The Seven Virtues of Simple Type Theory*

William M. Farmer†
McMaster University

20 December 2007

Abstract

Simple type theory, also known as *higher-order logic*, is a natural extension of first-order logic which is simple, elegant, highly expressive, and practical. This paper surveys the virtues of simple type theory and attempts to show that simple type theory is an attractive alternative to first-order logic for practical-minded scientists, engineers, and mathematicians. It recommends that simple type theory be incorporated into introductory logic courses offered by mathematics departments and into the undergraduate curricula for computer science and software engineering students.

1 Introduction

Mathematicians are committed to rigorous reasoning, but they usually shy away from formal logic. However, when mathematicians really need a formal logic—e.g., to teach their students the rules of quantification, to pin down exactly what a "property" is, or to formalize set theory—they almost invariably choose some form of *first-order logic*. Taking the lead of mathematicians, scientists and engineers usually choose first-order logic whenever they need a formal logic to express mathematical models precisely or to study the logical consequences of theories and specifications carefully. In science and engineering as well as in mathematics, first-order logic reigns supreme!

A formal logic can be a *theoretical tool* for studying rigorous reasoning and a *practical tool* for performing rigorous reasoning. Today mathematicians sometimes use formal logics as theoretical tools, but they very rarely

*To appear in the *Journal of Applied Logic*, doi:10.1016/j.jal.2007.11.001.

†Address: Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario L8S 4K1, Canada. E-mail: wmfarmer@mcmaster.ca.

use them as practical tools. Scientists and engineers do sometimes use formal logics as practical tools—but not often. Tomorrow things will be different. In the future, mathematicians, scientists, and engineers will routinely employ computer systems—the successors of today's computer algebra systems and computer theorem proving systems—that mechanize various aspects of mathematical reasoning. In order to be trustworthy, these systems will inevitably be based on formal logics. Also, both theoreticians and practitioners will utilize huge digital libraries of mathematics. The mathematical knowledge in these libraries will be formalized, organized, certified, searched, and retrieved with the help of formal logics.

First-order logic is certainly an effective logic for *theory*, but it is an awkward logic for *practice*. There is little built-in support for reasoning about higher-order objects such as sets and functions in first-order logic. As a result, many basic mathematical concepts, such as the transitive closure of a relation and the completeness principle for the real numbers, cannot be expressed in first-order logic directly. The overhead of formalizing basic mathematics like abstract algebra and calculus in first-order logic is quite high: one has to start with a sophisticated theory of sets or functions. Moreover, mathematical statements that are succinct in informal practice often become verbose and unwieldy when expressed in first-order logic because first-order logic lacks an abstraction mechanism for building predicates and functions and a definite description mechanism for specifying values. Mathematicians almost never use first-order logic to actually do mathematics; the benefits fall well short of justifying the pain and tedium that is involved in developing mathematical ideas in first-order logic.

Are there any logics that are more effective for practice? A good candidate is an old relative of first-order logic called *simple type theory*.

Simple type theory, also known as *higher-order logic*, is a natural extension of first-order logic. It is based on the same principles as first-order logic but differs from first-order logic in two principal ways. First, terms can be *higher-order*, i.e., they can denote higher-order values such as sets, relations, and functions. Predicates and functions can be applied to higher-order terms, and quantification can be applied to higher-order variables in formulas. Second, syntactic objects called *types*, which denote nonempty sets of values, are used to organize terms. They restrict the scope of variables, control the formation of terms, and provide a means to classify terms by their values.

Simple type theory is a logic with outstanding virtues. It is simple, elegant, highly expressive, and *practical*. Although it is familiar to many computer scientists, most mathematicians, engineers, and other scientists

have never heard of it. This is due in large part to the fact that simple type theory is rarely taught in mathematics departments at either the undergraduate or graduate level. However, an understanding of simple type theory would be beneficial to anyone who needs to work with or apply mathematical logic. This is particularly true for:

- Engineers who need to write (and read) precise specifications.
- Computer scientists who employ functional programming languages such as Lisp, ML, and Haskell.
- Software engineers who use higher-order theorem proving systems to model and analyze software systems.
- Mathematics students who are studying the foundations of mathematics or model theory.

In this paper, we will present a pure form of simple type theory that we call STT and then use it to illustrate seven virtues of simple type theory. Our ultimate objective is to show that simple type theory is an attractive alternative to first-order logic for practical-minded scientists, engineers, and even mathematicians. The paper ends with a recommendation that simple type theory be incorporated into introductory logic courses offered by mathematics departments and into the undergraduate curricula for computer science and software engineering students.

We assume that the reader is familiar with the syntax, semantics, and proof theory of first-order logic as it is usually presented in an introductory undergraduate logic course.

2 History

B. Russell proposed in 1908 [52] a logic now known as the *ramified theory of types*. Russell wanted a logic that would be free from the set-theoretic paradoxes such as Russell's paradox about the class of all classes that are not members of themselves as well as the semantic paradoxes such as Richard's paradox concerning the cardinality of the set of definable real numbers [26]. As a result, the ramified theory of types was formulated with the safety principle (the so-called *vicious-circle principle*) that the class $C = \{x : \varphi(x)\}$ can be defined only if C itself is not in the range of any variable in φ. This safety principle is enforced with a hierarchy of levels of types. Russell and A. Whitehead used the ramified theory of types as the logical basis for their

monumental, three-volume *Principia Mathematica* [56], the first attempt to formalize a significant portion of mathematics starting from first principles. The great achievement of *Principia Mathematica*, unfortunately, is marred by the fact that, in order to formalize standard proofs of induction, Russell introduced the Axiom of Reducibility which in effect nullifies the safety principle [21].

In the 1920s, L. Chwistek [8] and F. Ramsey [49] noticed that, if one is willing to give up Russell's safety principle, the hierarchy of levels of types in the ramified theory of types can be collapsed. The resulting simplified logic is called the *simple theory of types* or, more briefly, *simple type theory*. It is equivalent to the ramified theory of types plus the Axiom of Reducibility. Detailed formulations of simple type theory were published in the late 1920s and early 1930s by R. Carnap, K. Gödel, W. V. O. Quine, and A. Tarski (see [22]).

In 1940 [7] A. Church presented an elegant formulation of simple type theory, known as *Church's type theory*, that is based on functions instead of relations and that incorporates special machinery to build and apply functions (lambda-notation and lambda-conversion). Church's paper has had a profound influence on computer science, especially in the areas of programming languages, computer theorem proving, formal methods, computational logic, and formalized mathematics.[1] Today the field of *type theory* in computer science is largely the study of Church-style (classical and constructive) logics that are based on functions and equipped with lambda-notation and lambda-conversion.

Church's type theory has been extensively studied by two of Church's students, L. Henkin and P. Andrews. Henkin proved in [28] that there is a sense in which Church's type theory is complete (see section 7). Henkin also showed in [29] that Church's type theory could be reformulated using only four primitive notions: function application, function abstraction, equality, and definite description (see section 4). Andrews devised in [3] a simple and elegant proof system for Henkin's reformulation of Church's type theory (see section 6). Andrews formulated a version of Church's type theory called $\mathcal{Q}_0$ that employs the ideas developed by Church, Henkin, and himself. Andrews meticulously describes and analyzes $\mathcal{Q}_0$ in his textbook [4], and he and his students have implemented a computer theorem prover based on $\mathcal{Q}_0$ called TPS [5].

[1] In the September 2006 list of the most cited articles in Computer Science on the CiteSeer Web site (`http://citeseer.ist.psu.edu/`), Church's paper [7] is ranked 358 with 406 citations and is the oldest paper with more than 300 citations.

Like first-order logic, Church's type theory is classical in the sense that it admits nonconstructive reasoning principles such as the law of excluded middle and double negation elimination. P. Martin-Löf introduced in 1972 a constructive form of type theory now known as *Martin-Löf type theory* [42]. Most constructive type theories, including Martin-Löf type theory, embody the *Curry-Howard isomorphism* [33] that elegantly connects proving theorems in type theory to writing programs in lambda calculus. Constructive type theories also have a close connection to category theory and have been extensively used to formalize constructive mathematics and ideas from theoretical computer science [12, 39].

Since the 1980s, type theory has been a popular choice for the logical basis of computer theorem proving systems. HOL [24], IMPS [20], Isabelle [46], ProofPower [40], PVS [45], and TPS are examples of systems based on versions of Church's type theory, and Agda [11], Automath [44], Coq [10], LEGO [48], and Nuprl [9] are examples of systems based on constructive type theories.

3 The Definition of STT

There are many variants of simple type theory. STT is a version of Church's type theory [7]. STT is simple type theory boiled down to its essence. It is convenient for study, but it is not highly practical for use. In section 8, we will consider a number of ways that STT can be extended to a more practical form of simple type theory.

We will begin our exploration of simple type theory by defining the syntax and semantics of STT.

3.1 Syntax

STT has two kinds of syntactic objects. "Expressions" denote values including the truth values T (true) and F (false); they do what both terms and formulas do in first-order logic. "Types" denote nonempty sets of values; they are used to restrict the scope of variables, control the formation of expressions, and classify expressions by their values.

A *type* of STT is a string of symbols defined inductively by the following formation rules:

1. *Type of individuals*: ι is a type.

2. *Type of truth values*: $*$ is a type.

3. *Function type*: If α and β are types, then $(\alpha \rightarrow \beta)$ is a type.

Let $\mathcal{T}$ denote the set of types of STT. The definition of a type shows that $\mathcal{T}$ is composed of two *base types*, ι and $*$, and an infinite hierarchy of function types built from the base types. If a type γ denotes a domain D_γ of values, then a function type $(\alpha \rightarrow \beta)$ denotes the domain of total functions from D_α to D_β.

The *logical symbols* of STT are:

1. *Function application*: @.
2. *Function abstraction*: λ.
3. *Equality*: $=$.
4. *Definite description*: I (capital iota).
5. *Type binding*: : (colon).
6. An infinite set $\mathcal{V}$ of symbols used to construct variables (see below).

Function abstraction defines a function $x \mapsto E$ from an expression E usually involving x. Definite description builds an expression that denotes the unique value that satisfies a property P. Type binding constructs a variable by assigning a type to a member of $\mathcal{V}$.

A *language* of STT is a pair $L = (\mathcal{C}, \tau)$ where is $\mathcal{C}$ is a set of symbols called *constants* and $\tau : \mathcal{C} \rightarrow \mathcal{T}$ is a total function. That is, a language is a set of symbols with assigned types (what computer scientists usually call a "signature"). The constants are the nonlogical primitive symbols that are used to construct the expressions of the language.

An *expression* E of *type* α of an STT language $L = (\mathcal{C}, \tau)$ is a string of symbols defined inductively by the following formation rules:

1. *Variable*: If $x \in \mathcal{V}$ and $\alpha \in \mathcal{T}$, then $(x : \alpha)$ is an expression of type α.
2. *Constant*: If $c \in \mathcal{C}$, then c is an expression of type $\tau(c)$.
3. *Function application*: If A is an expression of type α and F is an expression of type $(\alpha \rightarrow \beta)$, then $(F \mathbin{@} A)$ is an expression of type β.
4. *Function abstraction*: If $x \in \mathcal{V}$, $\alpha \in \mathcal{T}$, and B is an expression of type β, then $(\lambda\, x : \alpha \,.\, B)$ is an expression of type $(\alpha \rightarrow \beta)$.
5. *Equality*: If E_1 and E_2 are expressions of type α, then $(E_1 = E_2)$ is an expression of type $*$.

Variable	$(x : \alpha)$
Constant	c
Function application	$(F @ A)$
Function abstraction	$(\lambda\, x : \alpha\ .\ B)$
Equality	$(E_1 = E_2)$
Definite description	$(\mathrm{I}\, x : \alpha\ .\ A)$

Table 1: The Six Kinds of STT Expressions

6. *Definite description*: If $x \in \mathcal{V}$, $\alpha \in \mathcal{T}$, and A is an expression of type $*$, then $(\mathrm{I}\, x : \alpha\ .\ A)$ is an expression of type α.[2]

A string of symbols is considered an expression only if it can be assigned a type according to the rules given above. Notice that the type assigned to an expression is always unique. "Free variable", "closed expression", and similar notions are defined in the obvious way.

We will see shortly that the value of a definite description $(\mathrm{I}\, x : \alpha\ .\ A)$ is the unique value x of type α satisfying A if it exists and is a canonical "error" value of type α otherwise. For example, if f is a constant with type $(\alpha \rightarrow \alpha)$, then

$$(\mathrm{I}\, x : \alpha\ .\ ((f\ @\ (x : \alpha)) = (x : \alpha)))$$

denotes the unique fixed point of f if it exists and denotes an error value otherwise.

Notice that different kinds of expressions are distinguished *by type* instead of *by form*, as shown by the following examples. An *individual constant* of L is a constant $c \in \mathcal{C}$ such that $\tau(c) = \iota$. A *formula* of L is an expression of L of type $*$. A *predicate* of L is an expression of L of type $(\alpha \rightarrow *)$ for any $\alpha \in \mathcal{T}$.

Let $A_\alpha, B_\alpha, C_\alpha, \ldots$ denote expressions of type α. We will often use the following abbreviation rules to write expressions in a more compact form:

1. A variable $(x : \alpha)$ occurring in the body B of $(\Box\, x : \alpha\ .\ B)$, where $\Box$ is λ or I, may be written as x if there is no resulting ambiguity.

2. A matching pair of parentheses in a type or an expression may be dropped if there is no resulting ambiguity.

[2]The definite description operator is often called the *iota operator* and is represented by a lower case iota (ι). Russell represented it by an inverted lower case iota (℩). We represent it by a capital iota (I) or an inverted capital iota (I), whatever one prefers.

3. A function application $(F \mathbin{@} A)$ may be written in the standard form $F(A)$.

Virtue 1 STT *has a simple and highly uniform syntax.*

The syntax of STT with types in addition to expressions is a bit more complex than the syntax of first-order logic. However, the syntax is also more uniform than the syntax of first-order logic since expressions serve as both terms and formulas and constants include individual constants, function symbols, and predicate symbols as well as constants of many other types. There are no propositional connectives or quantifiers in STT, but we will see later that these can be easily defined in STT using function application, function abstraction, and equality.

3.2 Semantics

The semantics of STT is based on "standard models". Later in the paper we will introduce another semantics based on "general models".

Fix two values T and F with T $\neq$ F to represent *true* and *false*. A *standard model* for a language $L = (\mathcal{C}, \tau)$ of STT is a triple $M = (\mathcal{D}, I, e)$ where:

1. $\mathcal{D} = \{D_\alpha : \alpha \in \mathcal{T}\}$ is a set of nonempty domains (sets).
2. $D_* = \{\mathrm{T}, \mathrm{F}\}$, the domain of truth values.
3. For $\alpha, \beta \in \mathcal{T}$, $D_{\alpha \rightarrow \beta}$ is the set of *all* total functions from D_α to D_β.
4. I maps each $c \in \mathcal{C}$ to a member of $D_{\tau(c)}$.
5. e maps each $\alpha \in \mathcal{T}$ to a member of D_α.

For each $\alpha \in \mathcal{T}$, $e(\alpha)$ is intended to be a canonical "error" value of type α.

D_ι, the domain of individuals, is the core domain of a standard model. It corresponds to the domain of values of a first-order model. Since D_*, the domain of truth values, is always the set $\{\mathrm{T}, \mathrm{F}\}$ of the standard truth values, the function domains of a standard model are determined by the choice of D_ι alone. We say that M is *infinite* if D_ι is infinite.

Fix a standard model $M = (\mathcal{D}, I, e)$ for a language $L = (\mathcal{C}, \tau)$ of STT. A *variable assignment* into M is a function that maps each variable $(x : \alpha)$ to a member of D_α. Given a variable assignment φ into M, a variable $(x : \alpha)$,

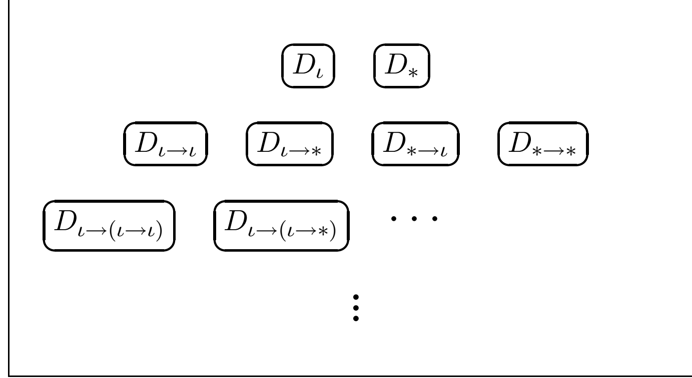

Table 2: The Domains of a Standard Model of STT

and $d \in D_\alpha$, let $\varphi[(x:\alpha) \mapsto d]$ be the variable assignment φ' into M such that $\varphi'((x:\alpha)) = d$ and $\varphi'(X) = \varphi(X)$ for all $X \neq (x:\alpha)$.

The *valuation function* for M is the binary function V^M that takes as arguments an expression of L and a variable assignment into M and that satisfies the six conditions below. (We write $V^M_\varphi(E)$ instead of $V^M(E,\varphi)$.)

1. Let E be a variable (i.e., E is of the form $(x:\alpha)$). Then $V^M_\varphi(E) = \varphi(E)$.

2. Let E be a constant of L (i.e., $E \in \mathcal{C}$). Then $V^M_\varphi(E) = I(E)$.

3. Let E be of the form $(F \mathbin{@} A)$. Then $V^M_\varphi(E) = V^M_\varphi(F)(V^M_\varphi(A))$, the result of applying the function $V^M_\varphi(F)$ to the argument $V^M_\varphi(A)$.

4. Let E be of the form $(\lambda\, x : \alpha \;.\; B)$ where B is of type β. Then $V^M_\varphi(E)$ is the function $f : D_\alpha \to D_\beta$ such that, for each $d \in D_\alpha$, $f(d) = V^M_{\varphi[(x:\alpha)\mapsto d]}(B)$.[3]

5. Let E be of the form $(E_1 = E_2)$. If $V^M_\varphi(E_1) = V^M_\varphi(E_2)$, then $V^M_\varphi(E) = \text{T}$; otherwise $V^M_\varphi(E) = \text{F}$.

6. Let E be of the form $(\mathrm{I}\, x : \alpha \;.\; A)$. If there is a unique $d \in D_\alpha$ such that $V^M_{\varphi[(x:\alpha)\mapsto d]}(A) = \text{T}$, then $V^M_\varphi(E) = d$; otherwise $V^M_\varphi(E) = e(\alpha)$.[3]

[3]Notice that the semantics for function abstraction and definite description is defined using the same trick (due to Tarski) that is used to define the semantics for universal and existential quantification in first-order logic.

Let E be an expression of type α of L and A be a formula of L. $V_\varphi^M(E)$ is called the *value* of E in M with respect to φ. $V_\varphi^M(E) \in D_\alpha$, and if E is closed, $V_\varphi^M(E)$ does not depend on φ. A value $d \in D_\alpha$ is *nameable* if there is some closed expression E of L such that d is the value of E in M. A is *valid* in M, written $M \models A$, if $V_\varphi^M(A) = \textsc{t}$ for all variable assignments φ into M. A *sentence* of L is a closed formula of L. A is a *semantic consequence* of a set Σ of sentences of L, written $\Sigma \models A$, if $M \models A$ for every standard model M for L such that $M \models B$ for all $B \in \Sigma$.

A *theory* of STT is a pair $T = (L, \Gamma)$ where L is a language of STT and Γ is a set of sentences of L called the *axioms* of T. A formula A is a *semantic consequence* of T, written $T \models A$, if $\Gamma \models A$. A *standard model* of T is a standard model M for L such that $M \models B$ for all $B \in \Gamma$.

Virtue 2 *The semantics of* STT *is based on a small collection of well-established ideas.*

The two semantics of first-order logic and STT are based on essentially the same ideas: domains of individuals, truth values, and functions; models for languages; variable assignments; and valuation functions defined recursively on the syntax of expressions.

We will conclude this section with an examination of isomorphic standard models.

Let $M = (\mathcal{D}, I, e)$ and $M' = (\mathcal{D}', I', e')$ be standard models for a language $L = (\mathcal{C}, \tau)$ of STT where $\mathcal{D} = \{D_\alpha : \alpha \in \mathcal{T}\}$ and $\mathcal{D}' = \{D'_\alpha : \alpha \in \mathcal{T}\}$. An *isomorphism* from M to M' is a set $\{\Phi_\alpha : \alpha \in \mathcal{T}\}$ of mappings such that:

1. Φ_α is a bijection from D_α to D'_α for all $\alpha \in \mathcal{T}$.
2. $\Phi_*(\textsc{t}) = \textsc{t}$ and $\Phi_*(\textsc{f}) = \textsc{f}$.
3. $\Phi_\beta(f(a)) = \Phi_{\alpha\to\beta}(f)(\Phi_\alpha(a))$ for all $\alpha, \beta \in \mathcal{T}$, $f \in D_{\alpha\to\beta}$, and $a \in D_\alpha$.
4. $\Phi_\alpha(I(c)) = I'(c)$ for all $\alpha \in \mathcal{T}$ and $c \in \mathcal{C}$ with $\tau(c) = \alpha$.

Notice that there is no requirement that $\Phi_\alpha(e(\alpha)) = e'(\alpha)$ for all $\alpha \in \mathcal{T}$. M and M' are *isomorphic* if there is an isomorphism from M to M'.

An expression is *definite description free* if it does not contain any subexpressions of the form $(\mathrm{I}\, x : \alpha \,.\, A)$.

Theorem 1 *Let* $\{\Phi_\alpha : \alpha \in \mathcal{T}\}$ *be an isomorphism from* M *to* M', φ *be a variable assignment into* M, φ' *be the variable assignment into* M' *that maps each variable* $(x : \alpha)$ *to* $\Phi_\alpha(\varphi((x : \alpha)))$, *and* E_α *be an expression of* L *that is definite description free. Then* $\Phi_\alpha(V^M_\varphi(E_\alpha)) = V^{M'}_{\varphi'}(E_\alpha)$.

Proof By induction on the structure of expressions. □

In other words, two isomorphic standard models for the same language have exactly the same structure except for the choice of error values.

4 Expressivity

On the surface, the expressivity of STT appears to be rather modest. The formulas of STT are limited to variables, constants, and definite descriptions of type $*$, equations between expressions of the same type, and applications of predicates; there are no propositional connectives nor quantifiers. However, STT can be used to reason about an infinite hierarchy of higher-order functions constructed over a domain of individuals and a domain of truth values. Since sets can be represented by predicates, the hierarchy includes a subhierarchy of higher-order sets constructed from individuals and truth values. Hidden within this hierarchy is the full power of higher-order predicate logic.

As Henkin shows in [29], the usual propositional connectives and quantifiers can be defined in a logic like STT using just function application, function abstraction, and equality. Here are their definitions in STT:

T	means	$(\lambda\, x : * \,.\, x) = (\lambda\, x : * \,.\, x)$.
F	means	$(\lambda\, x : * \,.\, \mathsf{T}) = (\lambda\, x : * \,.\, x)$.
$\neg A_*$	means	$A_* = \mathsf{F}$.
$(A_\alpha \neq B_\alpha)$	means	$\neg(A_\alpha = B_\alpha)$.
$(A_* \wedge B_*)$	means	$(\lambda\, f : (* \rightarrow (* \rightarrow *)) \,.\, f(\mathsf{T})(\mathsf{T})) =$ $(\lambda\, f : (* \rightarrow (* \rightarrow *)) \,.\, f(A_*)(B_*))$.
$(A_* \vee B_*)$	means	$\neg(\neg A_* \wedge \neg B_*)$.
$(A_* \Rightarrow B_*)$	means	$\neg A_* \vee B_*$.
$(A_* \Leftrightarrow B_*)$	means	$A_* = B_*$.
$(\forall\, x : \alpha \,.\, A_*)$	means	$(\lambda\, x : \alpha \,.\, A_*) = (\lambda\, x : \alpha \,.\, \mathsf{T})$.
$(\exists\, x : \alpha \,.\, A_*)$	means	$\neg(\forall\, x : \alpha \,.\, \neg A_*)$.
$\bot_\alpha$	means	$\mathrm{I}\, x : \alpha \,.\, x \neq x$.
$\mathsf{if}(A_*, B_\alpha, C_\alpha)$	means	$\mathrm{I}\, x : \alpha \,.\, (A_* \Rightarrow x = B_\alpha) \wedge (\neg A_* \Rightarrow x = C_\alpha)$ where $(x : \alpha)$ does not occur in A_*, B_α, or C_α.

Notice that we are using the abbreviation rules given in section 3.1. For example, the meaning of T is officially the expression

$$((\lambda\, x : * \,.\, (x : *)) = (\lambda\, x : * \,.\, (x : *))).$$

In addition to the definitions of the usual propositional connectives and quantifiers, we also included above two definitions that employ definite description. $\bot_\alpha$ is a canonical error expression of type α. if is an if-then-else expression constructor (i.e., $\mathsf{if}(A_*, B_\alpha, C_\alpha)$ denotes the value of B_α if A_* holds and denotes the value of C_α if A_* does not hold).

If we fix the type of a variable x, say to α, then an expression of the form $(\Box\, x : \alpha \,.\, E)$ may be written simply as $(\Box\, x \,.\, E)$, where $\Box$ is λ, I, $\forall$, or $\exists$. If desired, all types can be removed from an expression by fixing the types of the variables occurring in the expression. We will write a formula of the form $(\Box\, x_1 : \alpha \,.\, \cdots\, \Box\, x_n : \alpha \,.\, A)$ simply as $(\Box\, x_1, \ldots, x_n : \alpha \,.\, A$, where $\Box$ is $\forall$ or $\exists$. Similarly, we will write a formula of the form $(\Box\, x_1 \,.\, \cdots\, \Box\, x_n \,.\, A)$ (where the types of $x_1, \ldots, x_n$ have been fixed) simply as $(\Box\, x_1, \ldots, x_n \,.\, A)$, where $\Box$ is $\forall$ or $\exists$.

The definitions above show that, although STT is formulated as a "function theory", STT is actually a form of higher-order predicate logic. Moreover, first-order logic, second-order logic, third-order logic, etc. can be "embedded" in STT. The precise statement of this result is:

Theorem 2 *Let T be any theory of nth-order logic for any $n \geq 1$. Then there is a theory T' of* STT *such that there is a faithful interpretation of T in T' (i.e., there is a translation Φ from the sentences of T to the sentences of T' such that, for all sentences A of T, $T \models A$ iff $T' \models \Phi(A)$).*

Because STT is equipped with full higher-order quantification and definite description, most mathematical notions can be directly and naturally expressed in STT, especially if types are suppressed. We will give five simple examples, three in this section and two in the next section, that illustrate how "higher-order" concepts can be expressed in STT. None of these examples can be directly expressed in first-order logic.

For the first example, let $\mathsf{equiv\text{-}rel}$ be the expression

$$\begin{array}{l} \lambda\, p : (\iota \rightarrow (\iota \rightarrow *)) \,. \\ \quad \forall\, x : \iota \,.\, p(x)(x) \wedge \\ \quad \forall\, x, y : \iota \,.\, p(x)(y) \Rightarrow p(y)(x) \wedge \\ \quad \forall\, x, y, z : \iota \,.\, (p(x)(y) \wedge p(y)(z)) \Rightarrow p(x)(z). \end{array}$$

Then $\mathsf{equiv\text{-}rel}(r)$ asserts that a binary relation r represented (in curryed form[4]) as a function of type $(\iota \rightarrow (\iota \rightarrow *))$ is an equivalence relation.

For the second example, let $\mathsf{compose}$ be the expression

$$\lambda\, f : (\iota \rightarrow \iota)\ .\ \lambda\, g : (\iota \rightarrow \iota)\ .\ \lambda\, x : \iota\ .\ f(g(x)).$$

If f, g are expressions of type $(\iota \rightarrow \iota)$, then $\mathsf{compose}(f)(g)$ is an expression that denotes the composition of f and g as functions.

For the third example, let $\mathsf{inv\text{-}image}$ be the expression

$$\lambda\, f : (\iota \rightarrow \iota)\ .\ \lambda\, s : (\iota \rightarrow *)\ .\ \mathrm{I}\, s' : (\iota \rightarrow *)\ .\ \forall\, x : \iota\ .\ s'(x) \Leftrightarrow s(f(x)).$$

If f is an expression of type $(\iota \rightarrow \iota)$ representing a function and s is an expression of type $(\iota \rightarrow *)$ representing a set, then $\mathsf{inv\text{-}image}(f)(s)$ is an expression of type $(\iota \rightarrow *)$ that represents the inverse image of s under f.

Virtue 3 STT *is a highly expressive logic.*

In fact, nearly all theorems of mathematics can be straightforwardly expressed in STT.

Set theory can be formalized as a theory of first-order logic, e.g., as Zermelo-Fraenkel set theory (ZF) or as von-Neumann-Bernays-Gödel set theory (NBG). These first-order theories are extremely expressive but possess a very complex semantics. One might be tempted to argue that these theories show that first-order logic itself is extremely expressive. They show something different—that ZF and NBG are highly expressive foundations for mathematics. First-order logic, unlike simple type theory, is not a highly expressive foundation for mathematics. Simple type theory is not nearly as expressive as set theory, but its much simpler semantics makes it a far more suitable general-purpose logic than set theory.

Some readers might be wondering whether definite description is actually a necessary part of STT. Before we discuss this issue we will need to take a closer look at the nature of expressivity. The expressivity of a logic has two levels. The *theoretical expressivity* of a logic is the measure of what ideas can be expressed in the logic without regard to how the ideas are expressed. The *practical expressivity* of a logic is the measure of how readily ideas can be expressed in the logic. For example, first-order logic with predicate symbols

[4] The *curryed form* of a function $f : A \times B \rightarrow C$ is the function $f' : A \rightarrow (B \rightarrow C)$ such that, for all $x \in A$ and $y \in B$, $f(x, y) = f'(x)(y)$. The process of "currying" a function is named after the logician H. Curry, the founder of combinatory logic.

but no function symbols has the same theoretical expressivity as standard first-order logic (with both predicate and function symbols) but has much lower practical expressivity than standard first-order logic.

The use of definite description in expressions of STT can be eliminated according to the scheme Russell presented in his famous and highly influential paper "On Denoting" [51]. Thus definite description is not a necessary theoretical component of STT: for every formula employing definite description there is an equivalent definite-description-free formula. This latter formula will often be much more verbose than the former. If definite description were not part of STT, the semantics of STT could be simplified because error elements would no longer be needed. However, without definite description it would not be possible to directly express the many mathematical concepts that are defined in informal mathematics using the form "the x that satisfies property P" such as the notion of the limit of a function (which is illustrated in the next section). In summary, removing definite description from STT would not affect the theoretical expressivity of STT but would significantly diminish STT's practical expressivity.

5 Categoricity

Theories are used to specify structures. Some theories (e.g., a theory of groups) specify collections of structures (e.g., the collection of all groups). Other theories (e.g., the theory of a complete ordered field) specify a single structure (e.g., the ordered field of the real numbers). In other words, a theory specifies the collection of models that belong to the theory. A theory is *categorical* if it has exactly one model up to isomorphism. Categorical theories are very desirable in applications in which a theory is used as a "model" of a specific complex system.

In his 1889 booklet [47], G. Peano presented a theory of natural number arithmetic, commonly called *Peano Arithmetic*. The theory consists of Peano's famous five axioms for the natural numbers plus four axioms for equality. Independently of Peano, R. Dedekind developed in [14] a theory of natural number arithmetic very similar to Peano Arithmetic. He proved, in effect, that all structures satisfying Peano's axioms are isomorphic to the standard structure $(\mathbf{N}, 0, s)$ of the natural numbers, i.e., that Peano Arithmetic is categorical.

Let $\mathsf{PA} = (L, \Gamma)$ be the theory of STT where $L = (\{0, s\}, \tau)$, $\Gamma = \{A_1, A_2, A_3\}$,

1. $\tau(0) = \iota$,

2. $\tau(s) = \iota \to \iota$,

3. A_1 is $\forall\, x : \iota \,.\, s(x) \neq 0$,

4. A_2 is $\forall\, x, y : \iota \,.\, s(x) = s(y) \Rightarrow x = y$, and

5. A_3 is $\forall\, p : (\iota \to *) \,.\, (p(0) \wedge (\forall\, x : \iota \,.\, p(x) \Rightarrow p(s(x)))) \Rightarrow \forall\, x : \iota \,.\, p(x)$.

PA is a very direct formalization of Peano Arithmetic. The type ι denotes the set $\mathbf{N} = \{0, 1, 2, \ldots\}$ of the natural numbers, 0 denotes the first natural number, and s denotes the successor function. The five clauses of the definition of PA correspond to Peano's five axioms for the natural numbers. (A_1 says 0 is not a successor, A_2 says s is injective, and A_3 expresses the induction principle.) The other basic operations of natural number arithmetic, such as $+$, $\cdot$, and $<$, can be defined in PA.

Like Peano Arithmetic itself, PA is categorical. Its unique standard model (up to isomorphism) is $(\mathcal{D}, I, e)$ where $D_\iota = \{0, 1, 2, \ldots\}$, $I(0) = 0$, $I(s) =$ the successor function on D_ι, and e is any function that maps each type α to a value in D_α.

Peano Arithmetic cannot be directly formalized in first-order logic because the induction principle involves quantification over predicates, which is not directly expressible in first-order logic. There is, however, a standard first-order formalization $\mathsf{PA}' = (L', \Gamma')$ of Peano Arithmetic where L' is the first-order language containing an individual constant 0, a unary function symbol s, and two binary function symbols $+$ and $\cdot$ and where Γ is the following set of sentences of L':

1. $\forall\, x \,.\, s(x) \neq 0$.

2. $\forall\, x, y \,.\, s(x) = s(y) \Rightarrow x = y$.

3. $\forall\, x \,.\, x + 0 = x$.

4. $\forall\, x, y \,.\, x + s(y) = s(x + y)$.

5. $\forall\, x \,.\, x \cdot 0 = 0$.

6. $\forall\, x, y \,.\, x \cdot s(y) = (x \cdot y) + x$.

7. Each sentence A that is a universal closure of a formula of the form

$$(B[0] \wedge (\forall\, x \,.\, B[x] \Rightarrow B[s(x)])) \Rightarrow \forall\, x \,.\, B[x]$$

   where $B[x]$ is a formula of L'.

Clause 7 is an infinite collection of formulas called the *induction schema*. It is only an approximation of the induction principle. In fact, the induction schema includes just a countably infinite number of instances of the induction principle (one for each equivalence class of formulas related by logical equivalence), while the induction principle has a continuum number of instances (one for each property of the natural numbers).

PA' is not categorical. Since PA' does not contain all instances of the induction principle, Dedekind's proof of the categoricity of Peano Arithmetic fails. Moreover, PA' has "nonstandard" models containing infinite natural numbers by the compactness theorem of first-order logic.

The failure to achieve categoricity for a theory of natural number arithmetic in first-order logic is not an aberration. Rather it is an instance of a fundamental weakness of first-order logic that is a simple consequence of the compactness theorem of first-order logic:

Theorem 3 *Any first-order theory that has an infinite model has infinitely many (infinite) nonisomorphic models.*

Thus, a first-order theory that is intended to specify a single infinite structure cannot be categorical.

Let us consider a second example. It is well known that there is exactly one *complete ordered field* up to isomorphism, namely, the standard structure

$$(\mathbf{R}, +, 0, -, \cdot, 1, {}^{-1}, \mathsf{pos})$$

of the real numbers where pos denotes the set of positive real numbers.[5]

Let $\mathsf{COF} = (L, \Gamma)$ be the theory of STT such that:

- $L = (\{+, 0, -, \cdot, 1, {}^{-1}, \mathsf{pos}, <, \leq, \mathsf{ub}, \mathsf{lub}\}, \tau)$ where τ is defined by:

Constant c	**Type** $\tau(c)$
0, 1	ι
$-$, $^{-1}$	$\iota \rightarrow \iota$
pos	$\iota \rightarrow *$
$+$, $\cdot$	$\iota \rightarrow (\iota \rightarrow \iota)$
$<$, $\leq$	$\iota \rightarrow (\iota \rightarrow *)$
ub, lub	$\iota \rightarrow ((\iota \rightarrow *) \rightarrow *)$

[5] The first categorical axiomatization of the real numbers is generally considered to be the theory of a *maximal ordered Archimedean field* presented by D. Hilbert in 1900 [32].

- Γ is the set of the 18 sentences given below. We assume that the variables x, y, z are of type ι and the variable s is of type $(\iota \rightarrow *)$.

  1. $\forall\, x, y, z \,.\, (x + y) + z = x + (y + z)$.
  2. $\forall\, x, y \,.\, x + y = y + x$.
  3. $\forall\, x \,.\, x + 0 = x$.
  4. $\forall\, x \,.\, x + (-x) = 0$.
  5. $\forall\, x, y, z \,.\, (x \cdot y) \cdot z = x \cdot (y \cdot z)$.
  6. $\forall\, x, y \,.\, x \cdot y = y \cdot x$.
  7. $\forall\, x \,.\, x \cdot 1 = x$.
  8. $\forall\, x \,.\, x \neq 0 \Rightarrow x \cdot x^{-1} = 1$.
  9. $0 \neq 1$.
  10. $\forall\, x, y, z \,.\, x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.
  11. $\forall\, x \,.\, (x = 0 \wedge \neg\mathsf{pos}(x) \wedge \neg\mathsf{pos}(-x)) \vee$
      $(x \neq 0 \wedge \mathsf{pos}(x) \wedge \neg\mathsf{pos}(-x)) \vee$
      $(x \neq 0 \wedge \neg\mathsf{pos}(x) \wedge \mathsf{pos}(-x))$.
  12. $\forall\, x, y \,.\, (\mathsf{pos}(x) \wedge \mathsf{pos}(y)) \Rightarrow \mathsf{pos}(x + y)$.
  13. $\forall\, x, y \,.\, (\mathsf{pos}(x) \wedge \mathsf{pos}(y)) \Rightarrow \mathsf{pos}(x \cdot y)$.
  14. $\forall\, x, y \,.\, x < y \Leftrightarrow \mathsf{pos}(y - x)$.
  15. $\forall\, x, y \,.\, x \leq y \Leftrightarrow (x < y \vee x = y)$.
  16. $\forall\, x, s \,.\, x \;\mathsf{ub}\; s \Leftrightarrow (\forall\, y \,.\, s(y) \Rightarrow y \leq x)$.
  17. $\forall\, x, s \,.\, x \;\mathsf{lub}\; s \Leftrightarrow (x \;\mathsf{ub}\; s \wedge (\forall\, y \,.\, y \;\mathsf{ub}\; s \Rightarrow x \leq y))$.
  18. $\forall\, s \,.\, ((\exists\, x \,.\, s(x)) \wedge (\exists\, x \,.\, x \;\mathsf{ub}\; s)) \Rightarrow \exists\, x \,.\, x \;\mathsf{lub}\; s$.

Notes:

1. As a result of fixing the types for the variables x, y, z, s, the axioms of COF are free of types and thus look just like the axioms one might see in any mathematics textbook.

2. We write the additive and multiplicative inverses of x as $-x$ and x^{-1} instead of as $-(x)$ and $^{-1}(x)$, respectively.

3. $+$ and $*$ are formalized by constants of type $(\iota \rightarrow (\iota \rightarrow \iota))$ representing curryed functions. However, we write the application of $+$ and $*$ using infix notation, e.g., we write $x + y$ instead of $+(x)(y)$. $<$ and $\leq$ are handled in a similar way.

4. pos is a predicate that represents the set of positive real numbers.

5. $\mathsf{ub}(x)(s)$ and $\mathsf{lub}(x)(s)$ say that x is an upper bound of s and x is the least upper bound of s, respectively. We write $\mathsf{ub}(x)(s)$ and $\mathsf{lub}(x)(s)$ as $x \mathrel{\mathsf{ub}} s$ and $x \mathrel{\mathsf{lub}} s$, respectively, using infix notation.

6. Sentence 18 expresses the *completeness principle* of the real numbers, i.e., that every set of real numbers that is nonempty and has an upper bound has a least upper bound.

COF is a direct formalization of the theory of a complete ordered field. COF is categorical; its unique standard model (up to isomorphism) is $(\mathcal{D}, I, e)$ where $D_\iota = \mathbf{R}$, the set of real numbers, I assigns $+$, 0, $-$, $\cdot$, 1, $^{-1}$, pos, $<$, $\leq$, ub, lub their usual meanings, and e is any function that maps each type α to a value in D_α.

Like Peano Arithmetic, the theory of a complete ordered field is a theory of a single fundamental infinite structure that cannot be directly formalized in first-order logic as a categorical theory since the completeness principle involves quantification over predicates.

Virtue 4 STT *admits categorical theories of infinite structures.*

It is worthwhile to note that PA and COF are both formalized using only a small portion of the machinery of STT. In fact, both Peano Arithmetic and the theory of a complete ordered field can be formalized in *second-order logic* as categorical theories.

We will conclude this section by giving the fourth and fifth examples that illustrate the expressivity of STT.

For the fourth example, let abs be the expression

$$\lambda\, r : \iota \,.\, \mathsf{if}(0 \leq r, r, -r)$$

in COF. abs denotes the absolute value function on the real numbers.

For the fifth example, let lim be the expression

$$\begin{array}{l} \lambda\, f : (\iota \rightarrow \iota) \,.\, \lambda\, a : \iota \,. \\ \quad (\mathrm{I}\, l : \iota \,.\, (\forall\, \epsilon : \iota \,.\, 0 < \epsilon \Rightarrow \\ \quad\quad (\exists\, \delta : \iota \,.\, 0 < \delta \wedge \\ \quad\quad\quad (\forall\, x : \iota \,.\, (\mathsf{abs}(x - a) < \delta \wedge x \neq a) \Rightarrow \mathsf{abs}(f(x) - l) < \epsilon)))) \end{array}$$

in COF. Suppose f is an expression of type of $(\iota \rightarrow \iota)$ and a is an expression of type ι in COF. Then $\mathsf{lim}(f)(a)$ denotes the limit of f at a if f approaches a limit at a; otherwise $\mathsf{lim}(f)(a)$ denotes the canonical error value of type ι. Notice that the use of definite description in this example enables the standard definition of a limit of a function to be directly formalized.

6 Provability

Up to this point we have said nothing about formal proof. In this section we investigate provability in STT. We begin by giving some definitions.

A *(Hilbert-style) proof system* for STT consists of a finite set of axiom schemas and rules of inference. Let $\mathbf{P}$ be a proof system for STT and $T = (L, \Gamma)$ be a theory of STT. A finite sequence of formulas of L is a *proof* of a formula A from T in $\mathbf{P}$ if A is the last member of the sequence and every member of the sequence is an instance of one of the axiom schemas of $\mathbf{P}$, a member of Γ, or is inferred from previous members by a rule of inference of $\mathbf{P}$.

Let $T \vdash_{\mathbf{P}} A$ mean there is a proof of A from T in $\mathbf{P}$. $\mathbf{P}$ is *sound* if, for every theory $T = (L, \Gamma)$ and formula A of L,

$$T \vdash_{\mathbf{P}} A \ \text{ implies } \ T \models A.$$

$\mathbf{P}$ is *complete* if, for every theory $T = (L, \Gamma)$ and formula A of L,

$$T \models A \ \text{ implies } \ T \vdash_{\mathbf{P}} A.$$

It is well known that there is no sound and complete proof system for second-order logic, and hence, it is not surprising that there is no sound and complete proof system for STT. On the surface, this may appear to be a weakness of STT, but indeed it is a sign of its strength. The incompleteness of STT is an immediate consequence of Gödel's Incompleteness Theorem.

Theorem 4 (Incompleteness) *There is no sound and complete proof system for* STT.

Proof Suppose $\mathbf{P}$ is a sound and complete proof system for STT. By the soundness of $\mathbf{P}$ and Gödel's Incompleteness Theorem, there is a sentence A such that (1) $M \models A$, where M is the unique standard model for PA (up to isomorphism), and (2) $\mathsf{PA} \not\vdash_{\mathbf{P}} A$. By the completeness of $\mathbf{P}$, (2) implies $\mathsf{PA} \not\models A$ and hence $M \not\models A$ since M is the only standard model of PA, which contradicts (1). $\Box$

We will now present a very simple and elegant proof system for L called **A** which is adapted for STT from a proof system devised by Andrews [3]. Define $B_\beta[(x:\alpha) \mapsto A_\alpha]$ to be the result of simultaneously replacing each free occurrence of $(x:\alpha)$ in B_β by an occurrence of A_α. Let $(\exists\,!\,x:\alpha\,.\,A)$ mean

$$\exists\, x:\alpha\,.\,(A \wedge (\forall\, y:\alpha\,.\,A[(x:\alpha)\mapsto(y:\alpha)] \Rightarrow y=x))$$

where $(y:\alpha)$ does not occur in A. This formula asserts there exists a unique value x of type α that satisfies A.

A consists of the following six axiom schemas and single rule of inference:

A1 (Truth Values)

$$\forall\, f:(* \rightarrow *)\,.\,(f(\mathsf{T}) \wedge f(\mathsf{F})) \Leftrightarrow (\forall\, x:*\,.\,f(x)).$$

A2 (Leibniz' Law[6])

$$\forall\, x,y:\alpha\,.\,(x=y) \Rightarrow (\forall\, p:(\alpha \rightarrow *)\,.\,p(x) \Leftrightarrow p(y)).$$

A3 (Extensionality)

$$\forall\, f,g:(\alpha \rightarrow \beta)\,.\,(f=g) \Leftrightarrow (\forall\, x:\alpha\,.\,f(x)=g(x)).$$

A4 (Beta-Reduction)

$$(\lambda\, x:\alpha\,.\,B_\beta)(A_\alpha) = B_\beta[(x:\alpha)\mapsto A_\alpha]$$

provided A_α is free for $(x:\alpha)$ in B_β.[7]

A5 (Proper Definite Description)

$$(\exists\,!\,x:\alpha\,.\,A_*) \Rightarrow A_*[(x:\alpha)\mapsto(\mathrm{I}\,x:\alpha\,.\,A_*)]$$

provided $(\mathrm{I}\,x:\alpha\,.\,A_*)$ is free for $(x:\alpha)$ in A_*.[7]

[6]Leibniz' Law says that if two things are equal they satisfy exactly the same properties.

[7]A_α is *free* for $(x:\alpha)$ in B_β if no free occurrence of $(x:\alpha)$ in B_β is within a subexpression of B_β of the form $(\lambda\, y:\gamma\,.\,C)$ or $(\mathrm{I}\, y:\gamma\,.\,C)$ where $(y:\gamma)$ is free in A_α. That is, A_α is free for $(x:\alpha)$ in B_β if none of the free variables of A_α are "captured" by λ or I when the free occurrences of $(x:\alpha)$ in B_β are replaced by A_α.

A6 (Improper Definite Description)

$$\neg(\exists\,!\,x : \alpha \,.\, A_*) \Rightarrow (\mathrm{I}\,x : \alpha \,.\, A_*) = \bot_\alpha.$$

R (Equality Substitution) From $A_\alpha = B_\alpha$ and C_* infer the result of replacing one occurrence of A_α in C_* by an occurrence of B_α, provided that the occurrence of A_α in C_* is not a variable $(x : \alpha)$ immediately preceded by λ or I.

Notice that **A** does not include a comprehension axiom schema for defining functions from expressions. It is unnecessary because a function can be defined directly from an expression via function abstraction. Moreover, the comprehensive axiom schema is provable in **A** (see the proof of Theorem 5243 in [4] for details).

Theorem 5 **A** *is sound.*

Proof Each instance of the axiom schemas A1-A6 is valid in every standard model, and rule R preserves validity in every standard model. For details, see the proof of Theorem 5402 in [4]. □

Theorem 6 **A** *is not complete.*

Proof Corollary of Theorem 4. □

Since **A** is incomplete, it is not obvious whether the basic theorems of simple type theory can be proven in **A**. Does **A** have sufficient provability power to be useful? The answer is yes indeed—**A** has enough provability power to serve as a foundation for mathematics.

Let **A+I** be **A** plus an additional axiom that says that the domain of individuals is infinite.

Theorem 7 *The consistency of* **A+I** *implies the consistency of bounded Zermelo set theory and conversely.*

Bounded Zermelo set theory is Zermelo set theory (i.e., Zermelo-Fraenkel set theory without the replacement axiom) with a version of the separation axiom in which the quantifiers are bounded by sets. S. Mac Lane advocated bounded Zermelo set theory as an adequate foundation for mathematics [41], and as a result, bounded Zermelo set theory is commonly known as *Mac Lane set theory*.

A standard measure for provability power is consistency strength. Theorem 7 says that **A+I** and bounded Zermelo set theory have the same consistency strength and thus the same provability power. Therefore, if bounded Zermelo set theory is an adequate foundation for mathematics, so is **A** plus an axiom of infinity.

A variant of Theorem 7 was first proved by R. Jensen [35]; a more detailed proof is found in A. Mathias's paper [43].

> **Virtue 5** *There is a proof system for* STT *that is simple, elegant, and powerful.*

In the next section, we will show that there is a sense in which **A** is actually complete—which should expel any queasiness that one has about how well **A** captures the semantics of STT.

7 General Models

When Henkin was a graduate student at Princeton, he investigated the structure of nameable values (see section 3.2) in standard models of Church's type theory [30]. His investigation led to two extraordinary discoveries.

First, he discovered that the proof system for Church's type theory is actually sound and complete if the semantics is generalized by substituting the notion of a "general model" for the notion of a standard model [28]. Recall that the incompleteness of Church's type theory with respect to the ordinary semantics based on standard models follows from Gödel's incompleteness theorem (see Theorem 4).

Second, he found that the method employed in his proof of the generalized completeness of Church's type theory could be used to get a new proof of the completeness of first-order logic (which was first proved by Gödel) [27]. This proof is now one of the hallmarks of first-order model theory.

A *general structure* for a language $L = (\mathcal{C}, \tau)$ of STT is a triple $M = (\mathcal{D}, I, e)$ where:

1. $\mathcal{D} = \{D_\alpha : \alpha \in \mathcal{T}\}$ is a set of nonempty domains (sets).

2. $D_* = \{\text{T}, \text{F}\}$.

3. For $\alpha, \beta \in \mathcal{T}$, $D_{\alpha \to \beta}$ is *some* nonempty set of total functions from D_α to D_β.

4. I maps each $c \in \mathcal{C}$ to a member of $D_{\tau(c)}$.

5. e maps each $\alpha \in \mathcal{T}$ to a member of D_α.

M is a *general model* for L if there is a binary function V^M that satisfies the same conditions as the valuation function for a standard model. A general model is thus the same as a standard model except that the function domains of the model may not be "fully inhabited". Hence every standard model for L is also a general model for L. Let us say that M is a *nonstandard model* for L if it is a general model, but not a standard model, for L.

Let $T = (L, \Gamma)$ be a theory of STT and A be a formula of L. A is a *semantic consequence* of a set Σ of sentences of L *in the general sense*, written $\Sigma \models_g A$, if $M \models A$ for every general model M for L such that $M \models B$ for all $B \in \Sigma$. A is a *semantic consequence* of T *in the general sense*, written $T \models_g A$, if $\Gamma \models_g A$. A *general model* of T is a general model M for L such that $M \models B$ for all $B \in \Gamma$.

Let $\mathbf{P}$ be a proof system for STT. $\mathbf{P}$ is *sound in the general sense* if, for every theory $T = (L, \Gamma)$ and formula A of L,

$$T \vdash_{\mathbf{P}} A \text{ implies } T \models_g A.$$

$\mathbf{P}$ is *complete in the general sense* if, for every theory $T = (L, \Gamma)$ and formula A of L,

$$T \models_g A \text{ implies } T \vdash_{\mathbf{P}} A.$$

Theorem 8 **A** *is sound in the general sense.*

Proof This is a generalization of Theorem 5; see the proof of Theorem 5402 in [4]. □

A theory T is *consistent* if there is no proof of F from T in **A**. For a language $L = (\mathcal{C}, \tau)$, the cardinality of L, written $|L|$, is $|\mathcal{C}| + \aleph_0$. A general model $(\{D_\alpha : \alpha \in \mathcal{T}\}, I, e)$ for L is *frugal* if $|D_\alpha| \leq |L|$ for all $\alpha \in \mathcal{T}$.

Theorem 9 (Henkin's Theorem) *Every consistent theory of* STT *has a frugal general model.*

Proof See the proof of Theorem 5501 in [4]. □

Corollary 10 *There is a general model of* $\mathsf{COF} = (L, \Gamma)$ *such that* $|D_\iota| = \aleph_0$ *(provided* COF *is consistent).*

Thus by Henkin's Theorem, there exist nonstandard models of COF, the theory of the real numbers, in which D_ι contains only a countable number of real numbers. Such models result because $D_{\iota\to *}$ does not contain every predicate, and as a result, the completeness principle says that only the nonempty, bounded subsets of D_ι corresponding to members of $D_{\iota\to *}$ have least upper bounds. Note that, since COF has a nonstandard model, COF is not categorical with respect to general models.

Theorem 11 (Henkin's Completeness Theorem) **A** *is complete in the general sense.*

Proof Follows from Henkin's Theorem by an easy argument; see [4]. □

Analogs of the basic theorems of first-order model theory—such as the compactness theorem and the Löwenheim-Skolem theorem—can be derived from Theorems 9 and 11. By the compactness theorem for general models, PA has nonstandard models and thus is not categorical with respect to general models.

A theory T of STT can be encoded as a theory T' in many-sorted first-order logic as follows. Each type α of STT is represented by a sort s_α of T'. Each variable and constant of type α of T is represented by a variable and constant, respectively, of sort s_α of T'. For each function type $(\alpha \to \beta)$ of STT there is a function symbol of T' that represents the application of functions of type $(\alpha \to \beta)$ to arguments of type α. For each function abstraction f of T there is a function symbol that represents f. T' includes a set of extensionality and comprehension axioms. As a result of this encoding, each general model of T is represented by a model of T'. The ideas behind this encoding are found in [31, 36]. For a similar encoding of second-order logic in many-sorted first-order logic, see either [15] or [53].

Since any theory of many-sorted first-order logic can be encoded in ordinary (unisorted) first-order logic in a standard way, the encoding of T in many-sorted first-order logic gives an immediate proof of Henkin's Theorem as well as the STT analogs of the compactness theorem and the Löwenheim-Skolem theorem. The encoding also shows that a theory of STT with the general models semantics is just a first-order theory presented in a more natural and convenient form.

Nonstandard models in STT have the same importance to the model theory of STT as nonstandard first-order models have to first-order model theory. Each nonstandard model of a theory exhibits a way of interpreting what the theory means. Some nonstandard models (like the countable model of COF mentioned above) expose semantic defects (of the theory or model).

Others (like a model of COF which contains infinitesimals) provide foundations for different ways of analyzing the standard model(s) of the theory (as is done in nonstandard analysis).

In STT, the distinction between standard and nonstandard models is absolutely clear: in a standard model every function domain is fully inhabited, while in a nonstandard model some function domain is only partially inhabited. However, in first-order logic, there is no formal distinction between the standard and nonstandard models of a theory. In fact, for the first-order theories of the natural numbers and the real numbers, this distinction is meaningless without a "higher-order" perspective.

For example, consider a general model M of PA that includes nonstandard (infinite) natural numbers. We know such models exist by the generalized compactness theorem of STT. Is M standard or nonstandard? Since M is a general model of PA, the induction principle A, which has the form $(\forall\, p : (\iota \rightarrow *) \;.\; B)$, is valid in M. If d is the predicate that is true for all the standard (finite) natural numbers but false for all nonstandard natural numbers, then $V^M_{\varphi[(p:(\iota\rightarrow *))\mapsto d]}(B) = \mathrm{F}$ for any variable assignment φ into M, which contradicts the validity of A in M. Therefore, $d \notin D_{\iota\rightarrow *}$, and thus M is nonstandard.

Now consider a model of the first-order theory PA' that includes nonstandard natural numbers. We know such models exist by the compactness theorem of first-order logic. Is M standard or nonstandard? The theory PA' alone does not provide us with any way of distinguishing standard models from nonstandard models. The distinction actually relies on the following two facts:

1. PA' is intended to be a first-order approximation of Peano Arithmetic [38].

2. The usual model of the natural numbers is the only model of Peano Arithmetic (up to isomorphism).

Therefore, M is a nonstandard model of PA' since it is not the unique model of Peano Arithmetic (up to isomorphism). Notice that to distinguish between standard and nonstandard models of PA' we had to appeal to Peano Arithmetic, a "higher-order" theory.

These results about general models do not diminish the importance of the categoricity results mentioned earlier. In STT—but not in first-order logic—it is possible to formulate a theory T that fully specifies an infinite structure S like the natural numbers or the real numbers. The theory T is categorical in the sense that it has one standard model (up to isomorphism)

corresponding to S, but it is also noncategorical in the sense that it has (infinitely many) nonstandard models. The standard model is the intended subject of T. The nonstandard models can either be ignored or used to gain insight into the nature of the relationship between T and S.

Virtue 6 *Henkin's general models semantics enables the techniques of first-order model theory to be applied to* STT *and illuminates the distinction between standard and nonstandard models.*

We have seen that STT has two semantics, which are both useful. With the standard semantics, STT is a much more expressive logic than first-order logic which admits categorical theories of structures such as the natural numbers and the real numbers. With the general semantics, STT is a disguised version of first-order logic which is very convenient for expressing concepts concerning higher-order functions. With both semantics together, STT provides an excellent framework for studying the models of a theory, particularly the relationship between standard and nonstandard models.

8 Practicality

Although mathematical ideas can be formalized much more directly and succinctly in STT than in first-order logic, formalizing real mathematics in STT would still be quite burdensome. However, by extending the syntax and semantics of STT in certain ways, STT can be made into an effective logic for actual use. These extensions do not change STT in any fundamental way; they just make STT more convenient to employ.

Many practical ways of extending a simple type theory like STT have been proposed, beginning with [55]. We will briefly present in this section the most important examples.

8.1 Many sortedness

In STT, ι, the type of individuals, is the only type for basic values. As a result, all the basic values of an STT theory must be handled together. For example, in an STT theory of graphs, a variable of type ι would simultaneously range over both nodes and edges. It would be much more practical if STT were "many-sorted" with several types for basic values. Then, for example, in a theory of graphs, there could be one type for nodes and another for edges. A many-sorted version of STT can be achieved by simply

allowing a language L to include a nonempty set $\mathcal{B}_L$ of *base types*, each of which denotes a domain of individuals. Then a *type* of L would be defined inductively by the following formation rules:

1. *Base type*: If $\alpha \in \mathcal{B}_L$, then α is a type of L.

2. *Type of truth values*: $*$ is a type of L.

3. *Function type*: If α and β are types of L, then $(\alpha \rightarrow \beta)$ is a type of L.

The rest of the definitions for the syntax and semantics of STT would remain essentially the same.

8.2 Tuples, lists, sets, etc.

Machinery for working with basic mathematical objects like tuples, lists, and sets can be built into STT by adding new type constructors and new expression constructors or built-in constants. For example, the machinery for ordered pairs (and hence tuples) would be (1) a product type constructor $\times$ such that the type $\alpha \times \beta$ denotes the Cartesian product $D_\alpha \times D_\beta$, (2) an expression constructor pair such that $\mathsf{pair}(A_\alpha, B_\alpha)$ is an expression of type $\alpha \times \beta$ that denotes the ordered pair of A_α and B_α, and (3) the expression constructors first and second that form expressions that select the first and second components, respectively, of a member of $D_\alpha \times D_\beta$. Tuples could then be defined as iterated ordered pairs with this machinery. See the logic BESTT [18] for an example of a version of Church's type theory with tuples, lists, and sets.

8.3 Indefinite description

In comparison to definite description that builds an expression which denotes *the* value that satisfies a property P, indefinite description builds an expression that denotes *some* value that satisfies a property P. Indefinite description complements definite description and is useful for specifying objects with underspecified components. To add indefinite description to STT we need to (1) include the formation rule

> *Indefinite description*: If $x \in \mathcal{V}$, $\alpha \in \mathcal{T}$, and A is an expression of type $*$, then $(\epsilon\, x : \alpha \,.\, A)$ is an expression of type α.[8]

[8] The indefinite description operator is very often called the *Hilbert epsilon operator* because it is the chief operator in Hilbert's *epsilon calculus* [2], where it is denoted by epsilon (ϵ) and used to define universal and existential quantification.

in the definition of an expression, (2) modify the definition of the valuation function for a standard model to handle indefinite descriptions (see [24] for details), and (3) add the two axiom schemas

A7 (Proper Indefinite Description)

$$(\exists\, x : \alpha \;.\; A_*) \Rightarrow A_*[(x : \alpha) \mapsto (\epsilon\, x : \alpha \;.\; A_*)]$$

provided $(\epsilon\, x : \alpha \;.\; A_*)$ is free for $(x : \alpha)$ in A_*.

A8 (Improper Indefinite Description)

$$\neg(\exists\, x : \alpha \;.\; A_*) \Rightarrow (\epsilon\, x : \alpha \;.\; A_*) = \bot_\alpha.$$

to the proof system **A**.

Indefinite description cannot be defined in terms of definite description. In fact, STT plus indefinite description is strictly stronger than STT alone. This is because the presence of indefinite description implies the existence of a choice function for each predicate type. For example,

$$\lambda\, s : (\alpha \to *) \;.\; \epsilon\, x : \alpha \;.\; s(x)$$

is a choice function for the type $(\alpha \to *)$. Using this choice function and the A7 axiom schema, the axiom of choice for the type $(\alpha \to *)$ can be proved in **A**. (The axiom of choice is not provable in **A** without indefinite description.) Thus, in contrast to definite description, adding indefinite description to simple type theory increases both its theoretical and practical expressivity.

8.4 Undefinedness

A severe weakness of STT—as well as of first-order logic—is that expressions are assumed to be *defined* (i.e., to denote some value) and functions are assumed to be *total* (i.e., to be defined on all arguments). However, undefined expressions—resulting from undefined function applications (like 17/0) and improper definite descriptions (like $\lim_{x \to 0} \sin \frac{1}{x}$)—are commonplace in mathematical practice. With little difficulty, the semantics of STT can be modified to admit *undefined expressions* and *partial functions*. As a result, improper definite and indefinite descriptions would be undefined and error values would no longer be necessary. This extension of the semantics can be achieved by either preserving the classical two-valued nature of simple type theory [16, 19] or by extending simple type theory to a three-valued logic [37].

Mathematics can be formalized more concisely and handled more naturally in a logic in which undefined expressions and partial functions can be directly represented than in a standard logic in which all expressions are assumed to be denoting and all functions are assumed to be total. For support of this statement, see the examples from calculus in [19] and the many examples from analysis in the theory library of the IMPS theorem proving system [20].

8.5 Polymorphism

An operator is *polymorphic* if it can be applied to expressions of more than one type. For instance, the equality operator (=) in STT is polymorphic because it can be applied to any two expressions of the same type. One of the greatest shortcomings of STT is that it has no mechanism for introducing new polymorphic operators. For example, recall the constant compose defined in section 4 for expressing the composition of two functions of type $(\iota \rightarrow \iota)$. Let us rename compose to $\mathsf{compose}_{\iota\rightarrow\iota,\iota\rightarrow\iota}$ to show what kind of expressions it can be applied to. If we wanted to compose two functions of other types we would need to define in exactly the same way other composition constants of the form $\mathsf{compose}_{\beta\rightarrow\gamma,\alpha\rightarrow\beta}$. Since there are an infinite number of function types in STT, there is potentially an infinite number of distinct composition constants that might be needed. It would be far more efficient if we could define a single polymorphic composition operator.

There are several approaches that can be used to introduce polymorphic operators in simple type theory. First, an operator name (e.g., compose) can be overloaded so that it potentially denotes many different, but related, constants (e.g., $\mathsf{compose}_{\beta\rightarrow\gamma,\alpha\rightarrow\beta}$ for all $\alpha, \beta, \gamma \in \mathcal{T}$). When it (compose) is applied in a particular context, the context is used to determine which constant ($\mathsf{compose}_{\beta\rightarrow\gamma,\alpha\rightarrow\beta}$) it actually denotes.

Second, a polymorphic operator can be defined as a macro (e.g., compose) that can be applied to expressions of different types. An application of the macro (e.g., $\mathsf{compose}(F_{\beta\rightarrow\gamma})(G_{\alpha\rightarrow\beta})$) expands to an ordinary, nonpolymorphic expression (e.g.,

$$(\lambda\, f : (\beta \rightarrow \gamma) \,.\, \lambda\, g : (\alpha \rightarrow \beta) \,.\, \lambda\, x : \alpha \,.\, f(g(x)))(F_{\beta\rightarrow\gamma})(G_{\alpha\rightarrow\beta})\ ).$$

The IMPS theorem proving system uses polymorphic operators of this kind called *quasi-constructors* [20].

The third approach is to expand the type system to include type variables which can be instantiated with other types [25]. A polymorphic operator is simply an expression of a type that contains type variables. For example, a

constant $\mathsf{compose}$ that gives the composition of two functions of appropriate type could have the type

$$(\nu \rightarrow \xi) \rightarrow ((\mu \rightarrow \nu) \rightarrow (\mu \rightarrow \xi))$$

where μ, ν, ξ are type variables. If F is an expression of type $(\beta \rightarrow \gamma)$ and G is an expression of type $(\alpha \rightarrow \beta)$ where α, β, γ are variable-free types, then $\mathsf{compose}(F)(G)$ would be an expression of type $(\alpha \rightarrow \gamma)$ that denotes the composition of F and G. See the logic of the HOL theorem proving system [24] for details.

Polymorphism is a major research topic in the design of specification languages, programming languages, and constructive type theories. For other, more sophisticated approaches to polymorphism, see [6, 12, 23].

8.6 Subtypes

Suppose that $\mathsf{COF}_{\mathrm{ms}}$ is a theory of a complete ordered field formulated in many-sorted STT in which the base types $\mathbf{N}$ and $\mathbf{R}$ denote the natural numbers and real numbers, respectively. If $+$ is a constant of type $(\mathbf{R} \rightarrow (\mathbf{R} \rightarrow \mathbf{R}))$ and 2 is an expression of type $\mathbf{N}$, then $2 + 2$ (using infix notation for $+$) is not a well-formed expression due to an obvious type mismatch between $\mathbf{N}$ and $\mathbf{R}$. To avoid the type mismatch, the expression would have to be written as $J(2) + J(2)$ where J is an expression denoting the standard embedding of the natural numbers in the real numbers. If subtypes were allowed in many-sorted STT, this inconvenience would go away. $\mathbf{N}$ would be declared a subtype of $\mathbf{R}$, and then an expression of type $\mathbf{N}$ could be used wherever an expression of type $\mathbf{R}$ is prescribed.

For two types α and β, α is a *subtype* of β, written $\alpha \preceq \beta$, if the $D_\alpha \subseteq D_\beta$ (i.e., the denotation of α is a subset of the denotation of β). The relation $\alpha \preceq \beta$ can be either assumed as an axiom or defined by a predicate of type $(\beta \rightarrow *)$. For example, in $\mathsf{COF}_{\mathrm{ms}}$ we can define a subtype $\mathbf{CF} \preceq (\mathbf{R} \rightarrow \mathbf{R})$ of continuous functions by the predicate

$$\lambda\, f : (\mathbf{R} \rightarrow \mathbf{R}) \,.\, \forall\, x : \mathbf{R} \,.\, \mathsf{lim}(f)(x) = f(x).$$

If type variables are available, type constructors can be defined as parametric subtypes. For example, the product type constructor $\times$ mentioned above can be defined by the predicate

$$\lambda\, p : (\mu \rightarrow (\nu \rightarrow *)) \,.\, \exists\, a : \mu, b : \nu \,.\, p = (\lambda\, x : \mu, \lambda\, y : \nu \,.\, x = a \wedge y = b)$$

where μ and ν are type variables. Thus, if α and β are variable-free types, $D_{\alpha\times\beta} \subseteq D_{\alpha\to(\beta\to *)}$ and each $p \in D_{\alpha\times\beta}$ would be a function

$$f : D_\alpha \to (D_\beta \to \{\text{T}, \text{F}\})$$

for which there exists $a \in D_\alpha$ and $b \in D_\beta$ such that $f(x)(y) = \text{T}$ iff $\langle x, y\rangle = \langle a, b\rangle$.

Subtypes greatly increase the practical expressivity of simple type theory. There are several ways subtypes can be introduced into STT and other versions of simple type theory. See the logic of PVS [50] for an example of subtypes employed in a traditional version of Church's type theory, and see the logic of IMPS [17] for an example of subtypes used in a version of Church's type theory with undefinedness and partial functions.

8.7 Dependent types

Suppose $\mathbf{M}$ is a base type in $\mathsf{COF}_{\mathrm{ms}}$ that denotes the set of matrices over $\mathbf{R}$ and $\otimes$ is a constant of type $(\mathbf{M} \to (\mathbf{M} \to \mathbf{M}))$ that denotes the usual matrix multiplication function. $m \otimes m'$ is defined only if, for some positive integers a, b, c, m is an $a \times b$ matrix and m' is an $b \times c$ matrix. Hence $\otimes$ is a partial function. This suggests that $\otimes$ should be assigned the "parameterized" type $(\mathbf{M}_{a,b} \to (\mathbf{M}_{b,c} \to \mathbf{M}_{a,c}))$ where $\mathbf{M}_{x,y}$ denotes the set of $x \times y$ matrices over $\mathbf{R}$. Then, for all parameters a, b, c that are positive integers, $\otimes$ would be a total function of type $(\mathbf{M}_{a,b} \to (\mathbf{M}_{b,c} \to \mathbf{M}_{a,c}))$.

A type like $\mathbf{M}_{x,y}$ that depends on values (i.e., the positive integers x and y) is called a *dependent type*. As our example shows, dependent types arise quite naturally in mathematics and are useful. A key component of Martin-Löf type theory [42], dependent types have been extensively studied in the context of constructive type theory. Dependent types have received much less attention in nonconstructive type theory. Two noteworthy results in this direction are the definition of dependent types in the PVS logic using predicate subtypes [50], and B. Jacobs and T. Melham's embedding of dependent type theory in Church's type theory [34].

8.8 Implementations

Convenient versions of Church's type theory have been implemented in the computer theorem proving systems HOL [24], IMPS [20], Isabelle [46], ProofPower [40], PVS [45], and TPS [5]. Experience has shown that these implemented versions of Church's type theory are indeed effective for practical use. Hundreds of theories have been formulated and thousands of theorems have been proved using these logics in these theorem provers.

Virtue 7 *There are practical extensions of* STT *that can be effectively implemented.*

9 Conclusion

We have surveyed seven virtues of simple type theory. These virtues make simple type theory a very attractive general-purpose logic. One can argue that, as a logic for actual use—in science, engineering, and mathematics education and research—simple type theory is superior to first-order logic. It is much closer to mathematical practice than first-order logic, and as a result, mathematics expressed in simple type theory is more natural, more concise, and easier to work with. The greatest virtue of simple type theory is that it is a logic that is effective for *practice* as well as for *theory*.

We recommend that simple type theory be incorporated into introductory logic courses offered by mathematics departments by replacing the traditional two-logic sequence with a three-logic sequence—*propositional logic, first-order logic, simple type theory*. This is feasible since nearly all the basic ideas and principles of simple type theory are already found in first-order logic (the notion of a type is the most notable exception). After a shortened introduction to first-order logic, students could immediately proceed to simple type theory. Subjects, such as proof systems and model theory, could be briefly addressed in first-order logic and then explored further in simple type theory.

Although students would spend less time on directly studying first-order logic, their study of simple type theory would foster a deeper understanding of both first-order logic and the logical principles that transcend individual logics. But the most important benefit would be that the students—especially those who are going to become computer scientists and software engineers—would go into the real world with a practical logic in their toolkit.

We also recommend that the curricula for undergraduate computer science and software engineering students include instruction on how to apply the simple type theory they learn in a logic course to practical problems that arise in their other courses. Computer scientists and software engineers increasingly need to understand and actually use practical logics. For example, software engineers—and sometimes even other kinds of engineers—need to know how to read and write precise specifications in a formal logic. With the attributes we have discussed, simple type theory provides an excellent

logical foundation for learning how to use industrial specification languages like B [1], VDM-SL [13], and Z [54].

10 Acknowledgments

Special thanks is given to Peter Andrews for offering the author many useful suggestions and comments and for writing *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof* [4]. Many of the definitions associated with STT are modifications of definitions given in this excellent textbook. The author is also grateful to Wolfram Kahl, Jérémie Wajs, and Jeffery Zucker for reading preliminary drafts of the paper and to the referees for their suggestions.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] W. Ackermann. Begründung des "tertium non datur" mittels der Hilbertschen Theorie der Widerspruchsfreiheit. *Mathematische Annalen*, 93:1–36, 1924.

[3] P. B. Andrews. A reduction of the axioms for the theory of propositional types. *Fundamenta Mathematicae*, 52:345–350, 1963.

[4] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof, Second Edition*. Kluwer, 2002.

[5] P. B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfennig, and H. Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17:471–522, 1985.

[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[8] L. Chwistek. Antynomje logikiformalnej. *Przeglad Filozoficzny*, 24:164–171, 1921.

[9] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[10] Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*, 2003. Available at `http://pauillac.inria.fr/coq/doc/main.html`.

[11] C. Coquand and T. Coquand. Structured type theory. In A. Felty, editor, *LMF'99: Workshop on Logical Frameworks and Meta-languages*, 1999. Available at `http://www.site.uottawa.ca/~afelty/LFM99/`.

[12] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[13] J. Dawes. *The VDM-SL*. Pitman/UCL Press, 1991.

[14] R. Dedekind. *Was sind und was sollen die Zahlen?* Braunschweig, 1888.

[15] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, second edition, 2000.

[16] W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.

[17] W. M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.

[18] W. M. Farmer. A Basic Extended Simple Type Theory. SQRL Report No. 14, McMaster University, 2003. Revised 2004.

[19] W. M. Farmer. Formalizing undefinedness arising in calculus. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning—IJCAR 2004*, volume 3097 of *Lecture Notes in Computer Science*, pages 475–489. Springer-Verlag, 2004.

[20] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[21] S. Feferman. Predicativity. In S. Shapiro, editor, *Handbook of the Philosophy of Mathematics and Logic*, pages 590–624. Oxford University Press, 2005.

[22] R. O. Gandy. The simple theory of types. In R. Gandy and M. Hyland, editors, *Logic Colloquium 76*, pages 173–181. North-Holland, 1977.

[23] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[24] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[25] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[26] A. P. Hazen. Predicative logics. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume I, pages 331–407. Reidel, 1983.

[27] L. Henkin. The completeness of the first-order functional calculus. *Journal of Symbolic Logic*, 15:159–166, 1949.

[28] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[29] L. Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52:323–344, 1963.

[30] L. Henkin. The discovery of my completeness proofs. *Bulletin of Symbolic Logic*, 2:127–158, 1996.

[31] L. J. Henschen. N-sorted logic for automatic theorem proving in higher-order logic. In J. J. Donovan and R. Shields, editors, *Proceedings of the ACM Annual Conference*, pages 71–81. ACM Press, 1972.

[32] D. Hilbert. Über den Zahlbegriff. *Jahresbericht der Deutschen Mathematikervereinigung*, 8:180–184, 1900.

[33] W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[34] B. Jacobs and T. Melham. Translating dependent type theory into higher order logic. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[35] R. B. Jensen. On the consistency of a slight (?) modification of Quine's NF. *Synthese*, 19:250–263, 1969.

[36] M. Kerber. How to prove higher order theorems in first order logic. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 137–142. Morgan Kaufmann, 1991.

[37] M. Kerber and M. Kohlhase. A mechanization of strong Kleene logic for partial functions. In A. Bundy, editor, *Automated Deduction—CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 371–385. Springer-Verlag, 1994.

[38] G. Kreisel. Informal rigour and completeness proofs. In I. Lakatos, editor, *Problems in the Philosophy of Mathematics*, pages 138–157. North-Holland, 1967.

[39] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1986.

[40] Lemma 1 Ltd. *ProofPower: Description*, 2000. Available at `http://www.lemma-one.com/ProofPower/doc/doc.html`.

[41] S. Mac Lane. *Mathematics: Form and Function*. Springer-Verlag, 1986.

[42] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[43] A. R. D. Mathias. The strength of Mac Lane set theory. *Annals of Pure and Applied Logic*, 110:107–234, 2001.

[44] R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics*. North Holland, 1994.

[45] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[46] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[47] G. Peano. *Arithmetices principia nova methodo exposita*. Bocca, Turin, Italy, 1889.

[48] R. Pollack. *The Theory of LEGO*. PhD thesis, University of Ediburgh, 1994.

[49] F. P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society, Series 2*, 25:338–384, 1926.

[50] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24:709–720, 1998.

[51] B. Russell. On denoting. *Mind (New Series)*, 14:479–493, 1905.

[52] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908.

[53] S. Shapiro. *Foundations without Foundationalism: A Case for Second-order Logic*. Oxford University Press, 2000.

[54] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[55] A. M. Turing. Practical forms of type theory. *Journal of Symbolic Logic*, 13:80–94, 1948.

[56] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1910–13. Paperback version to section *56 published in 1964.