one of the usual definitions of substitution in typed \( \lambda \)-calculus as long as careful attention is paid to calculating needed grades of terms and needed integers mentioned in the grading. We outline how these integer data are carried through. The precise definition is by induction on the tree of subterms.

If \( \Gamma, \Gamma' \vdash t : B \) and \( \Gamma, x : A, \Gamma' \vdash r t : B \),
then \( \Gamma, \Gamma' \vdash m,q t\{u/x\} : B \),
where \( m, q \) are linear in \( i, k, p, \) and \( r \).

All integer data in the leaves of the tree of subterms of \( t \) are increased and then propagated through the tree toward the root according to the grading given in the clauses 1–9 above.

The definition of reduction for our calculus is patterned after the definition of \( \beta \)-reduction in typed \( \lambda \)-calculus, with special attention paid to grades. The precise definition is again by induction along the clauses 1–10. In particular, the reduction rules for terms introduced in clauses 8a–8k reflect the usual equations for the numerical functions indicated. Here we mention two reduction cases that correspond to clauses 7 and 9. The first of these cases deals with \( \beta \)-reduction:
\[
((\lambda x:A.t)\, v) \rightarrow t(v/x),
\]
if \( j \geq k+1 \), and the terms are legal.

The other case we mention deals with LIMREC reduction. This reduction rule is intended to reflect the limited recursion equations. There are three subcases. As above, each of these subcases should be stated as a consequence of certain typing judgements in a relevant context. For example, omitting grades for brevity, the last two subcases are rewrite rules that correspond to the equations (ii) in our analysis of Rule 9. For \( i = 1, 2 \), the two subcases are:
\[
g = g \quad h_1 = h_1 \quad \rho = \rho \quad u = u \quad t = t
\]
\[
z = \text{mult}(a, b)\).
\]

where \( z \) is LIMREC\[g, h_1, h_2, \rho]\{u, s\}(t),\]
\(b_1\) is \( h_1 = u, <t', \text{LIMREC}[g', h_1', h_2', \rho']\{u', t'> > > > > >)\), and
\(a_1\) is \( s((\rho')\{u', s(t') > > > b_1)\).

We say that a numerical total function \( f \) is representable in the graded calculus if there exist natural numbers \( m, q, \) and \( k \) and a derived typing judgement \( \Gamma \vdash m,q t : N \Rightarrow N \) such that for every numeral \( n \), \( (t(k+1) n) \) reduces to the numeral \( f(n) \) by innermost reduction. (In fact, \( f \) does not depend on the reduction strategy.)

**Theorem.** Numerical functions representable in this system are exactly the polynomial time (PTIME) computable functions.

**Summary.** We have presented a graded typed \( \lambda \)-calculus incorporating a restriction to polynomial time resources of Gödel's system T of functionals of finite type. In this calculus polynomial time resource bounds are an explicit part of every type and every term. The numerical functions representable in this calculus are exactly the PTIME functions. Such graded typed \( \lambda \)-calculi with built-in resource bounds may be viewed as a logical framework for extending the static type checking paradigm.
to functional programming languages in which requirements on computational resources would be treated as part of the program specification.

Directions of future research.
- In a calculus similar to the one developed in this paper the representable numerical functions are exactly the polynomial space (PSPACE) computable functions. The graded calculus for PSPACE is obtained by modifying clauses 8 and 9 and the values of the constants $b_j$ and $c_i$ to reflect the inductive definition of PSPACE given in Wagner and Wechsung ([15], ch. 4).
- Graded calculi can also be developed to represent classes of deterministic space-time computable functions such as $D$-SPACE--TIME (Lin, Pol), or other similar classes given by closure conditions.
- For the calculus in this paper it is possible to use ungraded abstraction $\lambda$ instead of graded abstraction $\lambda^k$, because the information $k$ gives is present explicitly in the syntax of types. But we have left graded abstraction in the calculus for comparison's sake, since we will develop in a later paper a calculus with a more general graded abstraction, but with ungraded implication. In the latter calculus full substitution will be available.
- It is possible to develop a calculus with much tighter bounds on computation time if we allow polynomial grades.
- Grades dependent only on "active" arguments can be introduced and give tighter computational bounds. Such grades allow analysis of the actual dependence of functions on particular arguments, rather than relying on lists in which arguments are embedded as inputs.
- Our framework should be extended to study polynomial time graded logics corresponding to HA, Girard's system F or polymorphic $\lambda$-calculus (see e.g. Mitchell [11]), Feferman's systems, NUPRL (Constable et al. [4]), the Huet--Coquand calculus of constructions, etc.
- An application of our framework to imperative or logic programming languages should be considered as well.
- It is worthwhile to develop polynomial graded calculi corresponding to models of computation other than Turing machines, such as those corresponding to $\lambda$-calculi, rewrite systems, register machines, Gurevich's model of PASCAL, etc.
- Girard, Scedrov and Phil Scott are developing the PTIME features of Girard's linear logic in a forthcoming paper.
- It is worthwhile to develop a detailed calculus corresponding to concrete languages such as ML.
- Our framework should be used to refine the polynomial time complexity theory of intuitionistic systems and their realizabilities in general.

The first draft of this paper was written during Scedrov's and Remmel's visit to the Mathematical Sciences Institute at Cornell University in August, 1987. The authors also wish to thank Val Breazu--Tannen, Sam Buss, John Crossley, Jean--Yves Girard, and Phil Scott for many conversations.

Research supported by ARO contract DAAG29--85--C--0018 and NSF grant DMS--86--01048.
**Research supported by ARO contract DAAG29--85--C--0018 and NSF grant DMS--87--02473.
***Research supported by ARO contract DAAG29--85--C--0018, NSF grant CCR--87--05596, ONR grant N00014--88--K--0635, and a Young Faculty Award of the Natural Sciences Association of the University of Pennsylvania.

REFERENCES