Mechanizing the Traditional Approach to Partial Functions[⋆]

William M. Farmer

The MITRE Corporation, 202 Burlington Road, Bedford, MA 01730-1420, USA

Abstract. In traditional mathematics it is legitimate to apply a (partial) function to an argument outside of its domain, but the resulting term is treated as having no value. Moreover, the informal logic of traditional mathematics is two-valued despite the presence of nondenoting terms. This paper shows how this traditional approach to partial functions can be implemented in mechanized mathematics systems.

1 Introduction

Partial functions are ubiquitous in both mathematics and computer science. It is therefore essential that a mechanized mathematics system provide good mechanical support for reasoning about partial functions. However, there does not yet exist a consensus on how partial functions should be mechanized. The developer of a mechanized mathematics system must choose among many different possible ways of representing and reasoning about partial functions—most of which have serious drawbacks.[2]

Given the state of partial functions in mechanized mathematics, it is a bit surprising that partial functions are not at all a problem in mathematics practice. Mathematicians employ (and students are taught) a simple, straightforward way of dealing with partial functions—what we will call the *traditional approach to partial functions*. Even though the traditional approach is well established in mathematics practice, very few contemporary mechanized mathematics systems support it. One major exception is IMPS, an Interactive Mathematical Proof System [6] developed by William M. Farmer, Joshua D. Guttman, and F. Javier Thayer Fábrega.

This paper describes the traditional approach to partial functions; illustrates how predicate logic can be modified to support it; and discusses mechanisms for implementing formalisms that support the traditional approach.

[⋆] Supported by the MITRE-Sponsored Research program. This paper is an abbreviated version of [4] which was presented at the Workshop on the Mechanization of Partial Functions, CADE-13, Rutgers University, New Brunswick, New Jersey, July 30, 1996.

[2] See [4] for references and [1] for a comparison of the common approaches to partial functions.

2 The Traditional Approach

In informal mathematics 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. Strictly partial functions are abundant in both mathematics and computer science. 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.

As we have just seen with division, partial functions—unlike total functions—may yield undefined applications. Suppose a partial function f is applied to an argument $a \in D^*_f$ such that $a \notin D_f$. The application $f(a)$ would then be undefined. The whole problem of partial functions comes down to what status should be granted to an undefined application like $f(a)$. Should $f(a)$ be considered a well-formed expression? Should it be assigned a value? Should the value be different from the values assigned to defined applications?

The traditional approach to partial functions deals with the problem of undefined applications in a very direct way. The approach is based on three principles:

1. Variables and constants are always defined, i.e., they always denote something.
2. Functions may be partial. An undefined application (e.g., $1/0$) is a well-formed expression which is not assigned a value. By convention, 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., a truth-valued function) is always defined. By convention, 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).

We 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 high school and college.

3 Partial First-Order Logic

In this section we introduce a variant of first-order logic called Partial First-Order Logic (PFOL) to illustrate how predicate logic can be modified to support the traditional approach. A more detailed presentation of PFOL is given in [5]. Several systems similar to PFOL have been presented in the literature; see [4] for references.

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 partial: undefined definite descriptions like $\mathrm{I}\, x \,.\, x \neq x$ and applications of the form $f(a)$ where the value of a is outside the domain of the value of f do not have values in the model.

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.

Proof See [5]. $\square$

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*[3] in which there is some mechanism for reasoning about nonexistent entities such as the present king of France.

In a similar way, other formalisms can be modified to support the traditional approach. For example, the logic of IMPS, LUTINS [1, 2, 3], is a variant of simple type theory which supports the traditional approach. See [4, 5] for an example of how set theory can be modified to support the traditional approach.

4 Sorts

Sorts are syntactic objects similar to types that are used to organize terms in a logic that supports the traditional approach. A *sort system* consists of a collection of *atomic* and *compound* sorts. For a logic like PFOL, a compound sort would have the form $\alpha_1 \times \cdots \times \alpha_n \to \alpha_{n+1}$ where $\alpha_1, \ldots, \alpha_{n+1}$ are atomic sorts. An atomic sort α denotes a nonempty domain D_α of values, and a compound sort $\beta = \alpha_1 \times \cdots \times \alpha_n \to \alpha_{n+1}$ denotes the domain D_β of partial functions from $D_{\alpha_1} \times \cdots \times D_{\alpha_n}$ to $D_{\alpha_{n+1}}$.

A sort α is a *subsort* of β if $D_\alpha \subseteq D_\beta$. For example, suppose $\mathbf{Z}$ and $\mathbf{R}$ are atomic sorts that denote the integers and reals, respectively. $\mathbf{Z}$ is thus a subsort

[3] There is a substantial literature on free logic; see [4] for references.

of $\mathbf{R}$. Also, $\mathbf{Z} \rightarrow \mathbf{Z}$, which denotes the domain of partial functions from the integers to the integers, is a subsort of $\mathbf{R} \rightarrow \mathbf{R}$, which denotes the domain of partial functions from the reals to the reals.

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

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

Second, every term t is assigned a sort $\sigma(t)$ on the basis of its syntax. Variables and constants are assigned sorts when they are specified. A term $\mathrm{I}\, x \,.\, t$ is assigned the sort $\sigma(x)$. And an application $f(t_1, \ldots, t_n)$ is assigned the sort α_{n+1} if $\sigma(f) = \alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$. $\sigma(t) = \alpha$ means, *if t is defined*, the value of t is a member of D_α. That is, if a term is defined, its assigned sort gives some immediate information about its value—which is very useful to both the human user and the computer.

Sorts are discussed in more detail in [2, 4, 5, 6]; [2] and [5] present sort systems for a partial simple type theory and a partial set theory, respectively.

5 Definedness Checking

In a logic like PFOL that does not assume that all functions are total and all terms are defined, many questions about the definedness of terms must be answered in the course of a proof. It is imperative that any system which implements such a logic must provide automated support for checking the definedness of many terms, 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 [6], 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 terms, 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 a term 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.

6 Conclusion

The traditional approach to partial functions has been well tested in mathematics practice, and it has been effectively implemented in the IMPS theorem proving system. We believe that, using the ideas described in this paper, the traditional approach can be advantageously employed in a wide range of other mechanized mathematics systems.

References

1. W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.
2. W. M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.
3. W. M. Farmer. Theory interpretation in simple type theory. In J. Heering et al., editor, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, 1994.
4. W. M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43:279–294, 1995.
5. W. M. Farmer and J. D. Guttman. A set theory with support for partial functions. In E. Thysse and F. Lepage, editors, *Partial, Epistemic, and Dynamic Logic*, Applied Logic Series. Kluwer. Forthcoming.
6. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

This article was processed using the LaTeX macro package with LLNCS style