1978

A Programming Language Theorem Which is Independent of Peano Arithmetic

Michael O'Donnell

Report Number:
78-299

A PROGRAMMING LANGUAGE THEOREM WHICH IS INDEPENDENT OF PEANO ARITHMETIC

TR- 299

Michael O'Donnell
Dept. of Computer Sciences
Purdue University
W. Lafayette, IN 47907

Introduction

In studying the logical foundations of Computer Science, it is important to know how powerful a formal system we need to derive the important facts about computing. In the past, all interesting concrete theorems about computing could in principle be cast as theorems about Turing machines, encoded by Gödel's techniques as statements about the integers, and proved using standard Peano Arithmetic (finite arguments plus induction). By Gödel's famous incompleteness result, there are true statements about numbers which cannot be proved in Peano Arithmetic. Hartmanis and Hopcroft [HH 76] pointed out that some such statements have the form of termination and complexity properties of programs, but the programs involved were contrived to make a point, and not intrinsically interesting.

This paper establishes that the termination of a natural and interesting class of programs is independent of (i.e., cannot be proved in) Peano Arithmetic. This termination property and the variation on Ramsey's theorem recently discovered by Paris [Pa 77] are the only known cases of inherently interesting results, outside of logic, which are expressible but not provable in Peano Arithmetic.

Programs in Algol-like languages usually fail to halt because of infinite while loops or infinite recursions. Languages which allow functions or procedures to be passed as parameters may provide subtler forms of nontermination, such as the following example of a program with a function f which takes as argument a function.

```
Define f: Function (g);
    Return g(g);
    end;
x := f(f);
```

Simple strongly typed languages such as PASCAL eliminate the above behavior by making it impossible to apply a function to itself. In PASCAL-like languages, the following theorem is proved by induction on the type structure of the program.

> Termination Theorem: Every program containing only total operations and nonrecursive function definitions and applications is total.

Programming languages, such as Alphard [WLS 76], which allow generic functions have a richer type structure which does not yield to simple induction. In Section IV, I show that the termination theorem for an idealized Alphard-like language is independent of Peano Arithmetic. The appendix shows how the idealized programs may actually be coded in Alphard. The independence result indicates that polymorphic languages like Alphard are very complex, and that Computer Scientists who wish to study such languages need to find logical tools more powerful than mathematical induction.

The independence proof of Section IV was inspired by three previous works. Dag Prawitz [Pr 65] showed the independence of a normal form theorem for second order Natural Deduction proofs. Sören Stenlund [Ste 72] documented the correspondence, noticed by Curry and Feys [CF 58] and W. Howard, between Natural Deduction and an extended typed lambda

calculus. John Reynolds [Re 74] pointed out the application of the lambda calculus to generic functions in programming languages.

The basic strategy for showing independence is to prove, using techniques which may be formalized within Peano Arithmetic, that the Termination Theorem implies the consistency of Peano Arithmetic. Then, by Gödel's theorem, if Peano Arithmetic is consistent, the Termination Theorem is independent. To prove that the Termination Theorem implies consistency, I encode proofs as programs in such a way that a contradictory proof must yield a nonterminating program. Thus, if all the appropriate programs terminate, then there is no way to prove a contradiction. Section I presents the programming language, Section II the proof system, Section III shows how to encode proofs as programs, and Section IV finishes the independence proof by showing that contradiction must cause nontermination.

I. The Programming Languages

I will present two idealized programming languages: Language 1 having essentially the type structure of PASCAL, and Language 2 allowing generic functions in a fashion similar to Alphard. I have chosen a notation for presenting programs and types similar to that used by Tennent in PASQUAL [Te 75], because it is fairly close to existing programming language syntax, but mathematically less intricate than the notation of Alphard.

Language 1

Types:

A countable set of primitive types t_0, t_1, t_2,...

For each sequence of types T_0, T_1,...,T_m, the type of

functions from $T_1X...XT_m$ to T_0 is written

function $(x_1:T_1,...,x_m:T_m)$: T_0, where $x_1,...,x_m$ are any m distinct identifiers. Renaming of these identifiers does not affect the type.

Constants: A countable set of symbols a_0, a_1, a_2,... Each constant is assigned exactly one type.

Identifiers: A countable set of variables $x_0,x_1,x_2,...$ whose types are determined by their context.

Expressions:

Let E:T mean that E is an expression of type T.

Constants and identifiers are expressions of their assigned types.

$E(E_1,...,E_m):T_0$ whenever

E:function $(x_1:T_1,...,x_m:T_m):T_0$ and

$E_1:T_1,...,E_m:T_m$.

Definitions are of the form

Define f: function $(x_1:T_1,...,x_m:T_m):T_0$;

zero or more definitions;

Return E;

End

where $E:T_0$.

Any identifier may appear in E, not just $x_1,...,x_m$.

Programs:

For simplicity a program is of the form

Begin

Zero or more definitions;

Return E;

End

where E:t_i for some primitive type t_i.

Inputs, assignments and conditional statements add no theoretical complexity to the language, so I omit them.

As usual, the type of an identifier is the type associated with it in the heading of the smallest containing definition, and defined functions may be referenced only within the smallest containing definition. Note that global variables are allowed in function definitions, and that definitions may be nested.

A nonrecursive program is one in which, whenever a defined function f is used within the definition of g, the definition of f appears before the definition of g. The semantics of Language 1 is given by the standard Algol copy rule.

The proof of the Termination Theorem for Language 1 is a double induction on the maximum number of nested parentheses in the type of any defined function and the number of functions which have maximal type.

Language 2

Specifiers include types, as well as the symbol type

and generic(x_1:t_1,...,x_m:t_m):type where t_1,...,t_m

are primitive types. T:type means T is a type.

Types are expanded to include identifiers (type variables), and

function (x_1:S_1,...,x_m:S_m):T where S_1,...,S_m are specifiers

and T is a type (the variable x_i may appear in $S_{i+1},\ldots,S_m,T$), and $G(E_1,\ldots,E_m)$ where G:generic $(x_1:t_1,\ldots,x_m:t_m)$:type and $E_1:t_1,\ldots,E_m:t_m$.

There is at least one primitive type which we'll call integer.

Each Constant and Identifier is assigned a specifier.

There must be a constant 0:integer, a function

s:function (x:integer):integer and a generic type

Equal:generic(x:integer,y:integer):type.

Finally, we need a function

Equate:function(x:integer,y:integer):Equal(x,y).

The meanings of these special objects are not important, as long as they have the proper types.

Expressions now include types and

$E(E_1,\ldots,E_m):T$ where E:function$(x_1:S_1,\ldots,x_m:S_m):T$ and $E_1:S_1,\ldots,E_m:S_m$

Definitions now include

Define p:generic$(x_1:t_1,\ldots,x_m:t_m)$:type;

T

End

In order to make sense of programs in Language 2, I restrict definitions further. In a definition of the form

Define f:function(...,t:type,...):T;

Body

End

the identifier t may not appear in the type of an identifier x in the body if x is global to this definition.

The simple inductive proof of the Termination Theorem used for Language 1 fails for Language 2 because applications of generic functions may yield objects of larger types. Functions may also be applied to themselves in a limited fashion as follows.

```
Begin
    Define Id:function(t:type,x:t):t;
        Return x;
        End;
    Return Id(function(t:type,x:t):t,Id)
End
```

The Termination Theorem for Language 2 is essentially a special case of a strong normalization Theorem of Stenlund [Ste 72, p. 164] and Prawitz [Pr 68] for a typed lambda calculus. The proof uses sophisticated impredicative second order reasoning. An alternate proof of an equivalent proof-theoretic normalization theorem called the Hauptsatz given by Löb [Lo 64] uses Gödel's set theoretic axiom of constructibility. The independence of the strong normalization theorem, in a third form relating to Natural Deduction systems, is proved by Prawitz [Pr 65, p. 72]. Since the typed lambda calculus of Stenlund and the Natural Deduction system of Prawitz include objects which cannot reasonably be encoded as programs (e.g., objects of type function(t:type):t), the independence of the Termination Theorem for Language 2 is not a direct corollary of the Prawitz independence result. But many of the ideas in the proof are variations of Prawitz's techniques.

II. A Natural Deduction Version of Number Theory.

Number Theory is usually developed within some version of the first order predicate calculus. When classical predicate logic is used, the result of the standard development is called Peano Arithmetic. Number Theory embedded in constructive predicate logic is called Heyting Arithmetic. There are simple proofs that Peano Arithmetic is consistent iff Heyting Arithmetic is consistent. In the sequel I will use Heyting Arithmetic to simplify some technical details.

The Constructive Predicate Calculus

Number Theory is usually treated as a first order theory, but in order to use a weaker set of axioms, I use a Natural Deduction version of a pure constructive second order predicate calculus with $\forall$ and $\rightarrow$. This particular calculus is actually too powerful to be acceptable to constructivists; it is chosen for technical rather than philosophical reasons. Prawitz [Pr 65, p. 67] shows how to define $\wedge$, $\vee$, $\exists$ in terms of $\forall$, $\rightarrow$. I use a different definition of negation, since Prawitz's version does not encode easily into Language 2. Equality is treated as a nonlogical symbol.

Number Theory

The usual nonlogical axioms for number theory include properties of successor, addition and multiplication, as well as, the following axioms of equality and successor.

1) $\forall_x x=x$

2) $\forall_{x,y,z} x=z \rightarrow (y=z \rightarrow x=y)$

3) $\forall_{x,y} x=y \rightarrow s(x)=s(y)$

4) $\forall_{x,y} s(x)=s(y) \rightarrow x=y$

5) $\forall_x s(x)\neq 0$

Finally, there is an induction scheme representing an infinite set of axioms. Using second order logic, induction may be given by the single axiom

6) $\forall_P P(0) \rightarrow ((\forall_x P(x) \rightarrow P(s(x)))) \rightarrow \forall_x P(x))$

In the presence of 1-6, addition and multiplication may be defined as follows.

a+b=c is $\forall_Q(\forall_{x,y,z} Q(x,0,x) \wedge (Q(x,y,z) \rightarrow Q(x,s(y),s(z)))) \rightarrow Q(a,b,c)$

a*b=c is $\forall_R(\forall_{x,y,z} R(x,0,0) \wedge (R(x,y,z) \rightarrow \exists_w(w+z=w \wedge R(x,s(y),w)))) \rightarrow R(a,b,c)$

From these definitions all the usual axiomatic properties of addition and multiplication are easily proved.

Any first order arithmetic formula using $\forall$, $\rightarrow$, $\wedge$, $\vee$, $\exists$, 0, s, +, *, = may now be written as a second order formula with $\forall$, $\rightarrow$, 0, s, = .

Negation is defined as follows.

$\neg P$ is $P \rightarrow 0=s(0)$

so 5 is really $\forall_x s(x)=0 \rightarrow 0=s(0)$. From 1-4 and 6 it is easy to prove (by a double induction) that $0=s(0) \rightarrow \forall_{x,y} x=y$. So, if M is any encoding of a first order arithmetic formula using $\forall$, $\rightarrow$, 0, s, =, then $0=s(0) \rightarrow M$ (proof by structural induction on the first order version of M). So $P \rightarrow 0=x(0)$ has the formal properties of $\neg P$ for first order number theory encoded in second order. From all the above, every theorem of first order Heyting Arithmetic may be proven in the second order predicate calculus from 1-6.

Now, Heyting Arithmetic is consistent iff there is no proof of 0=s(0). Prawitz shows [Pr 58, p. 72] that any atomic formula provable from 1-6 may be proved without using induction, from 1-5. So, first order Heyting Arithmetic is consistent iff there is no second order proof that 0=s(0) from 1-5.

III. Encoding Proofs as Programs

The key to encoding second order Natural Deduction proofs as programs is given by Stenlund [Ste 72, Ch. 5]; the basic idea goes back to Curry and Feys [CF 58]. Essentially, every logical formula is interpreted as a type, and a proof of a formula is an object of that type.

Given a formula P, let $\overline{P}$ be the associated type.

$\overline{A}$ is A where A is an atomic formula other than x=y (i.e. a propositional variable or a predicate variable applied to arguments)

$\overline{x=y}$ is Equal(x,y)

$\overline{P \to Q}$ is function(g:$\overline{P}$):$\overline{Q}$

$\overline{\forall_x P}$ is function(x:integer):$\overline{P}$

$\overline{\forall_P Q}$ is function(P:type):$\overline{Q}$ where P is a propositional variable

$\overline{\forall_P Q}$ is function(P:generic($y_1,\ldots,y_n$:integer):type):$\overline{Q}$ where P is an n-place predicate variable.

A proof which consists of a single axiom is one of the following functions. Axiom 5 is given in the correct form followed by an unCurried form, 2-4 are given only in the more readable unCurried form.

```
5.  Define A5:function(x:integer):function(a:Equal(s(x),0)):Equal(0,s(0));
        Define f:function(a:Equal(s(x),0)):Equal(0,s(0));
            Return Equate(0,s(0));
            End;
        Return f;
        End
5.  (unCurried)
    Define A5:function(x:integer,a:Equal(s(x),0)):Equal(0,s(0));
        Return Equate(0,s(0));
        End
1.  Define A1;function(x:integer):Equal(x,x);
        Return Equate(x,x);
        End
2.  Define A2:function(x,y,z:integer,a:Equal(x,z),b:Equal(y,z)):Equal(x,y);
        Return Equate(x,y);
        End
3.  Define A3:function(x,y:integer,a:Equal(x,y)):Equal(s(x),s(y));
        Return Equate(s(x),s(y));
        End
4.  Define A4;function(x,y:integer,a:Equal(s(x),s(y)):Equal(x,y);
        Return Equate(x,y);
        End
```

Now, every natural deduction proof Π of a theorem T from assumptions $A_1,\ldots,A_m$ translates into a sequence of definitions $\hat{\Pi}$ and an expression $\overline{\Pi}:\overline{T}$, with global variables of types $\overline{A}_1,\ldots,\overline{A}_m$ (and no other global variables) as follows.

(1) If T is the axiom i and Π is a one line proof, then $\overline{\Pi}$ is Ai and $\hat{\Pi}$ is the appropriate definition.

(2) If T is an assumption A_i then $\hat{\Pi}$ is empty and $\overline{\Pi}$ is a variable of type A_i.

(3) If Π consists of a proof Π_1 of $P \rightarrow Q$ and a proof Π_2 of P, ending with Q by Modus Ponens, then

$\hat{\Pi}$ is $\hat{\Pi}_1$; $\hat{\Pi}_2$

$\overline{\Pi}$ is $\overline{\Pi}_1(\overline{\Pi}_2)$

(4) If Π consists of a proof Π_1 of Q , from an assumption P, ending in $P \rightarrow Q$ by the deduction rule, then

$\hat{\Pi}$ is Define f:function(g:$\overline{P}$):$\overline{Q}$;
$\hat{\Pi}_1$;
Return $\overline{\Pi}_1$;
End

$\overline{\Pi}$ is f

(5) If Π consists of a proof Π_1 of $\forall_x P$, ending in P(E/x) by $\forall$ instantiation, then

$\hat{\Pi}$ is $\hat{\Pi}_1$

$\overline{\Pi}$ is $\overline{\Pi}_1(E)$

(The variable g in the function heading must be the same variable chosen to stand for the assumption of P in Π_1).

(6) If Π consists of a proof Π_1 of P (using no assumptions about x) ending in $\forall_x P$ by $\forall$ introduction, then

$\hat{\Pi}$ is Define f:function(x:integer):$\overline{P}$;
$\hat{\Pi}_1$;
Return $\overline{\Pi}_1$;
End

$\overline{\Pi}$ is f

(7) If Π consists of a proof Π_1 of $\forall_P Q$ ending in $Q(E(x_1,\ldots,x_n)/P(x_1,\ldots,x_n))$ by $\forall$ instantiation, and if P is an n place predicate variable, then

$\hat{\Pi}$ is $\hat{\Pi}_1$;

Define R:generic($x_1,\ldots,x_n$:integer):type;

$\overline{E}$;

End

$\overline{\Pi}$ is $\overline{\Pi}_1(R)$

The cases where Π involves instantiation of a propositional variable, or introduction of second order quantification, are essentially the same as (5) and (6). If Π is a proof of $0 = s(0)$ with no assumptions, then

Begin

$\hat{\Pi}$;

Return $\overline{\Pi}$;

End

is a nonrecursive program. Note that many incorrect pseudoproofs may also be translated into programs by using functions not corresponding to axioms 1-5. All that matters is that every correct proof may be encoded, the encoding of the bad ones doesn't hurt.

IV. Termination Theorem Implies Consistency

Notice that the only objects ever constructed in the programs of Section III were symmetric objects, i.e., objects of type Equal(x,x). A1 is the only function which constructs objects from scratch; A2 - A5 may produce asymmetric objects, but only if given asymmetric inputs.

So, a simple induction on the number of computation steps shows that the programs of Section III only return symmetric objects. A proof that 0 = s(0) would translate to a program of type Equal(0,s(0)). Since no such object could be constructed, the program would have to compute forever. Therefore, if all nonrecursive programs in Language 2 halt, 0 = s(0) is not provable and Peano and Heyting Arithmetics are consistent.

Appendix: Coding Language 2 in ALPHARD

In Alphard and other data abstraction languages, functions are not generally allowed to return functions as values. Any program constructed by the techniques of Section III from a proof of 0 = s(0) may be simulated by a program in which functions only return objects of type Equal(0,s(0)). The basic trick is to add to the argument list of each function a continuation argument which maps the original function value back to Equal(0,s(0)). So each object E:T becomes E':T' where

integer' is integer

(Equal(x,y))' is function(f:function(Cont:Equal(x,y)):Equal(0,s(0))):Equal(0,S(0))

(function(x:S):T)' is

function(x:S',Cont:function(y:T'):Equal(0,s(0))):Equal(0,s(0))

This is essentially the idea of continuations used in Scott-Strachey style semantics. An equivalent approach through the logic is to replace P by P' where

$(x=y)'$ is $\neg\neg\, x=y$

$(P \rightarrow Q)'$ is $\neg(P' \wedge \neg\, Q')$

$(\forall_x P)'$ is $\forall_x \neg\neg\, P'$

The primitives Equal, Equate, s and the axioms A1,...,A5 are coded in ALPHARD as

```
form Equal(x,y:int) is
    specs
        var left, right:int;
        func Equate(x,y:int):Equal(x,y);
        vproc &:=(var e,f:Equal):Equal;
```

```
impl
    var left, right:int;
    func Equate is
        value e:Equal(x,y) of
            e.left := x;
            e.right := y
        fo
        vproc & := is value e of
            e.left := f.left;
            e.right := f.right
        fo
end Equal;
func s(x:int):int is x+1;
func A1(x:int):Equal(x,x) is Equate(x,x);
func A2(x,y,z:int,e:Equal(x,z),f:Equal(y,z)):Equal(x,y) is
    Equate(x,y);
func A3(x,y,z:int,e:Equal(x,y)):Equal(s(x),s(y)) is
    Equate(s(x),s(y));
func A4(x,y:int,e:Equal(s(x),s(y))):Equal(x,y) is
    Equate(x,y);
func A5(x:int,e:Equal(0,s(x))):Equal(0,s(0)) is
    Equate(0,s(0));
```

generic definitions translate straightforwardly into form definitions in ALPHARD.

function definitions become func definitions, but every variable of simple (nonfunctional) type must be embedded in a trivial procedure which produces its value because simple variables are not allowed to be referenced globally in ALPHARD definitions.

Bibliography

[Cu 58] Curry, H. B. and Feys, R., Combinatory Logic, North-Holland Amsterdam (1958).

[Ha 77] Hartmanis, J., Relations Between Diagonalization, Proof Systems, and Complexity Gaps, Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 1977, pp. 223-227.

[HH 76] Hartmanis, J. and Hopcroft, J. E., Independence Results in Computer Science, SIGACT News, v. 8, No. 4 (1976), pp. 13-24.

[Lo 64] Löb, M. H., Cut Elimination in Type Theory (Abstract), Journal of Symbolic Logic, v. 29 (1964), p. 220.

[Pa 77] Paris, J. B., Some Independence Results for Peano Arithmetic. Manuscript.

[Pr 65] Prawitz, D., Natural Deduction. Almquist and Wiksell, Stockholm (1965).

[Pr 68] Prawitz, D., Some results for intuitionistic logic with second order quantification rules. Intuitionism and Proof Theory Conference, Buffalo, NY, North Holland, Amsterdam (1970), pp. 259-270.

[Re 74] Reynolds, J. C., Towards a Theory of Type Structure, Colloquium on Programming, Paris (1974).

[St 72] Stenlund, S., Combinators, Lambda Terms and Proof Theory, D. Reidel, Dordrecht, Holland (1972).

[Te 75] Tennent, R. D., PASQUAL: A Proposed Generalization of PASCAL, Technical Report 75-32, Dept. of Computing and Information Science, Queen's University.

[WLS 76] Wulf, W., London, R., and Shaw, M., An Introduction to the Construction and Verification of Alphard Programs, IEEE Transactions on Software Engineering, V. SE-2, No. 4, Dec. 1976.

[Yo 77] Young, P., Optimization Among Provably Equivalent Programs, JACM, V. 24, No. 4 (1977), pp. 693-700.

[Wu 78] Wulf, W. (ed.), An Informal Definition of Alphard, CMO-CS-78-105, Carnegie-Mellon University (1978).