Reasoning about Partial Functions with the Aid of a Computer[*]

William M. Farmer

11 August 1995

Abstract

Partial functions are ubiquitous in both mathematics and computer science. Therefore, it is imperative that the underlying logical formalism for a general-purpose mechanized mathematics system provide strong support for reasoning about partial functions. Unfortunately, the common logical formalisms—first-order logic, type theory, and set theory—are usually only adequate for reasoning about partial functions *in theory*. However, the approach to partial functions traditionally employed by mathematicians is quite adequate *in practice*. This paper shows how the traditional approach to partial functions can be formalized in a range of formalisms that includes first-order logic, simple type theory, and Von-Neumann-Bernays-Gödel set theory. It argues that these new formalisms allow one to directly reason about partial functions; are based on natural, well-understood, familiar principles; and can be effectively implemented in mechanized mathematics systems.

[*]© Kluwer Academic Publishers. Published in *Erkenntnis*, 43:279-294, 1995. Supported by the MITRE-Sponsored Research program. This paper is a written version (with references) of an address given at the Partial Functions and Programming: Foundational Questions conference held 17 February 1995 at the University of California at Irvine. Author's address: Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario L8S 4K1, Canada. E-mail: `wmfarmer@mcmaster.org`.

1 Introduction

Reasoning about partial functions is one of the fundamental problems of mechanized mathematics. But what are partial functions and what is mechanized mathematics?

In practice, a function f usually has both a *domain of definition* D_f consisting of the values at which it is defined and a *domain of application* D^*_f consisting of the values to which it may be applied. These two domains may be different from each other. For example, the division function is defined at $\langle x, y \rangle$ iff x and y are real numbers with $y \neq 0$, but it can be applied to any pair of real numbers. Hence, a statement like

$$\forall\, x \in \mathbf{R} \,.\, x \neq 0 \supset x/x = 1$$

makes perfectly good sense even though x/x would be undefined (nondenoting) if $x = 0$.

A function f is *total* if $D_f = D^*_f$ and is *partial* if $D_f \subseteq D^*_f$. Thus a total function is a special case of a partial function. In both mathematics and computer science, strictly partial functions are ubiquitous. In fact, mathematicians usually refer to partial functions simply as functions; for them there is nothing unusual about a function which is not defined at each value to which it can be applied.

The goal of *mechanized mathematics* is to produce computer environments that support and improve rigorous mathematical reasoning. Mechanized mathematics is a highly interdisciplinary, but very young field that involves logic, mathematics, automated reasoning (as used in automated theorem provers), symbolic computation (as used in computer algebra systems), and human-computer interaction. Its main application area is currently *formal methods* for developing and analyzing computer hardware and software. However, the most important application area in the future will certainly be *mathematics education.* And, as the discipline matures, mechanized mathematics will also be used increasingly in *mathematics research.* I believe that, by automating the process of doing mathematics, mechanized mathematics systems will transform how mathematics is learned and practiced in the next century.

One of the principal design decisions for a mechanized mathematics system is the choice of what the underlying logical formalism should be. Since partial functions are so important and so prevalent in mathematics and computer science, the chosen formalism ought to provide strong support for reasoning about partial functions. It is also imperative that the chosen formalism be natural (so that it is compatible with the user's intuition), well-understood (so that it can be effectively implemented), and widely familiar (so that it is not an obstacle to learning to use the system). Unfortunately, the common formalisms—first-order logic, type theory, and set theory—are inadequate for reasoning about partial functions *in practice*.

First-order logic with function symbols has some machinery for reasoning about total functions. Total functions can be represented using function symbols, but there is no support for quantifying over functions or for specifying functions by means of lambda-notation.

There are many kinds of type theory. All of them provide special machinery, such as types and lambda-notation, for working with functions. This machinery is effective for reasoning about total functions, but it usually can only be used to reason about partial functions in indirect and artificial ways.

Axiomatic (first-order) set theory is the most popular foundation for mathematics. Partial functions can be easily represented in set theory as certain sets of ordered pairs. (In fact, in set theories without proper classes, all functions are strictly partial.) However, in set theory there is no special machinery for reasoning about functions, total or partial. For instance, there is no built-in mechanism for directly applying a term representing a function to a term representing an argument to form a new term.

Many formalisms have been proposed that are intended to support partial functions (see the References section for examples). These formalisms have usually been rejected by system developers because their semantics is thought to be too arcane and their implementation too difficult.

The objective of this talk is to show that there are formalisms which support partial functions; are based on natural, well-understood, familiar principles; and can be effectively implemented.

2 IMPS

My ideas on reasoning about partial functions in mechanized mathematics systems have been greatly influence by IMPS, an Interactive Mathematical Proof System [13, 14] developed by Joshua Guttman, Javier Thayer Fábrega, and myself. IMPS is intended to be an environment for rigorous mathematical reasoning that is useful to a wide range of people. The strategy for achieving this objective has been to try to provide mechanical support for traditional mathematical techniques and styles of practice.

The logic of IMPS, named LUTINS[1], is a version of simple type theory that admits partial functions, undefined terms, and subtypes. We chose it for IMPS because of its familiarity, expressiveness, and strong support for partial and higher-order functions. I will say much more about LUTINS later in the talk.

The IMPS system has proven to be a very effective tool for formalizing and reasoning about traditional mathematics. The IMPS theory library contains significant portions of logic, abstract algebra, and mathematical analysis with over 1200 replayable proofs. In sophistication, several of the theorems that have been proved reach about the level of the fundamental theorem of calculus. A major part of the success of IMPS is due to its ability to effectively deal with partial functions and undefined terms.

3 The Traditional Approach

Although common formalisms are inadequate for reasoning about partial functions, mathematicians have no difficulty in dealing with partial functions. Most mathematicians employ what I will call the *traditional approach to partial functions*. This approach can be boiled down to three principles:

(1) Variables and constants are always defined, i.e., they always denote something.

(2) Functions may be partial. The application of an expression denoting a function to an expression denoting a value outside of the function's domain gives an expression that is undefined (e.g., $1/0$ and $\lim_{x\to\infty} \sin x$ are undefined). Moreover, an application is undefined if any argument is undefined (e.g., $0 * (1/0)$ is undefined since $1/0$ is undefined).

(3) Formulas are always true or false. The application of a predicate (i.e., an expression denoting a truth-valued function) is always defined. Moreover, an application of a predicate is false if any argument is undefined (e.g., $1/0 = 1/0$ is false since $1/0$ is undefined).

I claim that, not only is this approach commonly used by mathematicians, it is the approach for dealing with partial functions like division that is usually taught to American students in college, high school, and even junior high school.

[1]Pronounced as the word in French.

Several formalizations of the traditional approach have been proposed. The ones I know about are:

- R. Schock (1968) [35].
- T. Burge (1971) [4, 5].
- M. Beeson (1981) [2, 3].
- L. Monk (1986) [30].
- S. Feferman (1990) [16].
- W. Farmer, J. Guttman, J. Thayer (1990) [10, 11, 12].

The last formalization in the list is LUTINS, the logic of IMPS. It is noteworthy because it has been implemented and tested in a general-purpose mechanized mathematics system.

4 Partial First-Order Logic

I will now introduce a system named Partial First-Order Logic (PFOL) to illustrate how the traditional approach can be formalized in first-order logic. PFOL has the usual connectives of first-order logic:

$$=, \neg, \wedge, \vee, \supset, \equiv, \forall, \exists.$$

In addition, it has a definite description operator I that is used to construct terms of the form $\mathrm{I}\, x \,.\, \varphi$. I is given a free semantics: $\mathrm{I}\, x \,.\, \varphi$ denotes the unique x that satisfies φ if there is such an x and is undefined otherwise. For example,

$$\mathrm{I}\, x \,.\, x \neq x$$

is an undefined term.

Several other useful symbols can be introduced as abbreviations:

- $s{\downarrow} \;\equiv\; s = s$ ("s is defined").
- $s{\uparrow} \;\equiv\; \neg(s{\downarrow})$ ("s is undefined").
- $s \simeq t \;\equiv\; s{\downarrow} \vee t{\downarrow} \supset s = t$ ("s and t are quasi-equal").
- $\mathrm{if}(\varphi, s, t) \;\equiv\; \mathrm{I}\, x \,.\, ((\varphi \supset x = s) \wedge (\neg\varphi \supset x = t))$ where x does not occur in φ, s, or t (an if-then-else term constructor).

The semantics of PFOL is very similar to that of ordinary first-order logic. For a given PFOL language $\mathcal{L}$, a *model* for $\mathcal{L}$ consists of a nonempty domain D plus a function which maps each individual constant of $\mathcal{L}$ to an element of D, each function symbol of $\mathcal{L}$ to a *partial* function from $D \times \cdots \times D$ to D, and each predicate symbol to a *total* function from $D \times \cdots \times D$ to $\{\mathrm{T}, \mathrm{F}\}$. The valuation function with respect to a model is generally partial: a term of the form $f(a)$ has no value in the model if the value of a is outside the domain of the value of f.

The machinery in PFOL for partial functions and undefined terms—the function symbols and the I operator—is purely a convenience; it extends but does not alter the conceptual framework of classical first-order logic. The use of function symbols and the I operator in a PFOL theory can be eliminated, and a PFOL theory without function symbols and I has the same semantics as an ordinary first-order theory without function symbols. As a consequence of these two facts, any theory of PFOL can be translated into a logically equivalent theory of ordinary first-order logic. That is, the following theorem is true:

Elimination Theorem *For every* PFOL *theory T, there is an ordinary first-order logic (*FOL*) theory T^* and a translation from each formula φ of T to a formula φ^* of T^* such that T^* involves no use of function symbols or the* I *operator and*

$$T \models_{\mathrm{PFOL}} \varphi \quad \textit{iff} \quad T^* \models_{\mathrm{FOL}} \varphi^*.$$

Moreover, $\varphi^* = \varphi$ if φ contains no function symbols nor I.

Most of the logical axiom schemas of PFOL are exactly the same as those for ordinary first-order logic. However, those dealing with instantiation and equality substitution are slightly different. For example, universal instantiation holds only for defined terms:

$$(\forall\, x\, .\, \varphi) \wedge t{\downarrow} \supset \varphi[x \mapsto t]$$

where t is free for x in φ. And the law of substitution holds for terms that are quasi-equal instead of just equal:

$$s \simeq t \;\supset\; \varphi(s) \equiv \varphi(t).$$

There are also new axiom schemas that formalize the properties of the definite description operator I and the definedness operators $\downarrow$ and $\uparrow$. For example, the definedness axiom schemas are:

- $a\downarrow$ for each variable or individual constant a.
- $t_1\uparrow \vee \cdots \vee t_n\uparrow \supset f(t_1, \ldots, t_n)\uparrow$ for each function symbol f.
- $t_1\uparrow \vee \cdots \vee t_n\uparrow \supset \neg p(t_1, \ldots, t_n)$ for each predicate symbol p.

Notice that these three axiom schemas correspond to the three principles of the traditional approach to partial functions.

A very important property of PFOL is that undefined terms are indiscernible. This means that an undefined term in a formula can be replaced by any other undefined term without changing the meaning of the formula. This property distinguishes PFOL (and other formalizations of the traditional approach) from *free logics*[2] in which there is some mechanism for reasoning about nonexistent entities such as the present king of France.

5 LUTINS

Simple type theory à la Church [7] is a higher-order logic for reasoning about truth values, individuals, and simply typed total functions. I will briefly describe a version of simple type theory named Partial Simple Type Theory (PSTT) in which the traditional approach has been formalized. (For a detailed presentation of a system like PSTT, see [10].) PSTT has the usual hierarchy of types consisting of base types and functions types. The base types are $\iota, \iota', \ldots$ (which denote domains $D, D', \ldots$ of individuals) and $*$ (which denotes the domain $\{\mathrm{T}, \mathrm{F}\}$ of truth values). A function type has the form $\alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$ where $\alpha_1, \ldots, \alpha_{n+1}$ are base or function types.

A type is of *kind* $*$ if the type is $*$ or has the form $\alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$ where α_{n+1} is of kind $*$; a type is of *kind* ι if it is not of kind $*$. For example, $\iota \rightarrow (* \rightarrow \iota')$ is of kind ι and $\iota \times \iota \rightarrow *$ is of kind $*$. A function type of kind ι denotes a domain of *partial* functions, while a function type of kind $*$ denotes a domain of *total* functions (in accordance with the second and third principles of the traditional approach). For instance, $\iota \rightarrow (* \rightarrow \iota')$ denotes the domain of partial functions from D to the domain of partial functions from $\{\mathrm{T}, \mathrm{F}\}$ to D', and $\iota \times \iota \rightarrow *$ denotes the domain of total functions from $D \times D$ to $\{\mathrm{T}, \mathrm{F}\}$ (i.e., the domain of binary predicates on D).

[2]There is a substantial literature on free logic that begins approximately with H. Leonard's 1956 paper "The logic of existence" [26]. For some samples from this literature see [18, 19, 23, 24, 25, 33, 35, 36, 39].

There are a few new logical axiom schemas that are different from those of PFOL and ordinary simple type theory. A lambda-expression is always defined:

$$(\lambda\, x_1, \ldots, x_n \,.\, t)\!\downarrow .$$

Like universal instantiation, beta-reduction is restricted to defined expressions:

$$a_1\!\downarrow \wedge \cdots \wedge a_n\!\downarrow\, \supset\, (\lambda\, x_1, \ldots, x_n \,.\, t)(a_1, \ldots, a_n) \simeq t[x_1 \mapsto a_1,\, \ldots,\, x_n \mapsto a_n]$$

where each a_i is free for x_i in t. And extensionality holds for partial as well as total functions:

$$f = g \;\equiv\; \forall\, x_1, \ldots, x_n \,.\, f(x_1, \ldots, x_n) \simeq g(x_1, \ldots, x_n).$$

A shortcoming of PSTT is that the types indicate the domain and range of *total* functions, but not partial functions. What is needed is a system of subtypes and types for organizing partial functions like the PSTT system of types for organizing total functions. Let us define a *sort* to be a subtype or type. LUTINS is PSTT plus a system of sorts. I will give a quick sketch of the LUTINS sort system. For a detailed description of LUTINS and its sort system, see [11, 13].

A language $\mathcal{L}$ of LUTINS contains a hierarchy of *atomic* and *compound* sorts. Each atomic sort is assigned an enclosing sort (which may be itself). A compound sort has the form $\alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$ where $\alpha_1, \ldots, \alpha_{n+1}$ are atomic or compound sorts. The atomic/enclosing sort relationship determines a partial order $\preceq$ on the sorts of $\mathcal{L}$ with the following properties:

- If α is an atomic sort and β is its enclosing sort, then $\alpha \preceq \beta$.
- If $\alpha_i \preceq \beta_i$ for all i with $1 \leq i \leq n+1$, then $\alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1} \preceq \beta_1 \times \cdots \times \beta_n \rightarrow \beta_{n+1}$.
- The PSTT types are the maximal sorts in $\preceq$.
- For each sort α, there is a unique type $\tau(\alpha)$ such that $\alpha \preceq \tau(\alpha)$.

The sorts of $\mathcal{L}$ denote nonempty sets and $\preceq$ entails set inclusion. A compound sort $\alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$ whose type is of kind ι denotes the domain of partial functions from $D_1 \times \cdots \times D_n$ to D_{n+1}, where each D_i is the domain denoted by α_i. (A compound sort whose type is of kind $*$ denotes a certain domain of total functions.) Notice that the second property of $\preceq$

makes sense because a partial function from $D_1 \times \cdots \times D_n$ to D_{n+1} is also a partial function from $D'_1 \times \cdots \times D'_n$ to D'_{n+1} whenever $D_i \subseteq D'_i$ for all i with $1 \leq i \leq n+1$.

As an example, consider a language for the real numbers containing the atomic sorts $\mathbf{Z}$, $\mathbf{Q}$, $\mathbf{R}$, and $*$ whose enclosing sorts are $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{R}$, and $*$, respectively. ($\mathbf{R}$ and $*$ are the base types.) Then $\mathbf{Z} \preceq \mathbf{Q} \preceq \mathbf{R}$ and $\mathbf{Q} \to \mathbf{Z} \preceq \mathbf{R} \to \mathbf{R}$. If $\mathbf{Z}$, $\mathbf{Q}$, and $\mathbf{R}$ denote the integers, rationals, and reals, respectively, then $\mathbf{Q} \to \mathbf{Z}$ would denote the domain of partial functions from the rationals to the integers. This domain would be a subset of the domain denoted by $\mathbf{R} \to \mathbf{R}$, i.e., the domain of partial functions from the reals to the reals.

Sorts are used in three ways in LUTINS. First, they are used to restrict binding operators. For example, the Archimedean principle of the real numbers is expressed by the sentence

$$\forall\, x : \mathbf{R} \,.\, \exists\, y : \mathbf{Z} \,.\, x < y,$$

and the integer division function is specified by the expression

$$\lambda\, x, y : \mathbf{Z} \,.\, \mathrm{I}\, z : \mathbf{Z} \,.\, x = y * z,$$

where $*$ is the multiplication operator defined on the real numbers.

Second, every expression e is assigned a sort $\sigma(e)$ on the basis of its syntax. The main rules that govern the assignment are:

- Variables and constants are assigned sorts when they are specified.
- $\sigma(f(t_1, \ldots, t_n)) = \alpha_{n+1}$ if $\sigma(f) = \alpha_1 \times \cdots \times \alpha_n \to \alpha_{n+1}$.
- $\sigma(\lambda\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, t) = \alpha_1 \times \cdots \times \alpha_n \to \alpha_{n+1}$ if $\sigma(t) = \alpha_{n+1}$.
- $\sigma(\mathrm{I}\, x : \alpha \,.\, \varphi) = \alpha$.

$\sigma(e) = \alpha$ means, if e is defined, the value of e is in the set denoted by α. That is, if an expression is defined, its assigned sort gives some immediate information about its value, which is very useful to both the human user and the computer. (Of course, if the value of an expression is in a set denoted by a subsort of the expression's assigned sort, the subsort might be a more useful indicator of expression's value than the assigned sort.)

And third, sorts facilitate the construction of interpretations of one LUTINS theory in another (see [12]).

Definedness is expressed in LUTINS by two operators. The expression

$$e \downarrow \alpha,$$

read "e is defined in α", means the e is defined and its value is a member of the set denoted by α. The expression

$$e\downarrow,$$

read "e is defined", is an abbreviation for $e \downarrow \sigma(e)$.

Most of the axiom schemas of LUTINS are essentially the same as those of PSTT except in some schemas "defined-in" is used in place of "is-defined":

- $(\forall\, x : \alpha \,.\, \varphi) \wedge (t \downarrow \alpha) \;\supset\; \varphi[x \mapsto t]$ where t is free for x in φ.
- $(a_1 \downarrow \alpha_1) \wedge \cdots \wedge (a_n \downarrow \alpha_n) \;\supset$
  $(\lambda\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, t)(a_1, \ldots, a_n) \simeq t[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$
  where each a_i is free for x_i in t.
- $f\!\uparrow \vee (t_1 \uparrow \alpha_1) \vee \cdots \vee (t_n \uparrow \alpha_n) \;\supset\; f(t_1, \ldots, t_n)\!\uparrow$
  where $\sigma(f) = \alpha_1 \times \cdots \times \alpha_n \to \alpha_{n+1}$.
- $(t_1 \uparrow \alpha_1) \vee \cdots \vee (t_n \uparrow \alpha_n) \;\supset\; \neg p(t_1, \ldots, t_n)$
  where $\sigma(p) = \alpha_1 \times \cdots \times \alpha_n \to *$.

To illustrate how the sort mechanism works, assume $\mathbf{N}$ and $\mathbf{R}$ are sorts and !, /, and $\sqrt{\ }$ are constants such that:

- $\mathbf{N} \preceq \mathbf{R}$.
- $\sigma(!) = \mathbf{N} \to \mathbf{N}$.
- $\sigma(/) = \mathbf{R} \times \mathbf{R} \to \mathbf{R}$.
- $\sigma(\sqrt{\ }) = \mathbf{R} \to \mathbf{R}$.

Assume further that $\mathbf{N}$ and $\mathbf{R}$ denote the natural numbers and the real numbers and !, /, and $\sqrt{\ }$ denote the factorial, division, and square root functions, respectively.

Then $\sigma(2!) = \sigma((2/3)!) = \sigma((9/3)!) = \mathbf{N}$ by the syntax of !, and

- $2!\downarrow$,
- $(2/3)!\uparrow$, and

- $(9/3)! \downarrow$

by the semantics of ! and /. Similarly, $\sigma(\sqrt{4}) = \sigma(\sqrt{2/3}) = \sigma(\sqrt{-1}) = \mathbf{R}$ by the syntax of $\sqrt{\ }$, and

- $\sqrt{4} \downarrow \mathbf{N}$,
- $\sqrt{2/3} \downarrow$ but $\sqrt{2/3} \uparrow \mathbf{N}$, and
- $\sqrt{-1} \uparrow$.

by the semantics of / and $\sqrt{\ }$.

There are some important distinctions between types as they are normally used in computer science and sorts as they are used in IMPS. In computer science, "expression e has type α" means the value of e is in the set denoted by α. In addition, type systems usually satisfy the following two properties:

(1) e is assigned a type iff e is type correct (i.e., every operator in e is applied to an argument of the right type).

(2) Determining whether e is type correct is feasible.

For a sophisticated language of expressions, it can be very difficult to construct a type system that simultaneously satisfies both of these properties. There is a natural tension between them that makes type system design a nontrivial enterprise.

In the IMPS approach, "expression e has sort α" means, *if e is defined*, the value of e is in the set denoted by α. In contrast to the two properties for type systems are the following two properties for IMPS-style sort systems:

(1) Every expression e is assigned a sort (whether or not it is defined).

(2) Determining whether e is defined is undecidable.

There is no tension between achieving these two properties in an IMPS-style sort system since an expression does not need to be defined to be assigned a sort. The task of determining definedness in such sort systems—which is analogous to the task of determining type correctness in type systems—is thus separated completely from the task of assigning sorts to expressions.

6 Definedness Checking in IMPS

Because LUTINS does not assume that all functions are total and all expressions are defined, many questions about the definedness of expressions must be answered in the course of a proof. It is imperative that any system which implements a logic like LUTINS must provide automated support for checking the definedness of many expressions, for otherwise the user would be overwhelmed by the number of (mostly trivial) theorems that he or she would have to prove.

The algorithm in IMPS for definedness checking is embedded in the IMPS simplifier [13, 15], which automates most of the low-level reasoning—the kind of reasoning that the user would consider drudgework—that is done in IMPS. In addition to definedness checking, the simplifier performs arithmetic, algebraic, and order simplification; logical simplification; and the application of rewrite rules. The design of the simplifier is highly recursive. For example, algebraic simplification often requires definedness checking, and the definedness checking algorithm often makes calls to the top level of the simplifier.

A variety of theory-specific information is used by the simplifier when checking definedness:

- The sorts of expressions, particularly the sorts of variables and constants.
- The relationships between sorts.
- Facts about the domain and range of functions.
- Consequences of the local context of the definedness assertion that is being checked.

Although determining the definedness of an expression is an undecidable problem, definedness checking works quite well in IMPS: usually almost all the definedness checking required for an application of IMPS can be done automatically by the system.

As an example, consider the following definedness assertion:

$$\forall\, x, y : \mathbf{Z}, z : \mathbf{Q} \,.\, 2 < z \;\supset\; (x * y - 3!/z) \downarrow \mathbf{Q}.$$

Most of you would consider this assertion to be trivially true. However, a mechanical proof of this assertion requires the assemblage and application of a great many facts including:

- $\mathbf{N}$ is a subsort of $\mathbf{Z}$.
- $\mathbf{Z}$ is a subsort of $\mathbf{Q}$.
- $3 \downarrow \mathbf{N}$.
- ! is total on $\mathbf{N}$.
- $\mathbf{N}$ is closed under !.
- $*$ and $-$ are total on $\mathbf{Q} \times \mathbf{Q}$.
- $\mathbf{Q}$ is closed under $*$ and $-$.
- $\forall x, y : \mathbf{Q} \,.\, y \neq 0 \;\supset\; x/y \downarrow \mathbf{Q}$.
- $\forall x : \mathbf{Q} \,.\, 2 < x \;\supset\; x \neq 0$.

7 Partial NBG Set Theory

Von-Neumann-Bernays-Gödel set theory (NBG) is a well-known (first-order) set theory in which variables range over both sets and proper classes. This means that the universe of sets V can be defined as an individual constant in NBG even though it is a proper class. Also functions from V to V, such as the cardinality function, are first-class objects in NBG even when they are proper classes. (A good introduction to NBG is found in [29].)

NBG is closely related to Zermelo-Fraenkel set theory (ZF), the most popular formalization of set theory. NBG and ZF share the same intuitive model of the iterated hierarchy of sets. The nonlogical axioms of NBG are very similar to those of ZF; most of them are simply ZF axioms with some of the quantifiers restricted to sets. And there is a faithful interpretation of ZF in NBG [31, 34, 38], which implies that ZF is consistent iff NBG is consistent. However, NBG is finitely axiomatizable, while ZF is not. (See [17] or [29] for a proof of this celebrated fact about NBG.)

NBG provides strong support for reasoning about (partial and total) functions *in theory*. I will show how to formalize the traditional approach to partial functions in NBG.[3] The resulting system named NBG* provides strong support for reasoning about functions *in practice*.

[3]To simplify the discussion I will consider only unary functions.

I will construct NBG* in three steps. First, the underlying logic of NBG, ordinary first-order logic, is replaced with PFOL. The nonlogical axioms of NBG* are exactly the nonlogical axioms of NBG.

Second, term constructors for (unary) function application and lambda-abstraction are defined as follows:

- $f(a) \equiv \mathrm{I}\, b \,.\, \mathrm{fun}(f) \wedge \langle a, b\rangle \in f$.
- $\lambda\, x \,.\, t \equiv \mathrm{I}\, g \,.\, \mathrm{fun}(g) \wedge \forall\, x \in V \,.\, \mathrm{if}(t \in V, g(x) = t, g(x)\uparrow)$.

Here f, a, t are terms, x is a variable, b does not occur in f or a, and g does not occur in x or t. Also, $\mathrm{fun}(f)$ means f is a function, i.e., a class of ordered pairs such that, if $\langle a, b\rangle, \langle a, b'\rangle \in f$, then $b = b'$. These constructors, along with the definite description operator I, allow one to build complex terms in NBG* that denote classes.

Third, a LUTINS-style sort system is added. The sort system has a hierarchy of sorts similar to that of LUTINS, containing both atomic sorts and compound sorts of the form $\alpha \rightarrow \beta$. A sort denotes a nonempty domain of classes. (A domain of classes can be a set, a proper class, or a collection containing at least one proper class.) A compound sort $\alpha \rightarrow \beta$ denotes the domain of partial functions from the sets in the domain denoted by α to the sets in the domain denoted by β. Of course, if α and β denote sets, then the compound sort $\alpha \rightarrow \beta$ will also denote a set. Two very useful atomic sorts are **C**, the domain of all classes, and **V**, the domain of all sets. **C** is the maximum sort: every sort is a subsort of it. Every sort that denotes a class is a subsort of **V**, and every compound sort is a subsort of $\mathbf{V} \rightarrow \mathbf{V}$.

As a candidate logic for a mechanized mathematics system, NBG* has the following strengths:

- It is based on ideas and principles that are very familiar from mathematical practice.
- It has the same expressive power as ZF.
- It has the same convenient machinery for reasoning about functions as LUTINS.
- The simplification and definedness checking algorithms in IMPS for LUTINS can be used for NBG* with no major conceptual changes.
- LUTINS theories can be directly translated into NBG* theories.

On the other hand, NBG* has three fairly obvious weaknesses. First, it would be a very major undertaking to implement a mechanized mathematics system based on NBG*. In order to be useful to a wide range of people, the system would need a variety of mechanisms to bring the level of reasoning from the microscopic level of formal sets to a more human level comparable to that used in mathematical practice. Many of these could be borrowed from IMPS, but they would have to be fine-tuned for the new system and its new logic.

Second, arbitrary terms in NBG* are not allowed to be sorts. The problem here is that an arbitrary term may denote the empty set. Allowing sorts that may denote the empty set would greatly complicate the logic, its implementation, and the use of a mechanized mathematics system based on it. For example, variables would no longer be defined for free; they would only be defined when the sort of the variable denoted a nonempty class—which could vary from one context to another.

And third, the set/class distinction, which is a fundamental principle of NBG*, has a marginal status in everyday mathematical practice—particularly outside the community of professional mathematicians. For this reason, one could argue that NBG* does not conform to mathematical practice as successfully as LUTINS does.

8 Conclusion

I have argued that the traditional approach to partial functions can be formalized in a range of familiar formalisms without sacrificing the underlying intuition and semantics. The new machinery of these formalisms allows one to reason about (partial and total) functions in a natural and direct way. The IMPS system demonstrates that this partial functions machinery can be effectively implemented.

Some brief remarks on the other addresses. Three addresses were given at the Partial Functions and Programming: Foundational Questions conference in addition to the one presented in this paper. In the address entitled "Definedness", Solomon Feferman distinguished "logics of existence" from "logics of definedness". The formalisms that I have discussed—as well as most other formalizations of the traditional approach to partial functions—are logics of definedness. The logic that David Parnas presented in "A Logic for Describing, not Verifying Software" is essentially the same as PFOL without the definite description operator I. And Dana Scott pointed out in his address "Partial Functions and Type Theory" that the notion of a partial function used in my talk admits partial functions with arbitrary domains and ranges. He noted that there are other notions of a partial function—such as the partial recursive function and the partial function in intuitionistic type theory—in which domains and ranges are restricted to certain kinds of sets.

Acknowledgments

I would like to thank Dr. Joshua Guttman and Dr. Leonard Monk for their suggestions on how to improve the paper.

References

[1] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[2] M. Beeson. Formalizing constructive mathematics: Why and how? In F. Richman, editor, *Constructive Mathematics: Proceedings, New Mexico, 1980*, volume 873 of *Lecture Notes in Mathematics*, pages 146–190. Springer, 1981.

[3] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer, Berlin, 1985.

[4] T. Burge. *Truth and Some Referential Devices*. PhD thesis, Princeton University, 1971.

[5] T. Burge. Truth and singular terms. In K. Lambert, editor, *Philosophical Applications of Free Logic*, pages 189–204. Oxford University Press, 1991.

[6] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer, 1991.

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

[8] R. L. Constable. Partial functions in constructive formal theories. In A. B. Cremers and H. P. Kriegel, editors, *Theoretical Computer Science*,

volume 145 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1982.

[9] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In A. K. Chandra, editor, *Symposium on Logic in Computer Science (Proceedings)*, pages 183–193. Computer Society of the IEEE, 1987.

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

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

[12] W. M. Farmer. Theory interpretation in simple type theory. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer, 1994.

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

[14] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 1993. Available at `http://imps.mcmaster.ca/`.

[15] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Contexts in mathematical reasoning and computation. *Journal of Symbolic Computation*, 19:201–216, 1995.

[16] S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. *Contemporary Mathematics*, 106:101–136, 1990.

[17] K. Gödel. *The Consistency of the Axiom of Choice and the Generalized Continuum Hypothesis with the Axioms of Set Theory*, volume 3 of *Annals of Mathematical Studies.* Princeton University Press, 1940.

[18] T. Hailperin and H. Leblanc. Non-designating singular terms. *Philosophical Review*, 68:239–243, 1959.

[19] J. Hintikka. Existential presuppositions and existential commitments. *Journal of Philosophy*, 56:125–137, 1959.

[20] A. Hoogewijs. Partial-predicate logic in computer science. *Acta Informatica*, 24:381–393, 1987.

[21] 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, 1994.

[22] J. Kuper. An axiomatic theory for partial functions. *Information and Computation*, 107:104–150, 1993.

[23] K. Lambert. Existential import revisited. *Notre Dame Journal of Formal Logic*, 4:288–292, 1963.

[24] K. Lambert, editor. *Philosophical Applications of Free Logic.* Oxford University Press, 1991.

[25] K. Lambert and B. C. van Fraassen. *Derivation and Counterexample.* Dickenson Publishing, Encino, California, 1972.

[26] H. S. Leonard. The logic of existence. *Philosophical Studies*, 4:49–64, 1956.

[27] F. Lepage. Partial functions in type theory. *Notre Dame Journal of Formal Logic*, 33:493–516, 1992.

[28] F. Lucio-Carrasco and A. Gavilanes-Franco. A first order logic for partial functions. In B. Monien and R. Cori, editors, *STACS 89*, volume 349 of *Lecture Notes in Computer Science*, pages 47–58. Springer, 1989.

[29] E. Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, 1964.

[30] L. G. Monk. PDLM: A Proof Development Language for Mathematics. Technical Report M86-37, The MITRE Corporation, Bedford, Massachusetts, 1986.

[31] I. L. Novak. A construction for models of consistent systems. *Fundamenta Mathematicae*, 37:87–110, 1950.

[32] D. L. Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19:856–861, 1993.

[33] N. Rescher. On the logic of existence and denotation. *Philosophical Review*, 68:157–180, 1959.

[34] J. B. Rosser and H. Wang. Non-standard models for formal logics. *Journal of Symbolic Logic*, 15:113–129, 1950.

[35] R. Schock. *Logics without Existence Assumptions.* Almqvist & Wiksells, Stockholm, Sweden, 1968.

[36] D. S. Scott. Existence and description in formal logic. In R. Schoenmann, editor, *Bertrand Russell: Philosopher of the Century*, pages 181–200. Allen and Unwin, London, 1967.

[37] D. S. Scott. Identity and existence in intuitionistic logic. In M. P. Fourman et al., editor, *Application of Sheaves: Proceedings, Durham 1977*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer, 1979.

[38] J. Shoenfield. A relative consistency proof. *Journal of Symbolic Logic*, 19:21–28, 1954.

[39] T. Smiley. Sense without denotation. *Analysis*, 20:125–135, 1960.