A Basic Extended Simple Type Theory[*]

William M. Farmer[†]
McMaster University

June 9, 2004

Abstract

This paper presents an extended version of Church's simple type theory called *Basic Extended Simple Type Theory* (BESTT). By adding type variables and support for reasoning with tuples, lists, and sets to simple type theory, it is intended to be a practical logic for formalized mathematics.

1 Introduction

B. Russell introduced a logic in 1908 [16] now called the *ramified theory of types* to serve as a foundation for mathematics. It included a hierarchy of types to avoid the set-theoretic and semantic paradoxes that troubled mathematicians and philosophers in the early 1900s and was employed as the logic of Whitehead and Russell's monumental *Principia Mathematica* [18]. Being an overly complex system, L. Chwistek [5] and F. Ramsey [15] suggested in the 1920s a simplified formulation of the ramified theory of types called the *simple theory of types* or, more briefly, *simple type theory*.

A. Church presented in 1940 [4] a version of simple type theory that included lambda-notation. Church's simple type theory, which we will denote as CSTT, can be viewed as a "function theory": functions are the basic objects and reasoning with functions can be done with the help of full quantification over functions, types, and lambda-notation. Many other mathematical objects—such as tuples, sequences, and sets—can be represented in CSTT as certain kinds of functions. The primitive basis of CSTT can be

[*]Published as SQRL Report No. 14, 12 pp., McMaster University, 2003.

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

made exceptionally small. The full machinery of predicate calculus can be developed in CSTT from just function application, function abstraction, and equality (first shown by L. Henkin in [11] and improved by P. Andrews in [1]).

CSTT has been widely influential in formalized mathematics. Many people have argued (e.g., see Andrews' remarks in [2]) that CSTT, with its strong support for reasoning with functions, is a more practical reasoning system than traditional Zermelo-Fraenkel set theory. CSTT is the basis of the logics used in several computer theorem proving systems including HOL [10], IMPS [8, 9], Isabelle [14], ProofPower [12], PVS [13], and TPS [3].

This paper presents an extended version of CSTT called *Basic Extended Simple Type Theory* (BESTT). It adds the following facilities to CSTT:

(1) Type variables for forming polymorphic types and expressions as in the HOL logic [10].

(2) New type constructors and expression constants for reasoning with tuples, lists (finite sequences), and sets.

BESTT is intended to be a practical logic for formalized mathematics that can be used either informally by hand or within a mechanized mathematics system that implements it. BESTT is essentially the same as the background theory of the ML programming language [17]. Therefore, it is an ideal logic in which to write and reason about specifications of ML programs.

The paper gives a full presentation of the syntax of BESTT, but only a few remarks are made about the semantics of BESTT. The presence of type variables in BESTT makes a full presentation of the semantics both lengthy and complicated. Moreover, there are two very natural semantics for BESTT, a *traditional semantics* in which all functions are total and all terms are defined and a *partial semantics* in which functions may be partial and terms may be undefined (but formulas are always either true or false). We leave it as an exercise for the BESTT enthusiast to write down either the traditional or partial semantics for BESTT using previous work (see section 5 for references) as a guide.

2 Type Languages

Let $\mathcal{A}$ be a fixed infinite set of symbols called *type variables*. Assume $\mathcal{A}$ does not contain the symbol $*$. A *type language* of BESTT is a set $\mathcal{B}$ of symbols called *type constants* such that:

(1) $\mathcal{B}$ contains one built-in type constant $*$, the type of truth values, as well as possibly other type constants.

(2) $\mathcal{A}$ and $\mathcal{B}$ are disjoint.

Let $\mathcal{B}$ be a type language. A *type* of $\mathcal{B}$ is a string of symbols defined by the rules below. $\mathbf{type}_{\mathcal{B}}[\alpha]$ asserts that α is a type of $\mathcal{B}$.

$$\textbf{T1} \quad \frac{\alpha \in \mathcal{A} \cup \mathcal{B}}{\mathbf{type}_{\mathcal{B}}[\alpha]} \quad \textbf{(Atomic type)}$$

$$\textbf{T2} \quad \frac{\mathbf{type}_{\mathcal{B}}[\alpha],\ \mathbf{type}_{\mathcal{B}}[\beta]}{\mathbf{type}_{\mathcal{B}}[(\alpha \rightarrow \beta)]} \quad \textbf{(Function type)}$$

$$\textbf{T3} \quad \frac{\mathbf{type}_{\mathcal{B}}[\alpha],\ \mathbf{type}_{\mathcal{B}}[\beta]}{\mathbf{type}_{\mathcal{B}}[(\alpha \times \beta)]} \quad \textbf{(Product type)}$$

$$\textbf{T4} \quad \frac{\mathbf{type}_{\mathcal{B}}[\alpha]}{\mathbf{type}_{\mathcal{B}}[\mathsf{list}[\alpha]]} \quad \textbf{(List type)}$$

$$\textbf{T5} \quad \frac{\mathbf{type}_{\mathcal{B}}[\alpha]}{\mathbf{type}_{\mathcal{B}}[\mathsf{set}[\alpha]]} \quad \textbf{(Set type)}$$

Let $\mathcal{T}_{\mathcal{B}}$ denote the set of types of $\mathcal{B}$. For $\alpha, \beta \in \mathcal{T}_{\mathcal{B}}$, α is an *instance* of β if α can be obtained from β by simultaneously substituting types of $\mathcal{B}$ for the type variables in β. Let the *kernel type language* of BESTT be the type language $\mathcal{B}_0 = \{*\}$.

c	$\tau(c)$ (Note: $\alpha, \beta \in \mathcal{A}$.)
$=$	$((\alpha \times \alpha) \rightarrow *)$
Π	$((\alpha \rightarrow *) \rightarrow *)$
ι	$((\alpha \rightarrow *) \rightarrow \alpha)$
mpair	$(\alpha \rightarrow (\beta \rightarrow (\alpha \times \beta)))$
fst	$((\alpha \times \beta) \rightarrow \alpha)$
snd	$((\alpha \times \beta) \rightarrow \beta)$
nil	$\mathsf{list}[\alpha]$
mlist	$((\alpha \times \mathsf{list}[\alpha]) \rightarrow \mathsf{list}[\alpha])$
hd	$(\mathsf{list}[\alpha] \rightarrow \alpha)$
tl	$(\mathsf{list}[\alpha] \rightarrow \mathsf{list}[\alpha])$
mset	$((\alpha \rightarrow *) \rightarrow \mathsf{set}[\alpha])$
$\in$	$((\alpha \times \mathsf{set}[\alpha]) \rightarrow *)$

Table 1: The Built-In Constant Symbols of a Language

3 Languages

Let $\mathcal{V}$ be a fixed infinite set of symbols called *variable symbols*. Assume $\mathcal{V}$ does not contain the symbols in the left column of Table 1. A *language* of BESTT is a tuple $L = (\mathcal{B}, \mathcal{C}, \tau)$ such that:

(1) $\mathcal{B}$ is a type language of BESTT.

(2) $\mathcal{C}$ is a set of symbols called the *constant symbols* of L. $\mathcal{C}$ contains the built-in constant symbols given in the left column of Table 1 as well as possibly other constant symbols.[1]

(3) $\mathcal{V}$ and $\mathcal{C}$ are disjoint.

(4) $\tau : \mathcal{C} \rightarrow \mathcal{T}_{\mathcal{B}}$ is a total function. The definition of τ on the built-in constant symbols is given in the right column of Table 1.

Let $L = (\mathcal{B}, \mathcal{C}, \tau)$ be a language of BESTT. In BESTT terms and formulas are merged together and called "expressions". An *expression of type* α of L is a string of symbols defined by the rules below. $\mathbf{expr}_L[E, \alpha]$ asserts that E is an expression of type α of L.

[1] mpair, mlist, and mset are short for make-pair, make-list, and make-set, respectively.

E1 $\dfrac{x \in \mathcal{V},\ \mathbf{type}_{\mathcal{B}}[\alpha]}{\mathbf{expr}_L[(x : \alpha), \alpha]}$ **(Variable)**

E2 $\dfrac{c \in \mathcal{C},\ \alpha \text{ is an instance of } \tau(c)}{\mathbf{expr}_L[(c : \alpha), \alpha]}$ **(Constant)**

E3 $\dfrac{\mathbf{expr}_L[A, \alpha],\ \mathbf{expr}_L[F, (\alpha \rightarrow \beta)]}{\mathbf{expr}_L[F(A), \beta]}$ **(Function application)**

E4 $\dfrac{x \in \mathcal{V},\ \mathbf{type}_{\mathcal{B}}[\alpha],\ \mathbf{expr}_L[B, \beta]}{\mathbf{expr}_L[(\lambda\, x : \alpha \,.\, B), (\alpha \rightarrow \beta)]}$ **(Function abstraction)**

Let $\mathcal{E}_L$ denote the set of expressions of L. By induction on the structure of expressions, for each expression $E \in \mathcal{E}_L$, there is a unique type α such that $\mathbf{expr}_L[E, \alpha]$. A *formula* of L is an expression of L of type $*$. "Free variable", "closed", and similar notions are defined in the obvious way. A *sentence* is a closed formula. Let the *kernel language* of BESTT be the language $L_0 = (\mathcal{B}_0, \mathcal{C}_0, \tau_0)$ of BESTT such that $\mathcal{B}_0$ is the kernel type language of BESTT and $\mathcal{C}_0$ contains only the built-in constant symbols in Table 1.

In the rest of the paper, let $L = (\mathcal{B}, \mathcal{C}, \tau)$ be a language of BESTT. Let α, β, etc. denote types of $\mathcal{B}$, and let A_α, B_α, etc. denote expressions of type α of L.

4 Definitions and Abbreviations

The following definitions introduce additional notation and vocabulary:

Pair:

(A_α, B_β)	denotes	$(\mathsf{mpair} : \gamma)(A_\alpha)(B_\beta)$ where $\gamma = (\alpha \rightarrow (\beta \rightarrow (\alpha \times \beta)))$.

Tuple:

$(A^1_{\alpha_1}, \ldots, A^n_{\alpha_n})$	denotes	$(A^1_{\alpha_1}, (A^2_{\alpha_2}, \ldots, A^n_{\alpha_n}))$ where $n \geq 3$.

Extended product type:

$(\alpha_1 \times \cdots \times \alpha_n)$	denotes	$(\alpha_1 \times (\alpha_2 \times \cdots \times \alpha_n))$ where $n \geq 3$.

Multivariate function application:

$F_\beta(A^1_{\alpha_1}, \ldots, A^n_{\alpha_n})$	denotes	$F_\beta((A^1_{\alpha_1}, \ldots, A^n_{\alpha_n}))$ where $n \geq 2$ and $\beta = ((\alpha_1 \times \cdots \times \alpha_n) \rightarrow \gamma)$.

First tuple projection:

$\#1(A^1_{\alpha_1}, \ldots, A^n_{\alpha_n})$	denotes	$(\mathsf{fst} : \beta)(A^1_{\alpha_1}, \ldots, A^n_{\alpha_n})$ where $n \geq 2$ and $\beta = ((\alpha_1 \times \cdots \times \alpha_n) \rightarrow \alpha_1)$.

Higher tuple projection:

$\#m(A^1_{\alpha_1}, \ldots, A^n_{\alpha_n})$	denotes	$\#(m-1)(\mathsf{snd} : \beta)(A^1_{\alpha_1}, \ldots, A^n_{\alpha_n})$ where $2 \leq m \leq n$, $n \geq 2$, and $\beta = ((\alpha_1 \times \cdots \times \alpha_n) \rightarrow (\alpha_2 \times \cdots \times \alpha_n))$.

Equality:

$(A_\alpha = B_\alpha)$	denotes	$(= : \beta)(A_\alpha, B_\alpha)$ where $\beta = ((\alpha \times \alpha) \rightarrow *)$.

Logical equality:

$(A_* \Leftrightarrow B_*)$	denotes	$(A_* = B_*)$.

True:

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

False:

F	denotes	$((\lambda\, x : * \,.\, \mathsf{T}) = (\lambda\, x : * \,.\, (x : *)))$.

Negation:

$\neg A_*$	denotes	$(A_* = \mathsf{F})$.

Inequality:

$(A_\alpha \neq B_\alpha)$	denotes	$\neg(A_\alpha = B_\alpha)$.

Conjunction:

$(A_* \wedge B_*)$	denotes	$((A_*, B_*) = (\mathsf{T}, \mathsf{T}))$.

Disjunction:

$(A_* \vee B_*)$	denotes	$\neg(\neg A_* \wedge \neg B_*)$.

Implication:

$(A_* \Rightarrow B_*)$	denotes	$(\neg A_* \vee B_*)$.

Universal quantification:

$(\forall\, x : \alpha \,.\, A_*)$ denotes $(\Pi : \beta)((\lambda\, x : \alpha \,.\, A_*))$
where $\beta = ((\alpha \rightarrow *) \rightarrow *)$.

Existential quantification:

$(\exists\, x : \alpha \,.\, A_*)$ denotes $\neg(\forall\, x : \alpha \,.\, \neg A_*)$.

Multivariate binding:

$(\Box\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, A_\alpha)$
denotes $(\Box\, x_1 : \alpha_1 \,.\, (\Box\, x_2 : \alpha_2, \ldots, x_n : \alpha_n \,.\, A_\alpha))$
where $\Box \in \{\lambda, \forall, \exists\}$ and $n \geq 2$.

Definedness:

$(A_\alpha \downarrow)$ denotes $(\exists\, x : \alpha \,.\, ((x : \alpha) = A_\alpha))$
where $(x : \alpha)$ does not occur in A_α.

Undefinedness:

$(A_\alpha \uparrow)$ denotes $\neg(A_\alpha \downarrow)$.

Equivalence:

$(A_\alpha \simeq B_\alpha)$ denotes $(((A_\alpha \downarrow) \vee (B_\alpha \downarrow)) \Rightarrow (A_\alpha = B_\alpha))$.

Definite description:

$(\mathrm{I}\, x : \alpha \,.\, A_*)$ denotes $(\iota : \beta)((\lambda\, x : \alpha \,.\, A_*))$
where $\beta = ((\alpha \rightarrow *) \rightarrow \alpha)$.

Conditional expression:

$\mathsf{if}(A_*, B_\alpha, C_\alpha)$ denotes $(\mathrm{I}\, x : \alpha \,.\, ((A_* \Rightarrow ((x : \alpha) = B_\alpha)) \wedge$
$(\neg A_* \Rightarrow ((x : \alpha) = C_\alpha))))$.
where $(x : \alpha)$ does not occur in
A_*, B_α, or C_α.

Canonical undefined expression:

$\bot_\alpha$ denotes $(\mathrm{I}\, x : \alpha \,.\, ((x : \alpha) \neq (x : \alpha)))$.

Empty list:

$\langle\rangle_\alpha$ denotes $(\mathsf{nil} : \mathsf{list}[\alpha])$.

Single member list:

$\langle A_\alpha \rangle$ denotes $(\mathsf{mlist} : \beta)(A_\alpha, \langle\rangle_\alpha)$
where $\beta = ((\alpha \times \mathsf{list}[\alpha]) \rightarrow \mathsf{list}[\alpha])$.

Multimember list:

$\langle A_\alpha^1, \ldots, A_\alpha^n \rangle$	denotes	$(\mathsf{mlist} : \beta)(A_\alpha^1, \langle A_\alpha^2 \ldots, A_\alpha^n \rangle)$ where $n \geq 2$ and $\beta = ((\alpha \times \mathsf{list}[\alpha]) \rightarrow \mathsf{list}[\alpha])$.

First list projection:

$L_\alpha[0]$	denotes	$\mathsf{if}(L_\alpha \neq \langle\rangle_\alpha, (\mathsf{hd} : \gamma)(L_\alpha), \bot_\beta)$ where $\alpha = \mathsf{list}[\beta]$ and $\gamma = \alpha \rightarrow \beta$.

Higher list projection:

$L_\alpha[m]$	denotes	$\mathsf{if}(L_\alpha \neq \langle\rangle_\alpha, (\mathsf{tl} : \gamma)(L_\alpha)[m-1], \bot_\beta)$ where $0 < m$, $\alpha = \mathsf{list}[\beta]$, and $\gamma = \alpha \rightarrow \alpha$.

Set abstraction:

$(\mathrm{S}\, x : \alpha \,.\, A_*)$	denotes	$(\mathsf{mset} : \beta)((\lambda\, x : \alpha \,.\, A_*))$ where $\beta = ((\alpha \rightarrow *) \rightarrow \mathsf{set}[\alpha])$.

Traditional set abstraction:

$\{(x : \alpha) \mid A_*\}$	denotes	$(\mathrm{S}\, x : \alpha \,.\, A_*)$.

Set membership:

$(A_\alpha \in B_\beta)$	denotes	$(\in : \gamma)(A_\alpha, B_\beta)$ where $\beta = \mathsf{set}[\alpha]$ and $\gamma = ((\alpha \times \mathsf{set}[\alpha]) \rightarrow *)$.

The following abbreviation rules can be used to write expressions in a more compact form:

A1 A variable $(x : \alpha)$ occurring in the body B of $(\Box\, x : \alpha \,.\, B)$ where $\Box \in \{\lambda, \forall, \exists, \mathrm{I}, \mathrm{S}\}$ may be written as x if there is no resulting ambiguity.

A2 A variable $(x : \alpha)$ occurring freely in an expression E may be written as x if α can be determined from the rest of the expression.

A3 A constant $(c : \alpha)$ occurring in an expression E may be written as c if $\alpha = \tau(c)$ and contains no type variables or if α can be determined from the rest of the expression.

A4 A matching pair of parentheses in an expression may be dropped if there is no resulting ambiguity.

A5 An application

$$(c : ((\alpha \times \beta) \rightarrow \gamma))(A_\alpha, B_\beta)$$

will sometimes be written in infix notation as

$$(A_\alpha\ (c : ((\alpha \times \beta) \to \gamma))\ B_\beta)$$

or

$$(A_\alpha\ c\ B_\beta).$$

With the help of the definitions and abbreviation rules given in this section, expressions can usually be written in a natural and easy-to-read form. The definitions and abbreviation rules are used in the rest of the paper.

5 Semantics

As we mentioned in the introduction, there are two natural semantics for BESTT. The traditional semantics can be directly adapted from the semantics for the HOL logic given in [10]. According to the traditional semantics, every expression is defined, i.e., for every expression A_α of L, $(A_\alpha\downarrow)$ is true in every model for L. A 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 an unspecified member of type α otherwise.

The partial semantics for BESTT is based on the partial semantics for CSTT given in [6, 7]. This semantics can be straightforwardly extended to handle type variables (following the semantics in [10]) and the machinery in BESTT for tuples, lists, and sets. The definedness of expressions is determined by the following principles:

P1 (Partial functions) Expressions may denote partial functions.

P2 (Formulas) Formulas (i.e., expressions of type $*$) are always defined and denote either true or false.

P3 (Universally defined expressions) Variables, constants, function abstractions, and set abstractions are always defined.

P4 (Predicates) Predicates (i.e., functions of type $\alpha \to *$) are always total.

P5 (Function applications) A function application of type $\alpha \neq *$ is defined iff the function and argument expressions are defined, denoting f and a, respectively, and f is defined at a.

P6 (Predicate applications) A predicate application (i.e., a function application of type $*$) is false if either the predicate expression or the argument expression is undefined.

P7 (Definite descriptions) The value of a definite description $(\mathrm{I}\, x : \alpha \;.\; A_*)$ where $\alpha \neq *$ is the unique value x of type α satisfying A_* if it exists and $(\mathrm{I}\, x : \alpha \;.\; A_*)$ is undefined otherwise.

P8 (Built-in constants) Constants of the form $(\mathsf{mpair} : \alpha)$, $(\mathsf{fst} : \alpha)$, $(\mathsf{snd} : \alpha)$, and $(\mathsf{mlist} : \alpha)$ denote total functions, while $(\mathsf{hd} : \alpha)$ and $(\mathsf{tl} : \alpha)$ denote functions that are defined at all values except the empty list.

6 Theories

A *theory* of BESTT is a pair $T = (L, \Gamma)$ where L is a language of BESTT and Γ is a set of sentences of L called the *axioms* of T. A theory serves as a formal mathematical model or as a specification of a family of mathematical models.

A formula A of L is *valid* in T, written $T \models A$, if A is a logical consequence of Γ.

Example 6.1 (Orders) Let $\mathcal{B}$ be a type language of BESTT such that $\mathcal{B} = \{\mathbf{E}\} \cup \mathcal{B}_0$, and let $L = (\mathcal{B}, \mathcal{C}, \tau)$ be a language of BESTT such that $\mathcal{C} = \{\leq\} \cup \mathcal{C}_0$ and $\tau(\leq) = ((\mathbf{E} \times \mathbf{E}) \to *)$. Then let $T_1 = (L, \{A_1, A_2\})$, $T_2 = (L, \{A_1, A_2, A_3\})$, and $T_3 = (L, \{A_1, A_2, A_3, A_4\})$ where:

(1) A_1 is $\forall\, x : \mathbf{E} \;.\; x \leq x$ (Reflexivity).

(2) A_2 is $\forall\, x, y, z : \mathbf{E} \;.\; (x \leq y \wedge y \leq z) \Rightarrow x \leq z$ (Transitivity).

(3) A_3 is $\forall\, x, y : \mathbf{E} \;.\; (x \leq y \wedge y \leq x) \Rightarrow x = y$ (Antisymmetry).

(4) A_4 is $\forall\, x, y : \mathbf{E} \;.\; x \leq y \vee y \leq x$ (Comparability).

T_1, T_2, and T_3 are theories of a preorder, partial order, and total order, respectively. $\square$

Example 6.2 (Peano arithmetic) Let $\mathcal{B}$ be a type language of BESTT such $\mathcal{B} = \{\mathbf{N}\} \cup \mathcal{B}_0$, and let $L = (\mathcal{B}, \mathcal{C}, \tau)$ be a language of BESTT such that $\mathcal{C} = \{0, S\} \cup \mathcal{C}_0$, $\tau(0) = \mathbf{N}$, and $\tau(S) = (\mathbf{N} \to \mathbf{N})$. Let $T = (L, \{A_1, A_2, A_3\})$ where:

(1) A_1 is $\forall\, x : \mathbf{N}\ .\ 0 \neq S(x)$ (0 has no predecessor).

(2) A_2 is $\forall\, x, y : \mathbf{N}\ .\ S(x) = S(y) \Rightarrow x = y$ (S is injective).

(3) A_3 is

$$\forall\, P : (\mathbf{N} \to *)\ .\ (P(0) \wedge \forall\, x : \mathbf{N}\ .\ P(x) \Rightarrow P(S(x))) \Rightarrow \forall\, x : \mathbf{N}\ .\ P(x)$$

(Induction principle).

T is (second-order) Peano arithmetic, a theory that formalizes natural number arithmetic. Addition and multiplication on the natural numbers can be defined in this theory by primitive recursion. □

References

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

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

[3] P. B. Andrews, M. Bishop, and C. E. Brown. System description: TPS: A theorem proving system for type theory. In D. McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 164–169. Springer-Verlag, 2000.

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

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

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

[7] W. M. Farmer. Formalizing undefinedness arising in calculus. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning—IJCAR 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004. Forthcoming.

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

[9] W. M. Farmer, J. D. Guttman, and F. J. Thayer Fábrega. IMPS: An updated system description. In M. McRobbie and J. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 1996.

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

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

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

[13] 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.

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

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

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

[17] J. D. Ullman. *Elements of ML Programming.* Prentice Hall, 1998.

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