Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver, B.C. Canada V6T 1Z4
e-mail: gilmore@cs.ubc.ca

An Impredicative Simple Theory of Types*

by

Paul C Gilmore

A Paper Prepared for Presentation to the Fourteenth Workshop on
Mathematical Foundations for Programming Systems, May 1998

ABSTRACT

By the theory TT is meant the higher order predicate logic with the following recursively defined types:

1. 1 is the type of individuals;
2. $[\tau_1, \ldots, \tau_n]$ is the type of the predicates with arguments of the types $\tau_1, \ldots, \tau_n$, $n \geq 0$.

The type [] defined by (2) when n=0 is the type of the truth values.

TT is a version of the simple theory of types. The theory ITT described in this paper is an impredicative version of TT. The types of ITT are the same as the types of TT, but the membership of the type 1 of individuals in ITT is an extension of the membership of the same type in TT. The extension consists of allowing any higher order term, in which only variables of type 1 have a free occurrence, to be a term of type 1. This impredicative feature of ITT is motivated by a nominalist view of universals.

A nominalist understands a predicate of a universal to be a predicate of a name of the universal. For example, a nominalist interprets "Yellow is a colour" as "Yellow is a colour-word"; the sentence is understood as a description of the use of the word 'Yellow' in English. Since computers are consumate nominalists, nominalist interpretations of computer languages should be considered. But this does require a careful distinction between the use and mention of predicate names, especially when treating abstraction and quantification. For example, in "Yellow is a colour-word" the predicate name 'Yellow' is being mentioned while the predicate name 'colour-word' is being used.

The types of TT prevent impredicative definitions; as a consequence the logic must be supplemented with non-logical axioms. The impredicative types of ITT, on the other hand, permit both wellfounded and non-wellfounded recursive predicates to be defined as abstraction terms from which all the properties of the predicates can be derived without the use of axioms. The technique is demonstrated using higher order Horn sequent definitions. "Computations" by iteration are also defined for these predicates.

The consistency proof for TT can be adapted for ITT, as can also Prawitz's semantic proof of completeness and cut-elimination.

* The paper is available at: http://www.cs.ubc.ca/spider/gilmore

1. INTRODUCTION

This paper is in part a correction to the papers [12, 13] in which a logic NaDSet and two applications of it were described. NaDSet was an attempt to generalize the logics described in [6, 7, 8], but was subsequently shown to be inconsistent [14]. Here a consistent logic ITT, that has evolved from the logics of [6,7, 8, 10], is described and the applications of NaDSet are revisited and extended. Apart from an outline of the contents of this paper, the remainder of this introduction sketches the motivation for ITT.

The syntax for the logic is defined in §2, its semantics in §3, and its proof theory in §4. The logic ITT combines features from set theory and from a lambda calculus based logic. This is illustrated in §4 with definitions in the style of set theory of the zero and successor of a theory of natural numbers, and a definition of ordered pair in the style of the lambda calculus. All of Peano's axioms are then derivable without an axiom of infinity, and all the desired properties of ordered pair are derivable using features from both set theory and the lambda calculus. It is for this reason that ITT may meet the requirements of a logic sketched in [15] that combines some features of set theory and of higher order logic. In §5 a foundation for recursions in ITT is sketched based on terms called *recursion generators*; the technique is demonstrated using higher order Horn sequent definitions. Using these definitions the results of [13] can be repeated in ITT. The results of [12] that can be rescued are sketched in §6, along with an extension to ITT in which such impredicative sets as Russell's can be consistently defined.

1.1. Nominalism, Extensionality, and Computer Languages

By the theory TT is meant the higher order predicate logic with the following types:

1. 1 is the type of individuals;
2. $[\tau_1, \ldots, \tau_n]$ is the type of the predicates with arguments of the types $\tau_1, \ldots, \tau_n$, $n \geq 0$.

The type [], introduced in (2) when n=0, is the type of the truth values.

TT is the simple theory of types as described in [20], but without functions from individuals to individuals as in that paper. The theory ITT described in this paper is an impredicative version of TT. Functions of individuals have been dropped since they introduce unnecessary complications and are more than compensated for in ITT The types of ITT are the same as the types of TT, but the membership of the type 1 of individuals in ITT is an extension of the membership of the same type in TT. The extension consists in adding to 1 any higher order term in which only variables of type 1 have a free occurrence. The motivation for this impredicative feature of ITT, as it was for the predecessors of ITT, is a nominalist view of universals.

A nominalist understands a predicate of a universal to be a predicate of a name of the universal. For example, a nominalist interprets "Yellow is a colour" as "Yellow is a colour-word"; the sentence is understood as a description of the use of the word 'Yellow' in English. The discussion in [16] of a nominalist interpretation of the Church version of TT may be of interest.

Since computers are consumate nominalists, nominalist interpretations of computer languages should be explored. But this does require a careful distinction between the use and mention of predicate names, especially when treating abstraction and quantification. For example, in "Yellow is a colour-word" the predicate name 'Yellow' is being mentioned while the predicate name 'colour-word' is being used.

The distinction between the use and mention of predicate names is also emphasized in [17]: Used occurrences of predicate variables are said to be in "extensional" positions while mentioned occurrences are said to be in "intensional" positions.

Mentioning the name of a predicate means that the name is implicitly quoted. This is the reason why higher order terms that are also of type 1 must be restricted to those in which only variables of type 1 have free occurrences. For only such terms can be given a Herbrand interpretation when quoted. For example, let C be a constant of type [1] and x a variable of type 1. Then C(x) is of type [], and also of type 1. As a type 1 term, C(x) is to be interpreted as the function with domain and range the type 1 terms in which no variable has a free occurrence: The value of the function C(x) for a term t in its domain is the term C(t) in its range. Such an interpretation doesn't work for higher order terms not satisfying the restriction. For example, if X is a variable of type [1] and c a constant of type 1, then X(c) is of type [] but not also of type 1, since X(c) cannot be given a Herbrand interpretation as a first order term.

It is necessary to distinguish between the intension and extension of predicates in computer languages. Although the intensions of predicates used in mathematical logic can be defined and the extensions determined from them, in many computer languages this is not possible. Rather the intension of a predicate is known only informally and the extension of the predicate provided by data entry. For example, the extension of an Employee predicate in a company database is maintained in this manner along with usually a Sex predicate. From these two predicates the intension of a predicate MaleEmployee can be defined, and its extension retrieved and printed. [9] By an accident of hiring, however, the two predicates Employee and MaleEmployee may have the same extension; but clearly their intensions must be distinguished. For this reason an extensionality axiom concluding the intensional identity of predicates from their extensional identity is inappropriate for a logic intended for computer applications.

1.2. Acknowlegements

I am grateful to J.Y. Girard whose letter describing a contradiction in NaDSet initiated the development of its successors. [14] Conversations with George Tsiknis, Eric Borm, and Jamie Andrews, and correspondance with Hendrik Boom were helpful in the development of NaDSyl. [10] Correspondance with Henk Barendregt was helpful during the last stage of the development of ITT. The financial support of the Natural Science and Engineering Research Council of Canada is gratefully acknowledged.

2. THE SYNTAX

Type membership for ITT is defined in §2.1, providing the basic syntax for the logic. The lambda reduction relation between terms of ITT is defined in §2.2. In §2.3 some more usual notations for predicate logics are introduced by definitions, and a second equivalent definition of formula in the style of [20] is given.

2.1. Type Membership

The logic ITT is assumed to have denumerably many constants and variables of each type. The type of a constant or variable is not displayed but must be either declared or inferred from context. For a constant or variable cx, τ[cx] denotes its type; this is expressed in the usual fashion as cx:τ[cx].

In the style of the Church version of the simple theory of types [4], special constants introduce the logical connectives and the quantifiers. The binary logical connective of joint denial, denoted by $\downarrow$, is a special constant of type [[],[]]; that is, it is predicate of two arguments of type []. The more usual logical connectives, $\neg$, $\rightarrow$, $\wedge$, $\vee$, and $\leftrightarrow$, will be defined in terms of it; these are used in examples while $\downarrow$ is used in some definitions and proofs to reduce the number of cases needing consideration. Similarly a special constant $\forall$ of type [[τ]] is introduced for each type τ; it is the universal quantifier for a type τ variable. The type of each $\forall$ is not displayed but must be inferred from context. The existential quantifier $\exists$ is defined in terms of $\forall$ in the usual way. Again the single quantifier $\forall$ will be used in definitions and proofs to reduce the number of required cases

As in the lambda calculus, the *abstractor* λx in (λx.M) *binds* the variable x in the *scope* M of the abstractor. A free occurrence of a variable in an expression is defined in the usual way.

Definition of Type Membership

1. cx:τ[cx], for each constant or variable cx; $\downarrow$:[[],[]]; and $\forall$:[[τ]] for each type τ.
2. M:[$\tau_1, \ldots, \tau_n$] => (λx.M):[τ[x], $\tau_1, \ldots, \tau_n$], n≥0.
3. M:[$\tau, \tau_1, \ldots, \tau_n$] & N:$\tau$ => (MN):[$\tau_1, \ldots, \tau_n$], n≥0.
4. M:τ => M:1, provided no variable of a type other than 1 has a free occurrence in M.

The unusual clause (4) results from the nominalist interpretation discussed in §1. By a *term* is meant a member of a type.

The *count* ct[M] of a term is defined by induction as follows:

1. ct[M]=0 if M is a constant or variable, or ↓, or ∀ , or is of type 1 by clause (4).
2. ct[MN]=ct[M]+ct[N]+1; and ct[λx.M]= ct[M]+1.

Let N:τ[x]. The substitution notation [N/x]M denotes the result of replacing each free occurrence of x in a term M by N. The notation can result in changes of bound variables within M; a change is necessary if a free occurrence of x in M is within the scope of an abstractor λy for which y has a free occurrence in N.

Theorem (Substitution)

Let N:τ[x]. Then M:τ => [N/x]M:τ.

Proof

The proof is by induction on ct[M], with the base case ct[M]=0 being the only one requiring special care. It may be assumed that x has a free occurrence in M. If ct[M]=0 then either M is x or M:1 by clause (4). The conclusion is immediate if M is x. First ignore the second case and prove the result for all types other than 1. Then the result for M:1 by clause (4) follows since τ[x] is 1 and [N/x]M:1 since N:1.

End of proof

2.2. Lambda Reductions

The relation > of immediate *lambda reduction* between terms is defined in the usual way.

1. M > M' if one of the following conditions holds
   .1. M is (λx.P)Q and M' is [Q/x]P, where Q:τ[x].
   .2. M is (λx.P) and M' is (λy.[y/x]P), where y:τ[x].
   .3. M is (λx.Px) and M' is P, where x has no free occurrence in P.
2. Let M > M'. Then
   .1. NM > NM' and MN > M'N.
   .2. (λx.M) > (λx.M'), for any variable x.

A *reduction path* is a sequence M_i, $0 \le i \le n$, of terms for which $M_i > M_{i+1}$. M_0 is the *head* of the path and n the *length*. The relation >> holds between terms M and N if there is a reduction path of length n, $n \ge 0$, for which M is the head and N is M_n. Thus >> is the reflexive and transitive closure of >.

Many proofs are available for the following theorem of the pure lambda calculus

Theorem (Church-Rosser)

If M >> N and M >> P, then there exists a Q for which N >> Q and P >> Q.

Clause (4) in the definition of type membership does not affect the proofs provided in [1] for the theorem.

Define the relation ≈ between terms M and N to hold if there is a P for which M >> P and N >> P. That ≈ is an equivalence relation on terms is a corollary to the Church-Rosser theorem.

Define c1 to be the set of *closed* terms of type 1; that is the terms in which no variable has a free occurrence. Since c1 is closed under the reduction relation >>, ≈ is an equivalence relation on c1. The equivalence classes of ≈ on c1 will be used in the definition of the semantics of ITT given in §3.

2.3. Formula Notations

A *formula* of ITT is a term of type []. Formulas are the basis for the proof theory for ITT described in §4. But first the sparse notation of the lambda calculus is extended by definitions that introduce notations more common to predicate logics. The application notation is "sugared" by the definitions:

$M(N)$ <df> MN
$M(N_1, \ldots , N_m, N)$ <df> $M(N_1, \ldots , N_m)(N)$, $m \geq 1$.

The prefix notation for ↓ is replaced by an infix notation

$[M{\downarrow}N]$ <df> $\downarrow MN$, for M, N:[].

All of the usual logical connectives can be defined from ¬ and ∨ which are defined

$\neg M$ <df> $[M{\downarrow}M]$
$[M{\vee}N]$ <df> $[[M{\downarrow}N]{\downarrow}[M{\downarrow}N]]$

The notation for universal quantification is simplified and existential quantification defined in terms of it.

$(\forall x.M)$ <df> $\forall(\lambda x.M)$
$(\exists x.M)$ <df> $\neg(\forall x.\neg M)$

Parenthesis will be dropped when there is no risk of confusion.

Another Definition of Formula

Here is a definition of formula in the style of [20]:

1. $cx(S_1, \ldots , S_n)$ is a *prime* formula and a formula if $\tau[cx]$ is $[\tau_1, \ldots , \tau_n]$ and $S_i{:}\tau_i$, $0 \leq i \leq n$.
2. $[F{\downarrow}G]$ is a formula if F and G are formulas.
3. $(\forall x.F)$ is a formula if F is a formula and x is a variable.
4. $(\lambda x.T)(S, S_1, \ldots , S_n)$ is a formula if $([S/x]T)(S_1, \ldots , S_n)$ is a formula, where $T{:}[\tau_1, \ldots , \tau_n]$, $S_i{:}\tau_i$, $0 \leq i \leq n$, and $S{:}\tau[x]$.

That an expression defined in this way is a term of type [] follows from the definition of type membership. That a term of type [] can be defined as a formula in Schütte's style follows from the fact that all terms of a typed lambda calculus have a normal form. [1]

3. SEMANTICS

Here the semantics for ITT is described. An n-ary predicate is interpeted by its extension;

that is the set of n-tuples of arguments for which the predicate is true. Models of ITT are defined in §3.1 and a semantic version of the substitution theorem proved in §3.2. Results that motivate the proof theory of ITT are proved in §3.3 and §3.4.

3.1. Domains, Assignments, and Models

A *domain* for a model of ITT is a function D defined for each type τ for which:

D.1. D([]) is the set of truth values {true, false}.
D.2. D(1) is the set of equivalence classes on c1 defined by $\approx$.
D.3. $D([\tau_1, \ldots, \tau_n])$ is a nonempty set of subsets of the Cartsian product $D(\tau_1)\times \ldots \times D(\tau_n)$, $n\geq 1$.

The *standard* domain for models of ITT is the domain for which $D([\tau_1, \ldots, \tau_n])$ is the set of all subsets of $D(\tau_1)\times \ldots \times D(\tau_n)$, $n\geq 1$.

An *assignment* to a given domain D is a function Φ for which

A.1. $\Phi(\tau[cx], cx) \in D(\tau[cx])$, for each constant or variable cx.
A.2. $\Phi([[],[]], \downarrow)$ is the singleton set $\{\langle \text{false}, \text{false}\rangle\}$ that defines the joint denial predicate.
A.3. $\Phi([[\tau]], \forall)$ is the singleton set $\{D(\tau)\}$ that defines the universal quantification predicate $\forall$ for type τ variables.

An assignment Φ^y is a *y-variant* of an assignment Φ if Φ and Φ^y are to the same domain and $\Phi^y(\tau, x)$ differs from $\Phi(\tau, x)$ at most when x is y and τ is $\tau[y]$.

The function Φ is defined for every term as follows:

A.4. $\Phi(1, M)$ is the equivalence class of which the term $[N_1/x_1] \ldots [N_m/x_m]M$ is a member, where $x_1, \ldots, x_m$ are all the variables with a free occurrence in M, and N_i is a member of $\Phi(1, x_i)$ for $1\leq i\leq m$.
A.5. Let $\Phi^x([\tau_1, \ldots, \tau_n], M)$ be defined for each x-variant Φ^x of Φ, $n\geq 0$.
$\Phi([\tau[x], \tau_1, \ldots, \tau_n], \lambda x.M)$ is defined to be the set of n+1-tuples $\langle \Phi^x(\tau[x], x), d_1, \ldots, d_n\rangle$ for which $\langle d_1, \ldots, d_n\rangle$ is in $\Phi^x([\tau_1, \ldots, \tau_n], M)$ for each x-variant Φ^x of Φ.
A.6. Let both $\Phi([\tau, \tau_1, \ldots, \tau_n], M)$ and $\Phi(\tau, N)$ be defined, $n\geq 0$.
$\Phi([\tau_1, \ldots, \tau_n], MN)$ is defined to be the set of n-tuples $\langle d_1, \ldots, d_n\rangle$ for which $\langle \Phi(\tau, N), d_1, \ldots, d_n\rangle$ is in $\Phi([\tau, \tau_1, \ldots, \tau_n], M)$.

An assignment Φ to a domain D is a *model* for ITT if $\Phi(\tau, M) \in D(\tau)$ for each type τ and term M of type τ. Models do exist since for the standard domain a model is defined by any assignment to it; this is the basis for the consistency proof of ITT as it is for TT. The basis for the completeness proof of TT without cut in [19] is the construction of models with domains that are not standard; this proof can be adapted for ITT.

3.2. Semantic Substitution

The following theorem is a semantic version of the substitution theorem of §2.1. Note that neither its statement nor its proof is dependent upon the standard domain.

Theorem

Let Φ be an assignment to some domain. Let Q:τ[y], where y has no free occurrence in Q, and let P:σ. Let Φ^y be the y-variant of Φ for which $\Phi^y(\tau[y], y)$ is $\Phi(\tau[y], Q)$. Then $\Phi^y(\sigma, P)$ is $\Phi(\sigma, [Q/y]P)$.

Proof

It may be assumed that y has a free occurrence in P. The proof is by induction on ct[P]. Let ct[P]=0 so that P is either y, or P:1 by clause (4) of the type membership definition. In the first case [Q/y]P is Q, so that by definition $\Phi^y(\tau, P)$ is $\Phi(\tau, [Q/y]P)$. The conclusion in the second case follows from (A.4).

Assume the conclusion of the lemma whenever ct[P] < ct. Let ct[P]=ct and consider the forms that P may take.

P is λx.M, where it may be assumed that x has no free occurrence in Q. Then if M:$[\tau_1, ... , \tau_n]$, σ is $[\tau[x], \tau_1, ... , \tau_n]$. Since y is assumed to have a free occurrence in P, y is not x, so that [Q/y](λx.M) is (λx.[Q/y]M). By (A.5), $\Phi^y(\sigma, \lambda x.M)$ is the set of n+1-tuples $\langle \Phi^{x,y}(\tau[x], x), d_1, ... , d_n\rangle$ for which $\langle d_1, ... , d_n\rangle$ is in $\Phi^{x,y}([\tau_1, ... , \tau_n], M)$ for each x-variant $\Phi^{x,y}$ of Φ^y. But $\Phi^{x,y}$ is a y-variant of Φ^x, and since x and y are distinct $\Phi^{x,y}(\tau[x], x)$ is $\Phi^x(\tau[x], x)$. By the induction assumption $\Phi^y([\tau_1, ... , \tau_n], M)$ is $\Phi([\tau_1, ... , \tau_n], [Q/y]M)$. Hence $\Phi^y(\sigma, \lambda x.M)$ is the set of n+1-tuples $\langle \Phi^x(\tau[x], x), d_1, ... , d_n\rangle$ for which $\langle d_1, ... , d_n\rangle$ is in $\Phi^x([\tau_1, ... , \tau_n], [Q/y]M)$ for each x-variant Φ^x of Φ. But this is the predicate $\Phi(\sigma, \lambda x.[Q/y]M)$ which is the predicate $\Phi(\sigma,[Q/y](\lambda x.M))$.

P is MN. Then M:$[\tau, \tau_1, ... , \tau_n]$ and N:τ, for some $\tau, \tau_1, ... , \tau_n$, $n \geq 0$. By (A.6), $\Phi^y([\tau_1, ... , \tau_n], MN)$ is the set of n-tuples $\langle d_1, ... , d_n\rangle$ for which $\langle \Phi^y(\tau, N), d_1, ... , d_n\rangle \in \Phi^y([\tau, \tau_1, ... , \tau_n], M)$. But by the induction assumption $\Phi^y([\tau, \tau_1, ... , \tau_n], M)$ is $\Phi([\tau, \tau_1, ... , \tau_n], [Q/y]M)$ and $\Phi^y(\tau, N)$ is $\Phi(\tau, [Q/y]N)$.

End of proof

Corollary

Let M:τ and M > M'. Then M':τ and $\Phi(\tau, M')$ is $\Phi(\tau, M)$, for any assignment Φ.

Proof

The proof is by induction on the definition of the relation > in §2.2. The only case of any difficulty is (1.1) where M is (λx.P)Q, M' is [Q/x]P, and Q:τ[x]. Let P:τ, where τ is $[\tau_1, ... , \tau_n]$. Let σ be $[\tau[x], \tau_1, ... , \tau_n]$, so that ($\lambda$x.P):$\sigma$ and (λx.P)Q:τ. By (A.5), $\Phi(\sigma, (\lambda x.P))$ is the set of n+1-tuples $\langle \Phi^x(\tau[x], x), d_1, ... , d_n\rangle$ for which $\langle d_1, ... , d_n\rangle$ is in $\Phi^x(\tau, P)$. Let $\Phi^x(\tau[x], x)$ be $\Phi(\tau[x], Q)$. Then $\Phi(\tau, (\lambda x.P)Q)$ is the set of n-tuples $\langle d_1, ... , d_n\rangle$ for which $\langle \Phi(\tau[x], Q), d_1, ... , d_n\rangle$ is in $\Phi^x(\tau, P)$; that is, in $\Phi(\tau, [Q/x]P)$.

End of proof

3.3. Semantic Inferences

Since Φ is a function, the value $\Phi(\tau, M)$ for each τ and M is unique. In particular, therefore, if F:[], then $\Phi([], F)$ has as its value exactly one of the truth values. This observation together with the following theorem provides the justification for the proof theory described in §4.

Theorem

Let F, G:[], and let Φ be an assignment to any domain. Then

1. Φ([], [F$\downarrow$G])=true => Φ([], F)=false & Φ([], G)=false.
   Φ([], [F$\downarrow$G])=false => Φ([], F)=true or Φ([], G)=true.
2. Let T:τ[x], and let y have no free occurrence in F. Then
   Φ([], $\forall$x.F)=true => Φ([], [T/x]F)=true
   Φ([], $\forall$x.F)=false => Φ^y([], [y/x]F)=false, for some y-variant Φ^y of Φ.
3. Let F > G. Then
   Φ([], F)=true => Φ([], G)=true
   Φ([], F)=false => Φ([], G)=false

Proof

(1) follows immediately from the truth table for $\downarrow$, and (3) follows from the corollary. Consider (2). Φ([], $\forall$x.F)=true => Φ^x([], F)=true for every Φ^x. Let $\Phi^x(\tau$[x], x) be $\Phi(\tau$[x], T). Hence Φ([], $\forall$x.F)=true => Φ([], [T/x]F)=true by semantic substitution. Further Φ([], $\forall$x.F)=false => Φ^x([], F)=false for some Φ^x. Let $\Phi^{y,x}$ be the y-variant of Φ^x for which $\Phi^{y,x}(\tau$[x], y) is $\Phi^x(\tau$[x], x). Hence Φ^y([],[y/x]F) is $\Phi^{y,x}$([],[y/x]F), since x has no free occurrence in [y/x]F, which is Φ^x([],[x/y][y/x]F) by semantic substitution, and therefore Φ^x([], F).

End of proof

3.4. Sequents and Counter-Examples

A sequent is an expression $\Gamma \vdash \Theta$ where Γ, the *antecedent* of the sequent, and Θ, the *succedent* of the sequent, are finite possibly empty sets of formulas. A sequent $\Gamma \vdash \Theta$ is *satisfied* by an assignment Φ, if Φ([], F) is false for some F in the antecedent or is true for some F in the succedent. A sequent is *valid* if it is satisfied by every assignment that is a model. An assignment Φ is a *counter-example* for a sequent if Φ([], F) is true for every F in the antecedent and false for every F in the succedent.

The proof theory of §4 provides a systematic search procedure for a counter-example for a given sequent $\Gamma \vdash \Theta$. Should the procedure fail to find such an assignment, and if it does fail it will fail in a finite number of steps, then $\Gamma \vdash \Theta$ can be shown to be valid. The finite number of steps resulting in a failure is recorded as a *derivation*. Thus a derivation for a sequent is constructed under the assumption that a counter-example Φ exists for the sequent. *Signed* formulas are introduced to abbreviate assertions about the truth value assigned to a formula by Φ. Thus +F is to be understood as an abbreviation for Φ([], F)=true and –F for Φ([], F)=false, for some conjectured counter-example Φ. Note that $\Gamma \vdash \Theta$ has no counter-example if $\Gamma' \vdash \Theta'$ has no counter-example, where $\Gamma' \subseteq \Gamma$, $\Theta' \subseteq \Theta$, and $\Gamma' \cup \Theta'$ is not empty.

4. PROOF THEORY

The proof theory is presented as a logic of sequents using a semantic tree form of the sequent calculus that has evolved from the semantic tableaux derivations of [3]. Semantic rules, in terms of which semantic trees are defined, are described in §4.1; these rules are motivated by the theorem of §3.3. A *derivation* of a sequent is a *closed* semantic tree *based on* the sequent, as these terms are defined in §4.2. Some derivations of sequents are provided in §4.3. These derivations illustrate the point made in §1 that ITT combines advantages of a set theory with those of a lambda calculus based predicate logic.

4.1. The Semantic Rules

There are + and – rules for the propositional connective ↓, for each quantifier ∀ , and for λ. These rules are

$$+\!\downarrow \quad \frac{+[F\downarrow G]}{-F} \qquad \frac{+[F\downarrow G]}{-G} \qquad\qquad -\!\downarrow \quad \frac{-[F\downarrow G]}{+F \qquad +G}$$

$$+\forall \quad \frac{+\forall x.F}{+[T/x]F} \qquad\qquad -\forall \quad \frac{-\forall x.F}{-[y/x]F}$$

where $T{:}\tau[x]$ (for $+\forall$) and where $y{:}\tau[x]$ (for $-\forall$).

$$+\lambda \quad \frac{+F}{+G} \qquad\qquad -\lambda \quad \frac{-F}{-G}$$

where for each rule F > G, as defined in §2.2.

The last rule has a character different from these *logical* rules. It has no premiss and two conclusions:

$$\text{Cut} \quad \frac{}{+F \qquad -F}$$

Cut will be seen to be a redundant but nevertheless useful rule.

4.2. Semantic Trees and Derivations

A s*emantic tree* is a binary tree with nodes that are signed formulas. A semantic tree *based on* a given sequent is defined as follows:

1. A tree with a single branch consisting of one or more nodes +F and –G, where F is from the antecedent and G is from the succedent of the sequent, is a semantic tree based on the sequent.
2. Given a semantic tree based on a sequent, a tree obtained from it in any of the following ways is a semantic tree based on the sequent:
   .1. By attaching the conclusion of a single conclusion rule to the leaf of a branch a node of which is the premiss of the rule; provided that if the rule is –∀ then y does not have a free occurrence in the premiss of the rule nor in any node above it.
   .2. By attaching the two conclusions of the +↓ rule on separate branches to the leaf of a branch a node of which is the premiss of the rule.
   .3. By attaching +F and –F on separate branches to the leaf of a branch.

A branch of a semantic tree is *closed* if there is a *closing pair* of nodes F and –F on the branch. A semantic tree is *closed* if each of its branches is closed. A *derivation* of a sequent is a closed semantic tree based on the sequent.

The cut rule is redundant in the sense that a derivation of a sequent in which it is used can be replaced by a derivation in which it is not used. This is a corollary to the completeness theorem for TT of [19] that can be adapted for ITT. Nevertheless, the rule is useful since it allows for the reuse of previously given derivations. This is illustrated in one of the example derivations given in §4.3.

4.3. Example Derivations

The example derivations make use of definitions of the usual logical connectives defined in §2.3 and of the semantic rules that can be derived for them. These will be left to the reader to state and justify.

The following notational conventions will be followed. Strings of lower and upper case Latin letters and numerals beginning with the letters u, v, w, x, y, and z are variables. When a term is known to be a formula, the types of constants and variables occurring it it can often be inferred and in these cases will not be declared. Strings which are not variables may be used, along with special symbols such as = and <, as names of predicates introduced by definition. Such a string may often be assumed to be polymorhic since the type of a predicate and the relationship between the types of its arguments can often be determined from its definition.

The following type and type declaration notation will be used here and in the remainder of the paper. The notation $\overline{\tau}$ denotes a sequence of n types $\tau_1, \ldots, \tau_n$, for some $n \geq 1$; thus $[\overline{\tau}]$ is the type $[\tau_1, \ldots, \tau_n]$. A type declaration $\overline{z}{:}\overline{\tau}$ is to be understood as declaring that $\overline{z}$ is a sequence $z_1, \ldots, z_n$ of distinct variables of types $\tau_1, \ldots, \tau_n$ respectively, and a declaration $\overline{s}{:}\overline{\tau}$ that $\overline{s}$ is a sequence $s_1, \ldots, s_n$ of terms of types $\tau_1, \ldots, \tau_n$ respectively for some $n \geq 1$.

4.3.1. *Intensional and Extensional Identity*

As stressed in §1 it is essential to distinguish between intensional = and extensional $=_e$ identity. They are defined

$$= \ \langle df \rangle \ (\lambda u,v.\forall Z.[Z(u) \rightarrow Z(v)])$$
$$=_e \ \langle df \rangle \ (\lambda u,v.\forall \overline{z}.[u(\overline{z}) \leftrightarrow v(\overline{z})])$$

The type of = is determined from the type of its arguments; if u, v:τ, then Z:$[\tau]$ and =:$[\tau,\tau]$. Similarly, if $\overline{z}{:}\overline{\tau}$ then u,v:$[\overline{\tau}]$ and $=_e$:$[[\overline{\tau}],[\overline{\tau}]]$. The usual infix notation will be used for the identities.

In TT and in ITT, the sequent $\vdash \forall x,y.[x=y \rightarrow x=_e y]$ is derivable when $x,y:[\overline{\tau}]$, but not when x,y:1 since then $x=_e y$ is not wellformed. In ITT, on the other hand, each instance of the sequent scheme

IdEId) $\quad T1=T2 \vdash T1=_e T2$

is derivable, when $T1,T2:[\overline{\tau}]\cap 1$; that is, if T1 and T2 are each of type $[\overline{\tau}]$ and also of type 1 by clause (4) of type membership, so that in each of T1 and T2 only variables of type 1 have a free occurrence. Here is a full annotated derivation.

$+T1=T2$	initial node
$-T1=_e T2$	initial node
$+(\lambda u,v.\forall Z.[Z(u) \rightarrow Z(v)])(T1,T2)$	defn of =
$+(\lambda u.\forall Z.[Z(T1) \rightarrow Z(v)])(T2)$	$+\lambda$
$+\forall Z.[Z(T1) \rightarrow Z(T2)]$	$+\lambda$
$+[S(T1) \rightarrow S(T2)]$	$+\forall$ with S <df> $\lambda w.\ T1=_e w$
--	
L R	
$-S(T1)$	$+\rightarrow$
$-(\lambda w.\ T1=_e w)(T1)$	defn S
$-T1=_e T1$	$-\lambda$
$-\forall \overline{z}.[T1(\overline{z}) \leftrightarrow T1(\overline{z})]$	defn $=_e$
$-[T1(\overline{z}) \leftrightarrow T1(\overline{z})]$	$-\forall$
$-[[T1(\overline{z}) \rightarrow T1(\overline{z})] \wedge [T1(\overline{z}) \rightarrow T1(\overline{z})]]$	defn $\leftrightarrow$
--	
LL LR	
$-[T1(\overline{z}) \rightarrow T1(\overline{z})]$	$-\wedge$
$+T1(\overline{z})$	$-\rightarrow$
$-T1(\overline{z})$	$-\rightarrow$
======	
LR	
repeat LL	
R	
$+S(T2)$	$+\rightarrow$
$+(\lambda w.\ T1=_e w)(T2)$	defn S
$+T1=_e T2$	$+\lambda$
=======	

This derivation could be shortened by using the following derivable rules of deduction for =:

+=	$+s=t$		−=	$-s=t$
	---------------------			-----------
	$-T(s)$	$+T(t)$		$+Z(s)$
				$-Z(t)$

where if $s,t:\tau$, then $T:[\tau]$, and $Z:[\tau]$, with Z not free in a node above the conclusion of −=. Derivations of these rules are left to the reader. Comparable rules for $=_e$ can be derived.

For those familiar with a Gentzen sequent calculus presentation of logic such as appears in [12, 13], this derivation can be converted as follows. For any node η that is either the last of the initial nodes or a node below the last, define $\Gamma[\eta]$ and $\Theta[\eta]$ to be the set of formulas for which +F, respectively –F, is the node η or a node above η. When η is the last of the initial nodes, then $\Gamma[\eta] \vdash \Theta[\eta]$ is the sequent $\Gamma \vdash \Theta$ on which the derivation is based. The given derivation is then an upside down transparent translation of a Gentzen derivation of the sequent $\Gamma \vdash \Theta$, without structural rules, with a node $\Gamma[\eta] \vdash \Theta[\eta]$ for each node η below the initial nodes.

4.3.2. *Zero and Successor*

Two definitions that make use of = when it is assumed to have type [1,1] are:

  0 <df> $(\lambda u.\neg u{=}u)$
  S <df> $(\lambda u,v.u{=}v)$.

Here u:1 and 0:[1], but also 0:1, since no variable has a free occurrence in 0. Thus $0{:}[1]{\cap}1$. Similarly, $S{:}[1,1]{\cap}1$, so that $S(x){:}[1]{\cap}1$. This dual typing of S(x) is exploited in derivations of the following two sequents related to axioms of Peano:

S.1. $\vdash \forall x,y.[S(x)=S(y) \rightarrow x=y]$
S.2. $\vdash \forall x.\neg S(x)=0$

These two sequents ensure that the terms in the sequence 0, S(0), S(S(0)), ... can be proven to be distinct. In TT this is accomplished by adding an axiom of infinity.

A derivation of (S.1) that is not a derivation in TT follows. It illustrates the use of the cut rule for the reuse of derivations by making use of the derivation of (IdEId). The derivation is abbreviated somewhat by omitting some obvious nodes.

```
–∀x,y.[S(x)=S(y) → x=y]
–[S(x)=S(y) → x=y]
+S(x)=S(y)
–x=y
---------------------------------------------------- Cut
+S(x)=_eS(y)                        –S(x)=_eS(y)
+∀z.[S(x)(z) ↔ S(y)(z)]             ========= IdEId
+[S(x)(y) ↔ S(y)(y)]
+[S(y)(y) → S(x)(y)]
---------------------------------
–S(y)(y)              +S(x)(y)
–y=y                  +x=y
====                  ====
```

The following derivation of (S.2) is also not a derivation in TT since it makes use of the dual typing of S(x) and of 0.

```
–∀x.¬S(x)=0
–¬S(x)=0
+S(x)=0
--------------------------------------------- Cut
+S(x)=_e0                       –S(x)=_e0
+∀z.[S(x)(z) ↔ 0(z)]            ======== IdEId
+[S(x)(x) ↔ 0(x)]
```

```
+[S(x)(x) → 0(x)]
----------------------------------
–S(x)(x)                +0(x)
–x=x                    +¬x=x
======                  –x=x
                        =====
```

4.3.3. *Ordered Pair*

The definition of ordered pair comes from [5].

⟨⟩ <df> (λu,v,w.w(u,v))

If u:σ and v:τ, then w:[σ,τ] and ⟨⟩ :[σ,τ,[σ,τ]]. However here it will be assumed that u,v:1. The usual infix notation for ordered pairs will be used. Derivable sequents are

OP.1. ⊢ ∀x1,y1,x2,y2.[⟨x1,y1⟩ = ⟨x2,y2⟩ → x1=x2 ∧ y1=y2]
OP.2. ⊢ ∀x,y.¬⟨x,y⟩ = 0

Note the similarity to (S.1) and (S.2) and note that (OP.2) combines a definition of ordered pair from the lambda calculus with a definition of 0 from set theory. An important consequence of the pairs (S.1) and (S.2) and (OP.1) and (OP.2) of sequents being derivable will become apparent in §5.

The definitions of projection functions that select respectively the first and second members of a pair are also from [5].

Hd <df> (λw.w(λu,v.u))
Tl <df> (λw.w(λu,v.v))

That these definitions of ordered pair, head, and tail are appropriate is confirmed by the following derivable sequents.

Hd.1. ⊢ ∀x,y.Hd(⟨x,y⟩) = x
Tl.1. ⊢ ∀x,y.Tl(⟨x,y⟩) = y

Here is a derivation of (Hd.1) that illustrates the use of the +λ rule for the λ-reduction to normal form of a first order term.

```
–∀x,y.Hd(⟨x,y⟩) = x
–Hd(⟨x,y⟩) = x
+Z(Hd(⟨x,y⟩))                               –=
–Z(x)                                       –=
+Z((λw.w(λu,v.u))(⟨x,y⟩))                   defn of Hd
+Z((λw.w(λu,v.u))((λu,v,w.w(u,v))(x,y)))    defn of ⟨⟩
+Z((λw.w(λu,v.u))((λw.w(x,y))))             +λ rule
+Z((λw.w(x,y))(λu,v.u))                     +λ rule
+Z((λu,v.u)(x,y))                           +λ rule
+Z(x)                                       +λ rule
====
```

Derivations of (OP.1) and (OP.2) will be left to the reader.

5. FOUNDATIONS for RECURSIONS

The purpose of this section is to provide some fundamental results upon which the many applications of recursive definitions can be based. The recursive definitions most commonly used in mathematics and computer science are least predicate definitions and more rarely greatest predicate definitions. In §5.1 a general method is described for providing these definitions; it makes use of terms called *recursion generators*. The first order definition of Horn sequent is generalized in §5.2, and finite sets of Horn sequents defining a single constant are shown to provide a "sugared" definition of a recursion generator in terms of which the constant can be defined to be the name of a predicate. These results are generalized in §5.3 for simultaneous recursions.

All of the results proved in [13] can be established in ITT using the foundations sketched in these first three subsections. A new topic is broached in §5.4, namely the "computation" of a recursively defined predicate through iterations of the recursion generator defining the predicate.

5.1. Recursion Generators

A term of type $[[\overline{\tau}], \overline{\tau}]$ is a *recursion generator* for a predicate of type $[\overline{\tau}]$. Here some fundamental properties of recursion generators for unary predicates are described; these properties can be easily generalized for recursion generators for n-ary predicates, n > 1. Derivations of the properties are not given; these can be found in [11].

The basis for least and greatest set definitions of predicates of type $[\tau]$ are the predicates Lt and Gt defined as follows:

Lt <df> $(\lambda wg,u.\forall Z.[\forall x.[wg(Z,x) \rightarrow Z(x)] \rightarrow Z(u)])$
Gt <df> $(\lambda wg,u.\exists Z.[\forall x.[Z(x) \rightarrow wg(Z,x)] \wedge Z(u)])$

Here u, $x{:}\tau$, $Z{:}[\tau]$, and $wg{:}[[\tau], \tau]$. A recursion generator is so called because of the following two derivable sequents that express generalizations of mathematical induction for the unary predicates Lt(wg) and Gt(wg):

Lt.1. $\vdash \forall wg,Y.[\forall x.[wg(Y,x) \rightarrow Y(x)] \rightarrow \forall x.[Lt(wg)(x) \rightarrow Y(x)]]$
Gt.1. $\vdash \forall wg,Y.[\forall x.[Y(x) \rightarrow wg(Y,x)] \rightarrow \forall x.[Y(x) \rightarrow Gt(wg)(x)]]$,

where $Y{:}[\tau]$. For example, define the recursion generator RN of type [[1],1] as follows:

RN <df> $\lambda Z,y.[y=0 \vee \exists x.[Z(x) \wedge y=S(x)]$

where =, 0, and S are defined in §4.3, and define N

N <df> Lt(RN)

Then (Lt.1) is the Peano axiom expressing mathematical induction. Two other of Peano's axioms follow immediately from (S.1) and (S.2) of §4.3. The remaining two are

N.1. $\vdash N(0)$
N.2. $\vdash \forall x.[N(x) \rightarrow N(S(x))]$

These follow from a sequent that can be derived for positive recursion generators.

5.1.1. *Positive Recursion Generators*

Let wg be a recursion generator. Few useful properties of Lt(wg) and Gt(wg), beyond the instantiations of (Lt.1) and (Gt.1), can be derived without an additional assumption on wg, namely that the occurrence of Z in wg(Z,x) is *positive*. Positive occurrences of $Z{:}[\overline{\tau}]$ in a formula are defined inductively on the alternative definition of formula given in §2.3.

1. The initial occurrence of Z in $Z(\overline{s})$ is positive, where $\overline{s}{:}\overline{\tau}$.
2. A positive occurrence in F or G is a positive occurrence in $[F \wedge G]$ and $[F \vee G]$.
3. A positive occurrence in F is a positive occurrence in $\exists x.F$, where x is any variable other than Z, of any type.
4. A posititve occurrence in $[s/u]T(\overline{s})$ is a positive occurrence in $(\lambda u.T)(s,\overline{s})$, where $\overline{s}$ may be of any length including 0.

Following is the fundamental theorem on positive occurrences. It is stated for $Z{:}[\tau]$ but is immediately generalizable for $Z{:}[\overline{\tau}]$. The theorem makes use of the predicate UB expressing the existence of an upper bound for a binary predicate $w{:}[\sigma, \tau]$, where σ may be any type.

$$\mathrm{UB}\ \text{<df>}\ (\lambda w.\forall x1,x2.\exists x.\forall y.[[w(x1,y) \vee w(x2,y)] \rightarrow w(x,y)])$$

Theorem

Let all free occurrences of $Z{:}[\tau]$ in the formula F be positive. Then the following sequents are derivable, where in (M), $X,Y{:}[\tau]$ are without free occurrences in F, and in (C), $X{:}[\sigma, \tau]$ and $x{:}\sigma$ are without free occurrences in F.

M. $\vdash \forall X,Y.[\forall x.[X(x) \rightarrow Y(x)] \rightarrow [[X/Z]F \rightarrow [Y/Z]F]]$
C. $\vdash [\forall X{:}UB].[[(\lambda v.\exists x.X(x,v))/Z]F \rightarrow \exists x.[(\lambda v.X(x,v))/Z]F]]$

Two fundamental consequences of the theorem for recursion generators are expressed in terms of two predicates of recursion generators Mon, *monotonic*, and Con, *continuous*

$$\mathrm{Mon}\ \text{<df>}\ (\lambda wg.\forall X,Y.[\forall x.[X(x) \rightarrow Y(x)] \rightarrow \forall x.[wg(X,x) \rightarrow wg(Y,x)]])$$
$$\mathrm{Con}\ \text{<df>}\ (\lambda wg.[\forall X{:}UB].\forall y.[wg((\lambda v.\exists x.X(x,v)), y) \rightarrow \exists x.wg((\lambda v.X(x,v)), y)]\ ])$$

Corollary

A positive recursion generator wg is monotonic and continuous; that is $\vdash$ Mon(wg) and $\vdash$ Con(wg) are derivable.

The definition of Con has been suggested by Scott's theory of computable functions; see for example [18]. The definition makes use of the existential quantifier, but it could equally well have been defined in terms of the universal quantifier. For define the predicate LB

$$\mathrm{LB}\ \text{<df>}\ (\lambda w.\forall x1,x2.\exists x.\forall y.[w(x,y) \rightarrow [w(x1,y) \wedge w(x2,y)]])$$

Then the following sequent is derivable

Con∀. $\vdash [\forall wg{:}Con].[\forall X{:}LB].\forall y.[\forall x.wg((\lambda v.X(x,v)), y)] \rightarrow wg((\lambda v.\forall x.X(x,v)), y)]$

Conversely, if Con is defined in terms of the universal quantifier, then a sequent (Con∃) obtained from (Conv∀) by appropriate changes is derivable.

5.1.2. *Properties of Lt(wg) and Gt(wg) for Monotone Recursion Generators*

The following sequents are derivable:

Lt.2. $\vdash [\forall wg{:}Mon].\forall x.[wg(Lt(wg),x) \rightarrow Lt(wg)(x)]$
Lt.3. $\vdash [\forall wg{:}Mon].\forall x.[Lt(wg)(x) \rightarrow wg(Lt(wg),x)]$

Gt.2. $\vdash [\forall wg{:}Mon].\forall x.[Gt(wg)(x) \rightarrow wg(Gt(wg),x)]$
Gt.3. $\vdash [\forall wg{:}Mon].\forall x.[wg(Gt(wg),x) \rightarrow Gt(wg)(x)]$

LtGt. $\vdash [\forall wg{:}Mon].\forall x.[Lt(wg)(x) \rightarrow Gt(wg)(x)]$

The sequents (N.1) and (N.2) given earlier follow directly from (Lt.2). The sequents (Lt.2) and (Lt.3), respectively (Gt.2) and (Gt.3), can be understood to express that Lt(wg), respectively Gt(wg), is a fixed point of an extensional identity. The sequent LtGt asserts that Lt(wg) is the least and Gt(wg) the greatest fixed point. The definition needed is

FixPt <df> $\lambda wg,v.wg(v)=_e v$

The derivable sequents expressing the fixed point results are

FixLt. $\vdash FixPt(wg, Lt(wg))$
FixGt. $\vdash FixPt(wg, Gt(wg))$

These fixed point equations express properties of Lt(wg) and Gt(wg) for positive recursion generators wg. The properties have been obtained strictly from the definitions of the predicates alone, in contrast to the role of fixed point equations in LCF. [18]

The following sequents are also derivable.

RG∃. $\vdash [\forall wg{:}Mon].\forall X,y.[\exists x.wg((\lambda v.X(x,v)), y) \rightarrow wg((\lambda v.\exists x.X(x,v)), y)]$
RG∀. $\vdash [\forall wg{:}Mon].\forall X,y.[wg((\lambda v.\forall x.X(x,v)), y) \rightarrow \forall x.wg((\lambda v.X(x,v)), y)]$

Note their relationship to the definition of Con. Properties of Lt(wg) and Gt(wg) for continuous recursion generators wg will be described in §5.4.

5.2 Horn Sequents as Sugared Definitions of Recursion Generators

A sequent is said to be a *Horn* sequent if it has the form $[C/Z]F \vdash C(\overline{s})$ where C is a constant of type $[\overline{\tau}]$ that has no occurrence in the formula F, $\overline{s}{:}\overline{\tau}$, and all free occurrences of Z in F are positive. C is said to be the *defined* constant of the Horn sequent. A sequent $\vdash C(\overline{s})$ with an empty antecedent is understood to be the Horn sequent True $\vdash C(\overline{s})$, where True <df> $\forall x.x{=}x$. A sequent $[C/Z]F_1, \ldots, [C/Z]F_k \vdash C(\overline{s})$ with more than one formula of the form [C/Z]F in the antecedent is understood to be the sequent $[C/Z][F_1 \wedge \ldots \wedge F_k] \vdash C(\overline{s})$.

Let Δ be a finite set of Horn sequents all with the same defined constant C:

HC_i. $[C/Z]F_i \vdash C(s_{i,1}, \ldots, s_{i,n})$, $1 \le i \le m$.

From Δ a recursion generator R(Δ) can be defined as follows

$$R(\Delta) \text{ <df> } \lambda Z, z_1, \ldots, z_n.[\exists \overline{x}_1.[F_1 \wedge z_1{=}s_{1,1} \wedge \ldots \wedge z_n{=}s_{1,n}] \vee \ldots \vee \exists \overline{x}_m.[F_m \wedge z_1{=}s_{m,1} \wedge \ldots \wedge z_n{=}s_{m,n}]]$$

Here $\overline{x}_i$ is a sequence of all the variables other than Z with a free occurrence in F_i or in any one of $s_{i,j}$, and $\overline{z}:\overline{\tau}$, where no z_k has a free occurrence in F_i or in any one of $s_{i,j}$. Let the constant C be defined:

C <df> Lt(R(Δ))

Then it is not difficult to establish the following result from (Lt.2):

Theorem 1

Each of the sequents (HC_i) is derivable.

It is for this reason that a set of Horn sequents all with the same defined constant can be regarded as a "sugared" definition of a recursion generator. Of course a recursion generator R(Δ) defines also a greatest fixed point Gt(R(Δ)), but this fact has received little attention.

It is natural to ask if the converse

CHC_i. $\quad C(s_{i,1}, ..., s_{i,n}) \vdash [C/Z]F_i$

of (HC_i) is derivable for each i. Theorem 1 is a consequence of (Lt.2). That (CHC_i) is derivable for each i is a consequence of (Lt.3) but only under certain conditions.

The set Δ of Horn sequents is said to be *disjoint* if the following sequents are derivable:

D.1. $\quad \vdash \forall\overline{x}, \overline{y}.[[s_{i,1}=[\overline{y}/\overline{x}]s_{i,1} \wedge ... \wedge s_{i,n}=[\overline{y}/\overline{x}]s_{i,n}] \rightarrow \overline{x}=\overline{y}]$, $1\leq i\leq m$.
D.2. $\quad \vdash \forall\overline{x}, \overline{y}.\neg[s_{i,1}=[\overline{y}/\overline{x}]s_{k,1} \wedge ... \wedge s_{i,n}=[\overline{y}/\overline{x}]s_{k,n}]$, $1\leq i,k\leq m$ with $i\neq k$.

where $\overline{y}$ is the same length as $\overline{x}$, the variables $\overline{y}$ have no free occurrence in any $s_{i,j}$, and $\overline{x}=\overline{y}$ is the conjunction of all formulas $x_k=y_k$.

Theorem 2

Each of the sequents (CHC_i) is derivable if Δ is a disjoint set of Horn sequents.

The importance of the properties of zero, successor, and ordered pair stated in the derivable sequents (S.1), (S.2), (OP.1), and (OP.2) of §4.3, arises in part from the need to derive the sequents (D.1) and (D.2).

5.3. Simultaneous Recursions

The presentation will be for two simultaneous recursions, but can be generalized. Terms R and S are *simultaneous* recursion generators of predicates of types $[\overline{\sigma}]$ and $[\overline{\tau}]$ respectively, if they are of the types $[[\overline{\sigma}], [\overline{\tau}], \overline{\sigma}]$ and $[[\overline{\tau}], [\overline{\sigma}], \overline{\tau}]$ respectively. The simultaneous least predicate operators for recursion generators wg1 and wg2 of these types is defined

SLt1 <df> $(\lambda wg1,wg2,\overline{u}.\forall X,Y.[[\forall\overline{z}1.[wg1(X, Y, \overline{z}1) \rightarrow X(\overline{z}1)] \wedge$
$\forall\overline{z}2.[wg2(X, Y, \overline{z}2)) \rightarrow Y(\overline{z}2))]] \rightarrow X(\overline{u})]$

SLt2 <df> $(\lambda wg1,wg2,\overline{v}.\forall X,Y.[\forall\overline{z}1.[wg1(X, Y, \overline{z}1) \rightarrow X(\overline{z}1)] \wedge$

$$\forall\bar{z}2).[wg2(X, Y, \bar{z}2)) \rightarrow Y(\bar{z}2))] \rightarrow Y(\bar{v})]$$

Here $\bar{z}1{:}\bar{\sigma}$ and $\bar{z}2{:}\bar{\tau}$, and $\bar{u}{:}\bar{\sigma}$ and $\bar{v}{:}\bar{\tau}$. Corresponding greatest predicate operators SGt1 and SGt2 can be similarly defined.

The two wellfounded predicates Pr1 and Pr2 and the two non-wellfounded predicates Qr1 and Qr2 defined by the simultaneous recursion are

Pr1 <df> $(\lambda\bar{u}.SLt1(wg1,wg2)(\bar{u}))$
Pr2 <df> $(\lambda\bar{v}.SLt(wg1,wg2)(\bar{v}))$
Qr1 <df> $(\lambda\bar{u}.SGt(wg1,wg2)(\bar{u})$
Qr2 <df> $(\lambda\bar{v}.SGt(wg1,wg2)(\bar{v}))$

Derivable sequents expressing induction principles for Pr1 and Pr2 are

SLt1.1. $\vdash \forall X.[\exists Y.[\forall\bar{z}1).[wg1(X, Y, \bar{z}1)) \rightarrow X(\bar{z}1))] \wedge \forall\bar{z}2.[wg2(X, Y, \bar{z}2)) \rightarrow Y(\bar{z}2))]]$
$\rightarrow \forall\bar{z}1).[Pr1(\bar{z}1) \rightarrow X(\bar{z}1))]]$

SLt2.1. $\vdash \forall Y.[\exists X.[\forall\bar{z}1).[wg1(X, Y, \bar{z}1)) \rightarrow X(\bar{z}1))] \wedge \forall\bar{z}2.[wg2(X, Y, \bar{z}2)) \rightarrow Y(\bar{z}2))]]$
$\rightarrow \forall\bar{z}2).[Pr2(\bar{z}2) \rightarrow Y(\bar{z}2))]]$

Corresponding sequents for Qr1 and Qr2 can also be derived. Fixed point sequents similar to (Lt.2) to (Gt.3) can be derived.

Consider now a set Δ of Horn sequents with members of the following forms

SHC. $[C/Z][D/W]F \vdash C(s_1, ..., s_m)$
$[C/Z][D/W]G \vdash D(t_1, ..., t_n)$

such that there are members for which each of F and G has a free occurrence of Z and a free occurrence of W. Simultaneous recursion generators can be defined from Δ that provide definitions for the constants C and D. Theorems 1 and 2 can be repeated for Δ with an appropriate update for the definition of "disjoint".

5.4. Iterating Recursion Generators

Given a unary recursion generator wg of type $[[\tau], \tau]$ and a predicate ws:$[\tau]$, consider the following sequence of type $[\tau]$ predicates: ws, $(\lambda v.wg(ws,v))$, $(\lambda v.wg((\lambda v.wg(ws,v),v)))$, … . The sequence is the result of *iterating* wg, beginning with ws. A recursion generator RIt(wg,ws) for the iteration predicate It(wg,ws) is defined by the following two Horn sequents

It.1. $ws(x) \vdash It(wg,ws)(0,x)$
It.2. $wg((\lambda v.It(wg,ws)(xn,v)),x) \vdash It(wg,ws)(S(xn),x)$

A second *cumulative* iteration of wg from ws results in the sequence ws, $(\lambda v.[ws\vee wg(ws,v)])$, $(\lambda v.[ws\vee wg(ws,v)]\vee wg((\lambda v.[ws\vee wg(ws,v)]),v))$, … . A recursion generator RCIt(wg,ws) for the cumulative iteration predicate CIt(wg,ws) is defined by the following two Horn sequents

CIt.1. $ws(x) \vdash CIt(wg,ws)(0,x)$
CIt.2. $[wg((\lambda v.CIt(wg,ws)(xn,v)),x) \vee CIt(wg,ws)(xn,x)] \vdash CIt(wg,ws)(S(xn),x)$

Since the sequents (It.1) and (It.2), and the sequents (CIt.1) and (CIt.2), satisfy the conditions (D.1) and (D.2), by theorems 1 and 2 these sequents and their converses are derivable when It and CIt are defined

It <df> (λ.wg,ws.Lt(RIt(wg,ws)))
CIt <df> (λ.wg,ws.Lt(RCIt(wg,ws)))

That CIt can be used to "compute" the predicate Lt(wg) for any positive recursion generator wg is expressed in the following two derivable sequents:

CItLt.1. $\vdash$ [∀wg:Mon].∀y.[[∃xn:N].CIt(wg, ∅)(xn,y) → Lt(wg)(y)]
CItLt.2. $\vdash$ [∀wg:Mon∩Con].∀y.[Lt(wg)(y) → [∃xn:N].CIt(wg, ∅)(xn,y)]

where ∅:[τ] is the empty predicate, and Mon∩Con the conjunction of Mon and Con.

An understanding of the relationship between this "computation" of Lt(wg) and that of [17] requires further study.

Similarly, that It can be used to "compute" the predicate Gt(wg) for any positive recursion generator wg is expressed in the following two sequents:

ItGt.1. $\vdash$ [∀wg:Mon].∀y.[Gt(wg)(y) → [∀xn:N].It(wg,V)(xn,y)]
ItGt.2. $\vdash$ [∀wg:Mon∩Con].∀y.[[∀xn:N].It(wg,V)(xn,y) → Gt(wg)(y)]

where V:[τ] is the universal predicate.

6. CATEGORY THEORY and an EXTENSION to ITT

In §6.1 a result for category theory is described that is a correction to the main result of [12]. In §6.2 an extension to ITT is described in which impredicative sets, such as Russell's, can be defined.

6.1. Category Theory and ITT

The main result of [12], expressed in theorem 5.4, cannot be duplicated in ITT since the proof of lemma 5.2 makes use of the feature of NaDSet that leads to its inconsistency; namely allowing a single type of variable to play the role of both first and second order variables. Using the notation of the present paper, the main result is expressed in terms of a predicate Cat defined

C. Cat <df> λAr, $=_a$, Sr, Tg, Cp.Category(Ar, $=_a$, Sr, Tg, Cp)

where Category(Ar, $=_a$, Sr, Tg, Cp) is the conjunction of the axioms of category theory with the variables interpreted as follows: Ar is the unary predicate of arrows or morphisms, $=_a$ the binary predicate of identity of arrows, Sr a binary predicate with first argument an arrow and second argument its source object, Tg a binary predicate with first argument an arrow and second argument its target object, and Cp a ternary predicate the third argument of which is the composite of the arrows that are its first two arguments.

The terms $\mathbb{Ar}$, $=_{\mathbb{a}}$, $\mathbb{Sr}$, $\mathbb{Tg}$, and $\mathbb{Cp}$ are defined to be respectively the unary predicate Functor of categories, extensional identity between functors, the source and target predicates for functors, and the composition of functors. Theorem 5.4 asserts that the sequent

D. $\vdash$ Cat($\mathbb{Ar}$, $=_{\mathbb{a}}$, $\mathbb{Sr}$, $\mathbb{Tg}$, $\mathbb{Cp}$)

is derivable, expressing that the category of categories is a category. But in ITT Cat($\mathbb{Ar}$, $=_{\mathbb{a}}$, $\mathbb{Sr}$, $\mathbb{Tg}$, $\mathbb{Cp}$) is not wellformed. It is wellformed however if Cat is replaced by a form Cat* of Cat defined exactly as Cat in (C) except with the understanding that the types of the variables are raised to accomodate the types of the arguments.

6.2. An Extension to ITT

The variables of ITT have two distinct roles to play, as quantifiable variables and as abstraction variables. These two roles could be served in formulas by syntactically distinct variables as is evident from Schütte's definition of formula given in §2.3. As an abstraction variable, the variable x in clause (4) of the definition is used purely as a *placeholder* and need never be interpreted; that is, since $\Phi([], (\lambda x.T)(S, S_1, \ldots , S_n))$ can be defined to be $\Phi([], ([S/x]T)(S_1, \ldots , S_n))$ it is unnecessary to define $\Phi(\tau[x], x)$. But since $\Phi([], \forall x.F)$ is defined in terms of $\Phi^x([], F)$, it is necessary to define $\Phi(\tau[x], x)$ in this case. By using distinct variables for these two roles, and interpreting only quantification variables, a consistent extension of ITT can be constructed in which such impredicative sets as Russell's can be defined.

The Russell set R is defined to be $(\lambda u.\neg u(u))$. For $u(u)$ to be wellformed it is necessary that the two occurrences of u have different types; for example the first occurrence could have type [1] while the second has type 1; or briefly that $u:[1]\cap 1$. Then $u(u):[]$, $\neg u(u):[]$, and $(\lambda u.\neg u(u)):[[1]\cap 1]$. Arguments for R are therefore the empty predicate $\varnothing$ defined to be $\lambda u.\neg u=u$. and the universal predicate V defined to be $\lambda u.u=u$. It is then not difficult to derive the sequents

$\vdash \neg R(\varnothing)$ and $\vdash R(V)$

corresponding to the two assertions about R that Russell made when he defined it. It is essential to note, however, that R(R) is not wellformed since $R:[[1]\cap 1]\cap 1$, so that no contradiction can be derived from the fact that R is wellformed.

Whether such an extension to ITT has applications in programming semantics remains to be seen; however, there appears to be no obstacle to reproducing within the extended ITT all the applications of non-wellfounded sets described in [2].

7. REFERENCES

[1] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, Revised Edition, North-Holland. 1985.

[2] Jon Barwise & Lawrence Moss. *Vicious Circles*, CSLI Publications, 1996.

[3] E.W. Beth. Semantic Entailment and Formal Derivability, *Mededelingen de Koninklijke Nederlandse Akademie der Wetenschappen, Afdeeling Letterkunde, Nieuwe Reeks,* 18, no.13, 309-342, 1955

[4] Alonzo Church. A Formulation of the Simple Theory of Types, *J. Sym. Logic*, 5, 56-68, 1940.

[5] Alonzo Church. *The Calculi of Lambda Conversion*, Princeton U. Press, 1941.

[6] Paul C. Gilmore. A Consistent Naive Set Theory: Foundations for a Formal Theory of Computation, IBM Research Report RC 3413, June 22,1971.

[7] Paul C. Gilmore. Combining Unrestricted Abstraction with Universal Quantification, *To H.B. Curry: Essays on Combinatorial Logic, Lambda Calculus and Formalism*, Editors J.P. Seldin, J.R. Hindley, Academic Press, 99-123. This is a revised version of [6], 1980.

[8] Paul C. Gilmore. Natural Deduction Based Set Theories: A New Resolution of the Old Paradoxes, *J.Sym. Logic*, 51, 393-411, 1986.

[9] Paul C. Gilmore. A Foundation for the Entity Relationship Approach: How and Why, *Proceedings of the 6th Entity Relationship Conference*, S.T. March (Ed.), North-Holland 95-113, 1988.

[10] Paul C. Gilmore. NaDSyL and some Applications, *Computational Logic and Proof Theory,* Georg Gottlob, Alexander Leitsch, & Daniele Mundici (eds.), The Kurt Gödel Colloquium 97, Lecture Notes in Computer Science 1289. 153-166. Springer-Verlag, 1997.

[11] Paul C. Gilmore. *An Impredicative Symbolic Logic and Some Applications*, a monograph on ITT in preparation.

[12] Paul C. Gilmore & George K. Tsiknis. A Formalization of Category Theory in NaDSet, *Theoretical Computer Science*, vol. 111, 211-253, 1993.

[13] Paul C. Gilmore & George K. Tsiknis. Logical Foundations for Programming Semantics, *Theoretical Computer Science*, vol. 111, 253-290, 1993.

[14] J.Y. Girard, Letter to author, March 16, 1994.

[15] Michael J.C. Gordon. Set Theory, Higher Order Logic or Both?, *Proc. 9'th International Conference on Theorem Proving in Higher Order Logic*, Joakim von Wright, Jim Grundy, and John Harrison, editors, Turku, Finland 26-30 August 1996, 191-202, 1996.

[16] Leon Henkin. Some Notes on Nominalism, *J. Sym. Logic*, 18, 19-29, 1953.

[17] Gopalan Nadathur & Dale Miller. Higher-Order Logic Programming, CS-1994-38, Dept of Computer Science, Duke University. To appear in the *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. Gabbay, C. Hogger and A. Robinson (eds.), Oxford University Press.

[18] L.C. Paulson, *Logic and Computation, Interactive proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2, Cambridge U. Press, 1990.

[19] Dag Prawitz. Hauptsatz for Higher Order Logic, *J. Sym. Logic*, 33, 452-457, 1968.

[20] K. Schütte. Syntactical and Semantical Properties of Simple Type Theory, *J. Sym. Logic*, 25, 305-326, 1960.