Undefinedness in Z: Issues for Specification and Proof

R.D. Arthan

International Computers Ltd.
Lovelace Road,
Bracknell,
Berkshire. UK.
e-mail: rda@win.icl.co.uk

Abstract. This paper considers the treatment of undefined terms in the Z specification language. We argue, on pragmatic grounds, that specification and proof are activities which place conflicting requirements on the handling of undefinedness. We believe that the conflict can be reconciled by encouraging specifications that are independent of the treatment of undefined terms and by gaining a better understanding of the metatheory of undefinedness.

1 Introduction

Mathematical specification languages such as the Z notation [11] are becoming more widely used in the development of critical systems. A particular advantage of Z is (or should be) its familiar mathematical foundations. From the point of view of mathematical logic, the language as defined in [10, 11] or in the evolving Z standard can fairly readily be explicated as classical set theory subjected to a simple type discipline.

Providing effective tools for carrying out proofs in Z is an issue of some practical significance. For over six years, our group in ICL has been concerned with developing and using a suite of tools, known as ProofPower that, *inter alia*, provide a theorem-prover for Z. ProofPower supports Z via a so-called semantic embedding into (a re-engineered version of) the HOL theorem-prover.

An area which has long been a source of contention in the Z community has been the treatment of undefined terms. Given the possibility of undefined terms, one must ask how undefinedness propagates through the various language constructs. These issues must be settled one way or another if formal proofs in Z are to be carried out.

We will argue that, at least in an ideal world, the subtleties of different treatments of undefinedness should not be a central concern for people writing specifications. In our experience, most real-life specifications that do make essential use of undefined terms are just wrong — they do not say what their author intended. Even when the specification is free of such errors, there are grounds for concern about the methodological significance of proofs which depend essentially on undefined terms. We consider measures that might be taken to allay these concerns.

2 The Classical Approach to Undefinedness

Traditional — rigorous, but informal — pure mathematics generally takes an unsystematic, but effective, approach to the treatment of undefined terms. The mathematical literature contains many definitions whose sense is dependent on implicit or explicit side conditions. Different authors have different approaches to the fine detail of their subject matter; common sense and "mathematical maturity" are frequently required to determine the scope of applicability of a given formula.

It is not the purpose of this paper to survey possible approaches to formalising the various conventions used in the mathematical literature for working with partial functions. A brief survey may be found in [2]. What we do wish to do here is to point out the *usual* way of handling the issue of undefined terms in formal treatments of first order set theory. Set theory provides an adequate formal foundation for the overwhelming majority of mathematics and its metatheory has been the subject of intensive study over the last 100 years or so; we should be able to rely on the classical foundation systems not to give us too many unpleasant surprises.

Without going into too many of the details, let us assume that we start to develop set theory taking membership and equality as primitive predicates. The usual notation for sets (pairing, product, comprehension etc.) can be introduced by conservative extension (see, for example, [7], [8] or the chapters on set theory in [1]); let us assume that that has been done, so that we can write expressions such as (x, y), $x \times y$, $\{x : y \mid \phi(x)\}$ and so on. Let us assume that functions are represented in the usual way as sets of pairs and that function application is denoted by a 2-place function symbol, App. $\mathsf{App}(f, x)$ is intended to denote the result of applying f to x; at least when f is a function and x is in its domain, $\mathsf{App}(f, x)$ must denote the unique y such that $(x, y) \in f$. $\mathsf{App}(f, x)$ is commonly written as $f'x$ in the literature, and we shall do so in the rest of this section.

Let us consider four textbook developments of the best known axiomatisations of set theory: Takeuti & Zaring [12] for **ZF**, Mendelson [8] for **NBG**, Kelley [6] for **MKM**, and Rosser [9] for **ML**. Standardising the notation[1], these works define application as shown in the following table:

Takeuti & Zaring	$f'x = \{z : \bigcup \mathsf{ran} f \mid (\exists y \bullet z \in y \land (x, y) \in f) \land (\exists_1 y \bullet (x, y) \in f)\}$
Mendelson	$f'x = \begin{cases} y \text{ if } \forall z \bullet (x, z) \in f \Leftrightarrow z = y \\ \emptyset \text{ otherwise} \end{cases}$
Kelley	$f'x = \bigcap \{y \mid (x, y) \in f\}$
Rosser	$f'x = \iota\{y \mid (x, y) \in f\}$

[1] Throughout this paper, I will follow the Z convention of taking the quantifiers to have low precedence: for example, $\exists_1 x \bullet \phi \land \psi$ is a unique existential quantification not a conjunction.

Here ι is Rosser's definite description operator (which I have slightly recast[2]). Rosser gives axioms for ι which, in the present context, amount to the single axiom: $\forall x \bullet \iota\{x\} = x$. We have mentioned definite description here, as that is what Rosser uses. Since definite description and function application are interdefinable[3], we will concentrate on function application in the sequel.

Thus: Takeuti & Zaring and Mendelson agree in defining the empty set as the value of an "undefined" application; Kelley's "undefined" applications are defined to take whatever value the intersection happens to have[4]; and Rosser's position is that "undefined" applications just take some unknown value (Rosser also suggests that a reader who is unhappy with this can simply add an axiom to fix that value, e.g., by taking it always to be π).

To sum up, the semantics of classical first order logic require function symbols such as App to be total on the universe of discourse. Naive set theory provides functions represented as sets; this introduces the possibility of undefinedness by application of a relation outside its domain or to an element of its domain for which it fails to be single-valued. Nonetheless, for mathematical purposes it is entirely adequate when giving a formal treatment of set theory to adopt one of several *ad hoc* conventions for assigning a more-or-less definite value to intuitively undefined terms. In the sequel, we will use the term "classical" to describe Rosser's approach, in which the value of an intuitively undefined term is left indefinite. This amounts to taking the defining axiom for application to be:

$$\forall f \bullet \forall x \bullet (\exists_1 y \bullet (x, y) \in f) \Rightarrow (\forall y \bullet f'x = y \Leftrightarrow (x, y) \in f)$$

3 Approaches to Undefinedness for Z

For our purposes, the Z language [11] can just be thought of as a gloss for the language of classical set theory on which is imposed a type discipline. The primitive predicates of the language are equality and membership, and there are expression[5] forms for the basic idioms of set theory: formation of tuples, finite sets, set comprehensions etc.

Types are constructed from ground types called "given sets" using a fixed repertoire of type constructors. The type constructors for the language represent

[2] Rosser introduces ι as a variable-binding operator. This requires him to introduce additional axioms to determine the behaviour of the new operator; these extra axioms are not required if we introduce ι as an ordinary function symbol.

[3] Rosser's formula shows how to define application in terms of description. For the converse, one can use $\iota(x) = (\{\emptyset\} \times x)(\emptyset)$.

[4] For example, if ω is the set of finite ordinals defined in the usual way, and if $L \subseteq \omega \times \omega$ is the restriction of the relation $\leq$ to ω, then, for Kelley, $L'i$ denotes the intersection of all finite ordinals not less than i, which is i, if i is a finite ordinal.

[5] We follow the Z usage for the word "expression" but not for the word "predicate": in this paper, $1 = 2 \vee \mathsf{false}$ is a proposition, $=$ is a predicate, and $1 + x$, 1 and (x, y) are all examples of expressions. We use the word "term" to cover both propositions and expressions in situations where the distinction is unimportant.

formation of power sets, n-fold cartesian products, and n-fold labelled product types referred to as "schema types". Since schema types have no bearing on the matter at hand, we will not further consider them.

The type discipline imposed renders illegal certain expressions and formulae which would be permitted in untyped set theory. This discipline enables mechanised tools to detect a wide range of common errors in the intended applications of Z for computer system specification. Defined notions such as the natural numbers, which could not be introduced via the standard set-theoretic construction under the type discipline, are introduced axiomatically using various well-defined mechanisms for introducing new constants.

The type discipline comprises rules for each form of expression and formula enabling one to decide its legality on the basis of the types of its constituents. These rules also allow one to infer the type of a legal expression. An example rule for an expression form might be that, if a, b and c are expressions with types α, β and γ respectively, then the triple (a, b, c) is legal and has type $\alpha \times \beta \times \gamma$. The rules for the primitive atomic formulae are that $a = b$ is legal iff. a and b have the same type and that $a \in b$ is legal iff. a has type α and b has type $\mathbb{P}\alpha$ for some type α.

Four possibilities for the treatment of undefined expressions in Z are discussed by Spivey in his original presentation of a semantics for Z [10]. A fifth option also presents itself in the notation Spivey uses to describe the semantics. The impact of the different options on a logic for Z was considered in some detail in unpublished work of Jones [5]. Let us briefly review the five approaches and assign mnemonics to them for later reference.

UPU This approach uses a three-valued logic, i.e., Undefined Propositions are Undefined: they take the value $*$, say, which is neither true nor false.

UPF This approach takes as false the value taken by a primitive predicate when either of its operands is undefined, i.e., Undefined (primitive) Propositions are False.

UED This is the classical approach of the previous section, i.e., Undefined Expressions Denote an unspecified value.

UPD This is the approach officially adopted in [11]: a predicate with an undefined operand takes one of the values true or false but one cannot determine which, i.e, Undefined Propositions Denote an unspecified value.

UEE This approach is suggested by the so-called "strong equality[6]" used in the semantic definitions in [10]. The strong equality $a \cong b$ holds iff. a and b are both defined and equal or both undefined, i.e. Undefined Expressions are Equal.

Within most of these approaches, there are still some detailed questions to be answered. In particular, one must consider how undefinedness propagates through the various expression forms. In the case of UPU, one must decide what formulation of three-valued logic to use.

[6] Unfortunately, this term is used with a variety of different meanings in the literature.

4 Issues for Specification

The merits and demerits of the various approaches to undefinedness in Z have been much debated since Spivey's original discussion of the question. Many of the proposed treatments focus on the conciseness of expression which UPF and UEE can offer. For example, UEE has the advantage that it can save one having to specify the domain of a partial function. To define a function f by asserting $\forall x : X \bullet f(x) = \mathcal{E}$ ensures that f has the same domain as the domain of definition of the expression $\mathcal{E}$ with little further ado.

Now, I would argue on pragmatic grounds that the labour saved by devices exploiting the details of UPF or UEE is not worth the cost. The primary purpose of the Z notation is to serve as an agreed means of communication between parties involved in the development of computer systems. Many readers may only have a passing acquaintance with the notation; all but the most expert readers will be ill-served by formal expositions which make use of devious tricks. There may be cases where a special treatment of undefined terms *significantly* reduces the complexity of a description, although that seems to me unlikely. In such cases, the richness of the rest of the Z notation ought to provide alternative idioms which give conciseness while maintaining clarity.

The other three approaches, UPU, UPD, and UED, give less cause for concern than UPF and UEE on methodological grounds. UPU has the pragmatic disadvantage that three-valued logic is much less familiar than classical logic and its rejection in [10, 11] is very understandable, given the goals of the Z notation. UED has the advantage of conformity with classical logic: every term denotes and laws such as the reflexivity of equality hold in full generality. UPD on the other hand has a greater intuitive appeal than UED for many people.

In practice, most specifications which do depend on the values of undefined terms are just wrong; like programs that can give rise to run-time errors, such specifications just do not say what the author intended. The following example shows an attempt at a Z schema to model a database look-up operation:

$$
\begin{array}{|l}
\hline
\mathsf{LookUp} \\
\mathsf{table} : KEY \pfun DATA \\
\mathsf{k?} : KEY \\
\mathsf{d!} : DATA \\
\hline
\mathsf{d!} = \mathsf{table}(\mathsf{k?}) \\
\hline
\end{array}
$$

Here the state component table, has been declared to be a partial function. The meaning of the schema clearly depends on the interpretation of $\mathsf{table}(\mathsf{k?})$ when the input $\mathsf{k?}$ takes a value outside the domain of table. In practice, this example would simply be incorrect in the majority of situations. The error can readily be remedied, either, explicitly, by adding an appropriate pre-condition, or, implicitly, by replacing $\mathsf{d!} = \mathsf{table}(\mathsf{k?})$ by $(\mathsf{k?}, \mathsf{d!}) \in \mathsf{table}$.

5 Issues for Proof

ICL have developed support for mechanised reasoning in Z within the ProofPower suite of tools. ProofPower is founded on a completely re-engineered implementation of the HOL theorem-proving system [3]. ProofPower uses exactly the same logic as HOL, namely M.J.C. Gordon's formulation of simple type theory with Milner-style polymorphism. The HOL logic can be viewed as a polymorphic, typed set-theory following very classical lines.

In this section I am going to describe briefly some of the issues which arise in trying to mechanise the various approaches to undefinedness for Z. Obviously, this description is influenced by our approach using a mapping into HOL. However, many of the issues would still arise in other approaches. For example, effective mechanised proof in Z undoubtedly requires algorithms to automate proofs in various problem domains. Since most research on automated reasoning has been for more-or-less classical logics, to exploit that research one must address many of the issues we discuss here.

Z is supported in ProofPower using a technique known as semantic embedding. M.J.C. Gordon [4] gives a good account of this type of technique in the context of logics for programming languages. In a semantic embedding, fragments of Z syntax are represented as semantically equivalent fragments of HOL syntax. Customisations to the parser and pretty-printer ensure that Z can generally be entered and displayed using Z syntax, so that the embedding is largely invisible to the user. The semantics for Z implemented by the resulting system is determined by the definitions of HOL constants which support each Z construct. For example, in ProofPower, the semantics of function application for Z is carried by an HOL constant Z'App; a Z application $f\,x$ is translated into the HOL term Z'App $f\,x$. The definition in HOL of Z'App then determines the meaning of this term.

The HOL logic takes the classical approach to undefinedness. The way HOL is axiomatised means that any well-typed term denotes some unspecified member of its type. A more discriminating treatment of undefined terms can be modelled within a semantic embedding by including error-values in the HOL types which represent Z types. To show how this works, I must explain that the HOL type system can be extended by the user, or by the programmer on the user's behalf. An HOL type is either a type variable, or is formed by applying a type constructor to zero or more argument types. The primitive type constructors include a nullary (i.e., constant) type constructor, BOOL, to represent the truth-values, and a two-place type constructor $\rightarrow$ for forming the set of all functions between two types. By means of a conservative extension principle, new type constructors can be defined to represent sets, products, lists and so on. The following equations define a natural representation of a Z type, σ, say, as an HOL type $\overline{\sigma}$.

$$\begin{array}{rcl} \overline{G} & = & G \\ \overline{\mathbb{P}\tau} & = & (\overline{\tau})\mathbb{P} \\ \overline{\tau_1 \times \ldots \times \tau_k} & = & (\overline{\tau_1}, \ldots, \overline{\tau_k})\Pi_k \end{array}$$

Here, G denotes a Z given set, which we take to be represented by a nullary HOL type constructor of the same name. $\mathbb{P}$ on the right-hand side of the second equation denotes the type constructor used for sets in HOL, and Π_k denotes a k-fold cartesian product. Note that HOL type constructors are written after their arguments.

A type constructor such as $\mathbb{P}$ is introduced by a mechanism allowing a new type to be defined to be in 1-1 correspondence with a subset of an existing type ($\alpha \rightarrow \mathsf{BOOL}$ in the case of $(\alpha)\mathbb{P}$). The HOL logic does not support parameterised families of types such as the Π_k directly. However, one can write a program which will introduce the definition of each particular Π_k as and when it is needed.

If we need to allow for an error-value $\perp$ in each type, then our representation in HOL becomes a little more complicated, and is best defined using the above representation $\overline{\tau}$ as an auxiliary, say:

$$\overline{\overline{\tau}} \quad = \quad \overline{\tau} \sqcup \mathsf{UNDEF}$$

where UNDEF is an HOL type with exactly one element, $\perp$, say, and where $\sqcup$ is a type constructor which forms disjoint unions.

The representation of a Z type τ as $\overline{\overline{\tau}}$ does not seem much more complicated than its representation as $\overline{\tau}$. However, when we consider the corresponding representations of propositions and expressions, the complexity builds up very rapidly. For example, let us contrast the representation of a simple membership assertion under UED (the classical treatment, for which the simple representation of types will do) and under UPF (for which we have to distinguish between undefined and defined values so that "undefined" propositions can be made false):

$$\begin{array}{rcll} \overline{x \in A} & = & \overline{x} \in \overline{A} & \qquad\qquad\qquad\qquad \text{(UED)} \\ \overline{\overline{x \in A}} & = & \overline{\overline{x}} \neq \iota_2(\perp) \wedge \overline{\overline{A}} \neq \iota_2(\perp) \wedge \delta_1 \overline{\overline{x}} \in \delta_1 \overline{\overline{A}} & \qquad\qquad\qquad\qquad \text{(UPF)} \end{array}$$

Here, $\iota_i : \alpha_i \rightarrow \alpha_1 \sqcup \alpha_2$ $(i = 1, 2)$ is the injection of the i-th summand into the disjoint union and $\delta_i : \alpha_1 \sqcup \alpha_2 \rightarrow \alpha_i$ is its left inverse.

Since the representation $\overline{\overline{x \in A}}$ doubles the number of occurrences of $\overline{\overline{x}}$ and $\overline{\overline{A}}$ as compared with $\overline{x \in A}$, the increase in complexity could be considerable. Various devices could be introduced to ameliorate the problem, but there will always be a price to pay with the non-classical treatments; it is bound to be more complex if one must keep track of undefined terms.

In the light of the above sketch of how one might go about modelling the non-classical treatments of undefinedness classically, let us consider each of the five approaches to undefinedness in turn:

UPU To model three-valued logic requires us to use error-values for propositions as well as expressions. Reasoning has to be carried out at a metalevel. That is to say, we have to represent the judgments of the three-valued logic as members of some three-element type in HOL rather than as HOL propositions. This approach is technically feasible, but prevents our making direct use of the existing facilities for logical reasoning in HOL.

UPF Making primitive predicates with undefined operands take the value false requires error-values for expressions, but at least lets us conduct logical reasoning directly. A disadvantage is that the native HOL equality is not appropriate to represent the resulting Z equality. A new battery of facilities would have to be implemented to replace the tools the HOL system provides for equational reasoning.

UED The classical approach gives the best fit, requiring no error-values and allowing existing tools for logic and equational working to be used as they stand.

UPD The approach in which predicates with undefined operands take some indefinite truth-value gives rise to essentially the same problems as UPF, because one must still keep track of undefined expressions to implement a Z equality relation which admits the possibility that $f(x) \neq f(x)$ when $f(x)$ is not defined. This approach has the particular disadvantage that one has no direct general means of saying that an expression is well-defined, whereas, in UPF, for example, to assert $x = x$ is to assert that x is defined.

UEE The approach using the strong equality would require us to use error-values for expressions but would let us use existing facilities for equational reasoning, since the strong equality coincides with ordinary equality in the type augmented with an error-value.

The classical treatment, UED, was the approach of choice for our purposes. In fact, the definition of Z'App in ProofPower is (coincidentally) very like the formulation using a definite description operator that we have already discussed in connection with Rosser's exposition of **ML**. Given that there were many other much more pressing difficulties to overcome, we do not regret our decision to use the classical treatment of undefinedness in ProofPower. Most other attempts to define or implement a logic for Z of which I am aware have arrived at the same choice.

The above remarks have focussed on implementation issues. Nonetheless, they are reflected in issues for the user of the theorem-prover. Further problems arise both for implementors and users when one considers equational reasoning. The classical approach has the advantage that a valid equation $\mathcal{E}_1 = \mathcal{E}_2$, can be used without further ado as a rewrite rule: if we can instantiate the free variables of $\mathcal{E}_1 = \mathcal{E}_2$ to give say $\mathcal{E}_1' = \mathcal{E}_2'$, then it is classically a valid inference step to replace $\mathcal{E}_1'$ by $\mathcal{E}_2'$ in any formula containing $\mathcal{E}_1'$. So classically, we can use a law such as $x + 0 = x$, to simplify a formula such as $\forall i, j : \mathbb{Z} \bullet j \neq 0 \Rightarrow (i/j + 0)/j = i/j^2$ without having to consider the context in which $i/j + 0$ appears. In other approaches, we must do extra work to carry out the desired replacement. Care in the treatment of quantification *may* help in this regard, but the classical approach obviates the problem altogether.

6 The Way Forward

Methodologically, Spivey [11] seems to be right in trying to render specifications independent of the values of undefined terms. However, it would be useful for writers of Z to be able to check whether their specifications are actually affected by undefinedness. If we are to do formal proof, practical considerations make us veer towards a classical treatment of the issue. However, if we already had to hand a powerful theorem-prover supporting another treatment, we might prefer to use that instead.

It does not appear to be difficult to define proof obligations whose truth would ensure that a specification makes no essential use of undefined terms. Verification of such proof obligations would occupy the same methodological role as verifying consistency of the specification: it is an additional check that we have indeed said what we meant to say. Such an approach is analogous to the useful separation of concerns one can make in program verification by considering partial correctness and termination of the program independently. It remains to be seen whether the proof obligations can *conveniently* be formulated and proved using a logic which makes specific assumptions about undefined terms, but the problem does not seem to be theoretically intractable.

For proof, to allay the concerns of those who find the classical approach (or any of the others) repugnant, it would be highly desirable to have a better metatheoretic understanding of the problem. Since many variants on the classical approach have been intensively studied by mathematical logicians over the years, it would be rather surprising if there were any important facts which depended essentially on the detailed treatment of undefinedness. So we conclude this paper with a conjecture. This conjecture is somewhat informally stated, but can be made rigorous in first order set theory by introducing a predicate denoting definedness and considering possible equality and membership relationships which agree on defined terms. Let us call a formula ϕ *impartial* if ϕ makes no essential appeals to the values of undefined terms. We then conjecture that an impartial formula ϕ is provable (using any of the treatments of undefinedness we have discussed for two-valued logic) if and only if ϕ has a proof in which no essential appeals to undefinedness are made. If this conjecture is true, then it is largely irrelevant which particular logical treatment of undefinedness one chooses; if it is false, then the disproof should reveal something about the actual methodological significance of ones treatment of undefined terms.

Acknowledgments

I am grateful to Roger Jones for his help with the writing of this paper and to Roger and all my other past and present colleagues in the High Assurance Team at ICL for their work on ProofPower and its treatment of proof in Z.

The original development of ProofPower was jointly funded by the UK Department of Trade and Industry and ICL. Subsequent developments have been funded by ICL and by the Defence Research Agency, Malvern.

References

1. J. Barwise, editor. *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1977.
2. William M. Farmer. A Partial Functions Version of Church's Simple Theory of Types. *MITRE Corporation Technical Report M88-52*, 1990.
3. Michael J.C. Gordon. HOL:A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.
4. Michael J.C. Gordon. Mechanising Programming Logics in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Proceedings of the 1988 Banff Conference on Hardware Verification*. Springer-Verlag, 1988.
5. Roger Bishop Jones. Issues in the Semantics of Z and their Impact on Proof Rules for Z. *Unpublished Lecture Notes*, 1990.
6. John L. Kelley. *General Topology*. Springer-Verlag, 1955.
7. Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*. North Holland, 1980.
8. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadworth and Brook/Cole, third edition, 1987.
9. J. Barkley Rosser. *Logic for Mathematicians*. McGraw-Hill, 1953.
10. J.M. Spivey. *Understanding Z*. Cambridge University Press, 1988.
11. J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, 1992.
12. Gaisi Takeuti and Wilson M. Zaring. *Introduction to Axiomatic Set Theory*. Springer-Verlag, second edition, 1982.

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