On Partial-Function Application in Z

David A. Duffy

Department of Computer Science, University of York
Heslington, York, YO10 5DD
E-mail: dad@cs.york.ac.uk

Abstract

We discuss the application of partial functions to elements outside their domain in the context of the Z language and CADiZ tool. We illustrate some surprising results that can arise, and show that they may be readily justified, but also show that undesirable results can arise that are less readily resolved. We offer two possible resolutions, one that involves a simple change to, for example, free types, and another more contentious approach that involves, in particular, a new semantics for Z. We discuss the advantages and disadvantages of both approaches.

1 Introduction

Partial functions in Z [8] are simply relations with the property that if they “return a value” that value is unique. There is some debate as to the meaning of the application of such functions to arguments outside their domain. Arthan [2] outlines the main options and proposes a classical approach for reasoning purposes in which a function application in a specification Σ is assigned an arbitrary value from the function’s range in any particular interpretation (“pre-model”) for Σ . This semantics explains why, if f and g are partial functions from $\mathbb{N}$ to $\mathbb{N}$, then $f(0) = f(0)$ holds in all models, but $f(0) = g(0)$ need not even if both f and g are undefined for 0: $f(0)$ and $g(0)$ may be assigned different values from (the carrier set of) $\mathbb{N}$.

This semantics for partial-function application may or may not be embedded into the standard semantics for Z currently being developed; the standard semantics might assign the same range value to all “undefined” function applications of the same type. Any proof system that is sound with respect to Arthan’s proposed semantics is, of course, sound with respect to the more restrictive semantics. CADiZ [9], a tool developed at the University of York that supports analysis of Z specifications, is consistent with Arthan’s approach in that it accepts as valid $f(0) = f(0)$, but not $f(0) = g(0)$ (unless forced to by the axioms of the specification). In this paper, we take CADiZ as representative of proof systems for Z (such as ProofPower [5]), with regard to the treatment of partiality. Thus, Arthan’s approach provides the starting point for our investigation. However, we go a step further, and consider the scenario in which it is impossible to assign a range value to a function application in any model. In

other words, we consider the scenario in which adding a definedness condition such as $f(0) \in \mathbb{N}$ would make a specification unsatisfiable.

Apparently, this scenario has been investigated previously,¹ and there are unpublished notes on the subject, but in each case it was decided that any benefits accrued are outweighed by anomalous results. We argue that the partial-function applications of realistic specifications may have this property, that the apparently anomalous results they produce are not so disconcerting as they might at first seem, and that there is a major benefit from allowing them in that the specifier is no longer required to know the domain of a partial function in order to avoid contradictions when writing down axioms defining the function. However, we show that allowing such function applications can make a typical Z specification inconsistent, and we proceed to offer two possible resolutions to this problem, one involving modification of our specifications, and the other involving modification of our inference systems. We discuss the advantages and disadvantages of each approach.

The presentation of the paper is as follows. In the next section we discuss partial functions in a general setting, independently of Z , and describe a simple realistic specification in which the definedness problem arises. In Section 3, we then review partial functions from the Z standpoint, illustrating some surprising and undesirable properties that function application can have in Z , and show how the surprising results may be readily justified. We are then in a position to present in Section 4 our resolutions to the undesirable results. The second of these involves modification to the typical laws and inference rules implemented in reasoning systems for Z ; we specify the necessary changes (for a subset of Z) in Section 5, provide a semantics for our Z expressions in Section 6, and justify our proof rules with respect to this semantics in Section 7. We go on to discuss some of the consequences of both our resolutions in Section 8, including those that are apparently anomalous, and argue that these apparent anomalies are in fact reasonable. Finally, we present some conclusions on what we have done in Section 9, reviewing some of the advantages and disadvantages of our two approaches.

We are concerned almost exclusively with the semantics of function application in this paper, and how it interacts with the semantics of the rest of the system. For this reason, we will consider only a limited subset of Z , omitting discussion of, for instance, schemas; the subset should be sufficiently inclusive to demonstrate our approach in a non-trivial context. For similar reasons, we will also take only a semi-formal approach to the presentation of our semantics in Section 6. Typical current formal approaches translate each Z construct into a core language, and then define the semantics of the core language via a mapping into ZF set theory. We map each Z construct (of our subset) into an informal set-theoretic language with types. Our approach more closely resembles standard semantic definitions given in text books on logic. This approach was found not to work for the full Z language, principally due to the inclusion of schemas,² but it is adequate for our subset.

While we do not consider in detail Z -like languages such as B [1], we believe that our semantics should be applicable to them, perhaps with minor modifications. In the case of B itself, we must take account of a distinction that may

¹Sam Valentine, private communication

²Ian Toyn, private communication

be made between a well-formed and a well-typed formula. For instance, the formula $\forall x \bullet x + 0 = x$ is well-formed, but for type-checking purposes we must add the constraint $x \in \mathbb{N}$, say, to give $\forall x \bullet (x \in \mathbb{N} \Rightarrow x + 0 = x)$; we may say that this formula is well-typed. If we consider the well-typed language of B, then the application of our Z semantics to B is quite straightforward, since B has the same type system as Z. On the other hand, we consider later the possibility of including in Z formulae that are not well-typed; it might be argued that, in principle, B already allows such expressions.

2 What are partial functions

From the mathematical perspective, independently of Z, we may distinguish two types of partiality (cf. [11]).

Case 1: A function f is partial if it is not explicitly assigned any value for certain arguments. Viewed computationally, that is, viewing the “defining axioms” for f as, say, a set of rewrite rules (if the axioms are equations), f may or may not terminate on these arguments.

Example: Suppose we are given

$$\begin{aligned} \text{Spec1} = & \\ & [f, g, h : \mathbb{N} \rightarrow \mathbb{N} \mid \\ & \forall u : \text{nats} \bullet f(\text{succ}(u)) = u \\ & \forall v : \text{nats} \bullet g(\text{succ}(v)) = g(0) \\ & \forall w : \text{nats} \bullet h(0, w) = 0 \\ & \forall x, y : \text{nats} \bullet h(\text{succ}(x), y) = h(\text{succ}(x) \leftrightarrow y, y)]. \end{aligned}$$

In Spec1, f is not defined for 0 and g is not defined for any $v \in \mathbb{N}$. The function h is non-terminating for $x = 0, y = 0$ if its defining equations are treated as rewrite rules from left to right (as presented), and if, for t of type $\mathbb{A}$, $t \leftrightarrow 0$ simplifies to t ; since there are no other defining axioms for $h(\text{succ}(0), 0)$, h is undefined over these arguments — $h(\text{succ}(0), 0)$ could, however, be assigned the value 0, say, without inconsistency with Spec1 arising.

Case 2: A function f is partial if imposing definedness conditions on f results in inconsistency.

Example: Suppose we are given

$$\begin{aligned} \text{Spec2} = & \\ & [q : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \mid \\ & \forall w : \text{nats} \bullet q(0, w) = 0 \\ & \forall x, y : \text{nats} \bullet q(\text{succ}(x), y) = \text{succ}(q(\text{succ}(x) \leftrightarrow y, y))]. \end{aligned}$$

The partial function q may be interpreted as a “quotient” function over $\mathbb{N}$; the important property of Spec2 is that $q(\text{succ}(0), 0)$, representing $1/0$, is assigned no numeric value. Not only is q nonterminating for $x = 0, y = 0$, but we have $q(\text{succ}(0), 0) = \text{succ}(q(\text{succ}(0), 0))$, so if we add the definedness constraint $q(\text{succ}(0), 0) \in \mathbb{N}$, then we get $\exists z : \mathbb{N} \bullet z = \text{succ}(z)$ (contradicting the usual axioms for $\mathbb{N}$).

Case 1 is consistent with the standard Z-user view of partial functions in the sense that any function that satisfies Case 1 could be declared as partial in

a Z specification without any perceived inconsistency arising. This is the case that Arthan accounts for [2]. Case 2 is more contentious. It might be argued that any function (in a Z specification) satisfying Case 2 should be rejected as inconsistent with its declaration. According to this view, it should always be possible to assign to any function application some value from the function's *range* (that is, the set S' in the declaration $f : S \rightarrow S'$ for f). However, there are several counterarguments.

Firstly, *Spec2* describes what seems to be a reasonable specification in which the function q cannot return a numeric value for certain arguments, though its range is $\mathbb{N}$. Secondly, since we cannot bar non-termination, it would seem pointless (though not impossible for “semi-recursive” functions [6]) to bar Case 2. Thirdly, and perhaps most significantly of all, barring Case 2 would, from the perspective of function application, make redundant the declaration of functions as partial in Z, since a partial function $f : S \rightarrow S'$ would have the property

$$\forall x : S \bullet \exists_1 y : S' \bullet f(x) = y$$

(at least if S' is a type expression), exactly as for a total function. We discuss this fourth justification for Case 2 further in the next section.

3 Partial Functions and Z

In Z, a relation r with declaration $r : S \leftrightarrow S'$, for sets S and S' , is a subset of $S \times S'$: r has type $\mathcal{P}(T \times T')$, where $\mathcal{P}T$ and $\mathcal{P}T'$ are the types of, respectively, S and S' , and the defining axiom for r is $\forall p : r \bullet (p.1 \in S \wedge p.2 \in S')$. Partial and total functions, with declarations of the form $f : S \rightarrow S'$ and $f : S \rightarrow S'$, respectively, are relations with further restrictions expressed by axioms in the language of Z. A partial function f has the extra property

$$\forall p, q : f \bullet p.1 = q.1 \Rightarrow p = q,$$

while a total function $f : S \rightarrow S'$ is a partial function with the restriction that its domain is the whole of S . These definitions raise the question: what happens when a partial function is “applied” to an object outside of its domain. In fact, Z allows arbitrary relations to be applied as functions, and thus we cannot rely on the special property of partial functions in defining application.

The fundamental point to note is that the definition of relation imposes only the constraint of well-typedness on the application of a relation as a function, and the definition of partial function is only a little more restrictive. A relation application $r(s)$ is well-typed if r has type $\mathcal{P}(T \times T')$ and s has type T ; $r(s)$ then has type T' . Partial functions impose the additional constraint that if $(x, y) \in f$ is a consequence of the specification, then $f(x) = y$ is a consequence; the converse need not hold. Even total functions may be applied to objects outside their domain without introducing inconsistency. We illustrate these points with the following specifications.

$$\begin{aligned} \text{SpecA} = & \\ [f : \mathbb{N} \leftrightarrow \mathbb{N} \mid & \\ (1, 1) \in f & \\ f(1) = 2] & \end{aligned}$$

$$\begin{aligned}
& \text{SpecB} == \\
& [f : \mathbb{N} \leftrightarrow \mathbb{N} \mid \\
& f(\Leftarrow 1) = 0 \\
& f(2) = \Leftarrow 2]
\end{aligned}$$

$$\begin{aligned}
& \text{SpecC} == \\
& [f : \mathbb{N} \rightarrow \mathbb{N} \mid \\
& f(\Leftarrow 1) = \Leftarrow 1 \\
& f(\Leftarrow 2) = 2]
\end{aligned}$$

These specifications are “consistent” in Z in the sense that, from the axioms currently associated with relation and function declarations, we cannot derive a contradiction ($P \wedge \neg P$, for some P) via the inference rules normally associated with Z (as in CADiZ). The first two specifications illustrate the facts that a relation may be applied to objects (e.g. 1 in SpecA) for which it is not uniquely defined, and a partial function may be applied to (or return) objects outside its domain (or range), as in the first and second axioms of SpecB. As illustrated by SpecC, even a total function may be consistently applied to objects outside its domain, and, if so, may return an object outside its range. (We are not claiming that anybody would wish to write specifications such as that above, but if they are consistent in Z , then we should be able to offer, for instance, a semantical justification for them.)

The justification for these surprising properties of function application, that is, the consistency of the above specifications, is that the axioms given assert properties about each function application, not about the functions themselves. In particular, $f(a) = b$, where $f(a)$ and b are appropriately typed, does not imply $(a, b) \in f$. Formally, in the models of a specification, a function application $f(a)$ may be assigned a total function *App* [2]. To account for the fact that, if there is a unique b such that $(a, b) \in f$, then $f(a) = b$, we must impose constraints on *App* as in the following definition.

Definition: In a specification Σ , let f be an expression (representing an n -ary function) of type $\mathbb{P}(T_1 \times T_2)$, and let arg be an expression of type T_1 . If, in an interpretation for Σ , T_1 and T_2 are assigned, respectively, the (“carrier”) sets T_1^* and T_2^* , f is assigned the function f_I , and arg is assigned the value $a \in T_1^*$, then

$$\begin{aligned}
App(f_I, a) &= \text{the unique } z \text{ such that } (a, z) \in f_I \text{ if such a } z \text{ exists} \\
&= \text{some } z \text{ in } T_2^* \text{ otherwise.}
\end{aligned}$$

Thus, the value of a function application is not a value assigned to the function itself (that is, the function is not “totalised” — cf. [12]), but to the application operator. In fact, what we assign to any Z function symbol (partial or total) in a model is simply a relation satisfying the constraints imposed by Z ’s partial and total function axioms.

More problematic are specifications involving definitions such as that of q in the previous section. The problem becomes most apparent if we reexpress the definition in terms of a free type, as opposed to $\mathbb{N}$ (which is defined as a subset of the type $\mathbb{A}$ in standard Z). We thus introduce the free type *nats*, and define the functions *diff* and *quot* in terms of *nats*.

```

section SpecQ
nats ::= o | s(nats)
|-----
diff, quot : nats × nats → nats
|-----
∀ x : nats • diff(x, o) = x
∀ x, y : nats • diff(s(x), s(y)) = diff(x, y)
∀ y : nats • quot(o, y) = o
∀ x, y : nats • quot(s(x), y) = s(quot(diff(s(x), y), y)

```

The first line of *SpecQ* introduces a free type: a given set *nats* together with Peano-style axioms for *o* (denoting *zero*) and *s* (denoting *successor*). The axioms CADiZ associates with the free type *nats* have some (syntactic) differences from typical Peano axioms for $\mathbb{N}$ [4], but the induction principle is typical, and the property $\forall x : \textit{nats} \bullet \neg x = s\ x$ is a logical consequence.

The main difficulty with *SpecQ* is that it is provably inconsistent in CADiZ. We discuss a proof of its inconsistency further in the next section. We note here merely that we can prove $\textit{quot}(s\ o, o) \in \textit{nats}$ in CADiZ, and therefore that

$$\forall x, y : \textit{nats} \bullet \textit{quot}(s(x), y) = \textit{succ}(\textit{quot}(\textit{diff}(\textit{succ}(x), y), y))$$

contradicts the property $\forall x : \textit{nats} \bullet \neg x = s\ x$ of the free type *nats*.

There are several reasons why the inconsistency of *Q* is unfortunate. Firstly, the definition looks reasonable. Secondly, it is correct for the intended set, $\{x, y : \mathcal{N} \mid x > y \wedge y > 0\}$, where $\mathcal{N}$ denotes the natural numbers. Thirdly, the fact that the definition for *quot* is nonterminating for $\textit{quot}(s\ o, o)$ is consistent with our assumption that division of 1 by 0 is undefined (that is, is not a natural number). Fourthly, and perhaps most importantly, if we bar this definition, then this means that the Z specifier must know the domain of *quot*, before defining it by a set of equations, in order to avoid inconsistency; but in that case it would seem preferable that *quot* be defined as a total function from $\textit{nats} \times \textit{nats} \setminus \{o\}$ to *nats*. Thus, the utility of partial functions is substantially undermined if definitions such as that of *quot* are inconsistent.

4 Resolving the Inconsistency

We propose two resolutions to the inconsistency issue raised in the preceding section. The first is the more straightforward, but involves explicitly modifying specifications, while the second is more contentious, but leaves specifications unchanged, modifying instead the usual laws and inference rules for Z.

4.1 Modifying Free Types

The first solution is straightforward: the sets introduced by a free type (e.g. *nats* above) are not given type status, but are made subsets of types. In particular, we may associate with the set *nats* the type *nats*^{*}. Thus, *nats*^{*} becomes the given set, and *nats* is declared by *nats* : $\mathbb{P}\textit{nats}^*$. The other declarations and axioms for *nats* in the original free-type expansion remain unchanged.

Now any partial function $f : \textit{nats} \rightarrow \textit{nats}$ may return a value from *nats*^{*} when it is not specified or is unable consistently to return a value from *nats*. If we refer to as *nats*[−] the set $\textit{nats}^* \setminus \textit{nats}$, then *nats*[−] may be viewed as the “undefined values” of *nats*, though, as we indicate in Section 8 below, it is preferable to view them as “super values” of *nats*.

One incidental benefit of this process is that we may reintroduce the intuitive relationship between function application and membership. We may add as a property of any partial function over, say, *nats*, that

$$\forall x, y : nats \bullet f x = y \Rightarrow (x, y) \in f.$$

The problem with adding this property if *nats* is taken to be a type is that it collapses *f* into a total function. For if *nats* is a type, then we have

$$\forall x : nats \bullet \exists y : nats \bullet f x = y;$$

consequently, with the above property, we would have $\forall x : nats \bullet \exists y : nats \bullet (x, y) \in f$, making *f* total. However, if *nats** is the type and *nats* merely a subset, then we do not automatically have $\forall x : nats \bullet \exists y : nats \bullet f x = y$, and *f* remains total only over its *domain* (that subset of *nats* over which its application always returns a value of *nats*).

The problem with this approach is that the set *nats** must be explicitly included within the specification. Thus, either the specifier must build up the free type from scratch in order to include *nats** or the analysis tool must automatically add it and the declaration *nats* : $\mathbb{P} nats^*$. The first possibility clearly undermines the benefits of free types, while the second solution is likely to be disconcerting to the specifier, particularly if no partial functions are involved in the specification.

4.2 Modifying the Inference Rules

The alternative approach is more contentious in that it involves distinguishing between ‘having type *T*’ from ‘being a member of *T*’, in order to allow a well-typed function application to “return a value” not a member of its range type. For instance, if *nats* is a given set, *f* is a partial function from *nats* to *nats*, and *o* ∈ *nats* holds, then *f(o)* has type *nats*, but may or may not be in *nats*.

The advantage of this approach is that the set *nats** now becomes the carrier set of *nats* in a model, and thus becomes a metalogical concept; the specifier is now unable to refer directly to *nats**, and consequently is unable to make statements about all of its possible members. Moreover, specifiers need not even be aware of *nats** if they do not have partial functions in their specifications; the name *nats** may now, of course, be used freely within a specification.

The major disadvantage is that we must make some changes to the laws of Z and the inference rules of typical proof systems for Z such as CADiZ. However, these changes are not especially contrived; the new laws and rules coincide with our intuitions about the properties of sets. We believe that the changes also highlight some interesting properties of Z and CADiZ, and make the concept of a type more than just a way of avoiding meaningless expressions (such as $(0,1) = \{1,2,3\}$) and Russel-style paradoxes [8].

The required changes hinge on the fact that we can currently prove $f(o) \in nats$ in CADiZ if *nats* is a given set (or free type), if *o* ∈ *nats* is an axiom, and if *f* has declaration $f : nats \rightarrow nats$. Analogously we can prove $quot(s\ o, o) \in nats$, our earlier example. One such proof proceeds as follows.

Sketch of Proof:

Consider the set $Q == \{x, y : nats \mid true \bullet quot(x, y)\}$. First, we show, in

CADiZ, that $\text{Spec}Q \vdash \text{quot}(s\ o, o) \in Q$ — we omit $\text{Spec}Q$ in the following for brevity. This involves expanding $\text{quot}(s\ o, o) \in Q$ to $\exists x, y : \text{nats} \mid \text{true} \bullet \text{quot}(s\ o, o) = \text{quot}(x, y)$; x and y are then instantiated with $s\ o$ and o , respectively, via $\text{quantification tac}$, and we subsequently show $\vdash o \in \text{nats}$ and $\vdash s\ o \in \text{nats}$ (properties of the free type nats), and $\vdash \text{quot}(s\ o, o) = \text{quot}(s\ o, o)$, an axiom in CADiZ.

Now we show that $Q = \text{nats}$. This involves an expansion of set equality via the following `expansion` rule:

$$S = S' \implies \forall u : T \bullet u \in S \Leftrightarrow u \in S',$$

where S, S' are sets and $\mathbb{P}T$ is their type. Thus, we derive the subgoal

$$\vdash \forall u : \text{nats} \bullet u \in Q \Leftrightarrow u \in \text{nats}.$$

$u \in \text{nats}$ reduces (via CADiZ's absorption rule) to true , and another absorption reduces $u \in Q \Leftrightarrow \text{true}$ to $u \in Q$. We now apply $\forall$ elimination and then membership `expansion` to derive

$$u : \text{nats} \vdash \exists x, y : \text{nats} \mid \text{true} \bullet u = \text{quot}(x, y).$$

We may now instantiate x and y with, respectively, u and $s\ o$, and use the definitions of *diff* and *quot* and an induction over nats to complete the proof of this step.

Finally, we apply Leibniz to $\text{quot}(s\ o, o) \in Q$ via the equality $Q = \text{nats}$ to get $\text{quot}(s\ o, o) \in \text{nats}$, completing the proof of our original goal. ■

Since we have a proof that $\text{quot}(s\ o, o) \in \text{nats}$ holds in CADiZ, we cannot separate the carrier set nats^* into nats^+ (the elements of nats) and nats^- (the “undefined” values of type nats) and argue that the meaning of $\text{quot}(s\ o, o)$ in a model is some element of nats^- . Thus, if we are to take this line (formalised in Section 6), we must find a “flaw” in the above proof. We agree that $\text{quot}(s\ o, o) \in Q$ holds, and accept the application of Leibniz. This leaves the second part of the proof in which we apply the expansion rule for sets. We propose the following alternative expansion rule:

$$S = S' \implies \forall u : S \bullet u \in S' \wedge \forall u : S' \bullet u \in S.$$

Unfortunately, we can still prove $Q = \text{nats}$ via CADiZ's rule of normalisation:

$$\frac{x : \tau(x) \mid x \in t \vdash}{x : t \vdash},$$

where $\tau(x)$ denotes the type of x . The proof proceeds as follows.

Sketch of Proof:

Expanding $Q = \text{nats}$ via the new expansion rule we get

$$\forall u : Q \bullet u \in \text{nats} \wedge \forall u : \text{nats} \bullet u \in Q.$$

Via $\wedge$ and $\forall$ elimination we derive two subgoals:

$$u : Q \vdash u \in \text{nats}, \text{ and}$$

$$u : \text{nats} \vdash u \in Q.$$

The proof of the second subgoal proceeds via an inductive proof of $\forall u : \text{nats} \bullet u = \text{quot}(u, s\ o)$, as above, and we do not contend it. The proof of the first subgoal proceeds via two applications of normalisation:

$$u : \text{nats} \mid u \in Q \vdash u \in \text{nats}, \text{ then}$$

$u : nats \mid u \in nats, u \in Q \vdash u \in nats$,
which is an axiom. ■

Thus, in outline, we have proven $Q = nats$. Consequently, we must change the rule of normalisation to sustain the distinction between $nats^*$ and $nats^\top$: we replace $\tau(x)$ by t in the hypothesis; see Section 5. We are now unable to prove $Q = nats$, since we are unable to prove $u : Q \vdash u \in nats$, normalisation now reducing this merely to $u : Q \mid u \in Q \vdash u \in nats$.

We must show that these new expansion and inference rules do support the possibility of a term having a value not a member of its type. This may be done via a soundness proof for the rules with respect to an appropriate semantics. We present such a semantics and an outline of the soundness proof in Sections 6 and 7, respectively. First we must describe the syntax of the expressions we allow and present our inference rules more fully.

5 Syntax and Inference Rules

In this section we describe the subset of Z expressions that we are concerned with, and our expansion and inference rules relating to this subset. We refer to all allowed syntactic objects as ‘expressions’, distinguishing between ‘predicates’ (i.e. propositions) and ‘terms’, as opposed to ‘predicates’ and ‘expressions’.

5.1 The Syntax of Expressions

Types:

$$Type ::= TypeName \mid \mathbb{P}Type \mid Type \times \cdots \times Type$$

(TypeName denotes a given set.)

Declarations:

$$Decl ::= BasicDecl; \dots; BasicDecl$$

$$BasicDecl ::= DeclName, \dots, DeclName : Term$$

Each *DeclName* is said to be *introduced by the declaration*; its type is *T* if the type of *Term* is $\mathbb{P}T$.

Terms:

$$Term ::= Const \mid FreeVar \mid Tuple \mid TupleSel \mid Application \mid SetExp$$

Const is a name introduced by a declaration within a specification, while *FreeVar* is a free variable, a name introduced by either a declaration within a sequent or a quantification in a predicate (e.g. all occurrences of *x* in the *P* part of $\forall x : \mathbb{N} \bullet P$; all occurrences of *x* are said to be *bound* in $\forall x : \mathbb{N} \bullet P$).

$$Tuple ::= (Term, \dots, Term)$$

An *n*-tuple of terms $(t_1, \dots, t_n)$, where $n \geq 2$, may be abbreviated **t**; the type of $(t_1, \dots, t_n)$ is $T_1 \times \cdots \times T_n$, where T_i is the type of t_i .

$$TupleSel ::= t.i$$

where *t* a term of type $T_1 \times \cdots \times T_n$ for some *n*, and *i* is a value of $\mathbb{N}$ between 1 and *n*.

$$Application ::= f \ Term$$

where *f* is a Term of type $\mathbb{P}(T_1 \times T_2)$ and Term has type T_1 ; the type of the application is T_2 .

$SetExp ::=$
 $SetName \mid \{Term, \dots, Term\} \mid \{Decl \mid Pred\} \mid \{Decl \mid Pred \bullet Term\}$
 $\mid \mathbb{P} SetExp \mid SetExp \times \dots \times SetExp$

A $SetName$ is a declared name (possibly a $TypeName$) of type $\mathbb{P}T$ for some $Type T$. The elements of a set must all have the same type T ; the type of the set is then $\mathbb{P}T$.

Predicates:

$Pred ::=$
 $true \mid false \mid R Term \mid Term = Term \mid Term \in Term$
 $\mid \neg Pred \mid Pred \wedge Pred \mid Pred \vee Pred \mid Pred \Rightarrow Pred \mid Pred \Leftrightarrow Pred$
 $\mid \forall Decl \bullet Pred \mid \exists Decl \bullet Pred \mid \exists_1 Decl \bullet Pred$

where R is a relation symbol — $R Term$ is, of course, not an application. Such relational predicates may be represented with infix notation, as in $2 < 3$.

5.2 The Inference Rules

We describe our inference rules in terms of *sequents*, here expressions of the form $D \mid , \vdash \Delta$ — we omit generic parameters — where D is a declaration, and $, ,$ called the *antecedent(s)*, and Δ , called the *consequent(s)*, are finite lists (representing sets) of Z predicates separated by commas. Our rules are closely based upon those of CADiZ. However, apart from the necessary modifications of CADiZ's inference and expansion rules indicated above, we have simplified several of CADiZ's rules to take account of our more restricted language; in particular, since we have not included μ expressions in our subset, we combine into one inference CADiZ's expansion of a function application into a μ expression and its μ -tac inference rule.

We omit from our proof system explicit inference rules for $\Rightarrow, \Leftrightarrow, \exists$ and $\exists_1$, making the assumption that predicates involving these logical symbols are merely abbreviations for others involving only $\neg, \vee, \wedge, \forall$ and, in the case of $\exists_1, =$. As is usual in the descriptions of sequent proof systems for Z, we also omit from the sequents in our rules any information that is not relevant to the inference.

Our inference rules have either the form $\frac{H}{C}$, where H is a collection of sequents referred to as the *hypotheses* and C is a sequent referred to as the *conclusion*, or the form $\overline{C}$, denoting axioms (inference rules without hypotheses). We denote by $P(t)$ the fact that the predicate P contains the term t , and by $P(s/t)$ the replacement of t with s . A substitution of an n -tuple of terms $\mathbf{t}$ for an n -tuple of free variables $\mathbf{x}$ within a predicate P is denoted $P\{\mathbf{x} \rightarrow \mathbf{t}\}$.

Tautologies: We have as axioms

$$\overline{\mid False \vdash} \qquad \overline{\vdash True} \qquad \overline{\mid \alpha \vdash \alpha}$$

Declaration normalisation: For a set or type expression Ψ , we have

$$\frac{x : \Psi \mid x \in \Psi \vdash}{x : \Psi \vdash}$$

Function-application expansion: Let P be a predicate containing an application $f t$, let $\tau(s)$ denote the type of the term s , and let F abbreviate $(t, x) \in f$. We

have:

$$\frac{\vdash \exists_1 x : \tau(f t) \bullet F, P(f t) \quad | \exists_1 x : \tau(f t) \bullet F \vdash \exists x : \tau(f t) \bullet F \wedge P(x/f t)}{\vdash P(f t)}$$

$$\frac{| P(f t) \vdash \exists_1 x : \tau(f t) \bullet F \quad | \exists_1 x : \tau(f t) \bullet F, \exists x : \tau(f t) \bullet F \wedge P(x/f t) \vdash}{| P(f t) \vdash}$$

Equality reasoning: For $=$ we have an axiom and two pairs of inference rules:

reflection: $\vdash t = t$

commutation:

$$\frac{| s = t \vdash}{| t = s \vdash} \quad \frac{\vdash s = t}{\vdash t = s}$$

Leibniz:

$$\frac{| s = t, P(s/t) \vdash}{| s = t, P(t) \vdash} \quad \frac{| s = t \vdash P(s/t)}{| s = t \vdash P(t)}$$

Connectives:

$\neg$ introduction:

$$\frac{\vdash P}{| \neg P \vdash} \quad \frac{| P \vdash}{\vdash \neg P}$$

$\vee$ introduction:

$$\frac{\vdash P, Q}{\vdash P \vee Q} \quad \frac{| P \vdash \quad | Q \vdash}{| P \vee Q \vdash}$$

$\wedge$ introduction:

$$\frac{\vdash P \quad \vdash Q}{\vdash P \wedge Q} \quad \frac{| P, Q \vdash}{| P \wedge Q \vdash}$$

$\forall$ quantifier:

$\forall$ introduction:

$$\frac{D' \vdash P}{\vdash \forall D \bullet P},$$

where D' is D with any variables renamed to avoid clashes with names introduced by declarations in the specification or in the conclusion.

$\forall$ substitution:

$$\frac{| \forall D \bullet P \wedge Q \Rightarrow P'\{\mathbf{x} \rightarrow \mathbf{t}\} \vdash}{| \forall D \bullet P \vdash},$$

where $\mathbf{x}$ is the n -tuple of variables introduced by $Decl$, $\mathbf{t}$ is an n -tuple of terms, Q is a conjunction of memberships $t_i \in S_i$ for each declaration $x_i : S_i$ in D , and P' is P with any bound variables renamed to avoid clashes with names occurring in $\mathbf{t}$. (This rule combines CADiZ's quantification `tac` tactic, `onpoint` rule, and `absorption` rules.)

The inference rules are applied backwards in CADiZ to reduce a goal to subgoals. Thus, in particular, the introduction rules become elimination rules.

Expansions: An expansion rule $X \Longrightarrow Y$ is a law that allows any instance of X to be replaced by Y . We assume that these laws are meta-level (expressed

outside Z), like the inference rules, but always produce Z expressions. Our expansion rule for memberships are exactly as in CADiZ; for example, $S \in \mathbb{P} S'$, for set expressions S and S' , may be expanded to $\forall x : S \bullet x \in S'$. However, we replace CADiZ's expansion rule for set equality as indicated in Section 4.2.

This new expansion rule is, of course, entirely reasonable, as is our rule of normalisation; it is unfortunate that set equality cannot be defined as equivalence of membership (say, $\forall x \bullet x \in S \Leftrightarrow x \in S'$), as in classical set theory [4], but then we would need a type for x , and this would take us back to CADiZ's definition. More importantly, when new concepts — and corresponding expansion rules — are introduced, such as $\subseteq$, we must be careful to ensure that they satisfy the expected properties. For example, $\subseteq$ should be defined by a law with the same expansion as membership of $\mathbb{P}$. If, instead, it is defined by

$$S \subseteq S' \Leftrightarrow \forall x : T \bullet x \in S \Rightarrow x \in S',$$

where $\mathbb{P} T$ is the type of S and S' , then we will no longer have

$$S \in \mathbb{P} S' \Leftrightarrow S \subseteq S'$$

as a consequence of our proof system (though CADiZ's unchanged normalisation rule would allow this equivalence to be proven). For example, we would have (trivially) $\{f o\} \subseteq nats$ (assuming $f o$ has type $nats$), but not necessarily $\{f o\} \in \mathbb{P} nats$. We will discuss this issue further in Section 8.

6 The Semantics of the Subset

In this section we will describe what we believe to be an intuitively straightforward and natural semantics for the expressions of our subset of Z . This semantics provides a meaning for terms that may not be assigned a value of their type. We will subsequently show that our expansion and inference rules are sound with respect to this semantics.

As mentioned previously, since our principal goal is merely to describe the meaning of partial-function applications in the context of a substantial subset of Z , our semantic “target language” will be an informal set-theoretic one, which itself includes disjoint “types” $T_1^*, T_2^*, \dots$ for each Z type expression $T_1, T_2, \dots$. We do not concern ourselves with the interpretation of such a language, in particular, the sets $T_1^*, T_2^*, \dots$, in terms of, say, ZF set theory, this being a problem to be resolved by the Z standards committee.

In our semantics, we follow the “classical” approach discussed by Arthan [2], where undefined expressions are assigned some element of the appropriate type. However, in order to allow a well-typed function application to “return a value” not a member of the function's range type, we distinguish ‘having type T ’ from ‘being a member of T ’. We formalise this as follows.

Let Σ be a well-typed specification involving a type T . In an interpretation for Σ we assign a set T^* to T , constructed according to the form of T as follows.

- If T is a given set, then T^* is the union of two (possibly empty) disjoint sets $T^{*+} \cup T^{*-}$.
- If T is a power set $\mathbb{P} T_1$, then $T^* = \mathbb{P}(T_1^*)$.

- If T is a product $T_1 \times \dots \times T_n$, then $T^* = T_1^* \times \dots \times T_n^*$.

Note that $+$ and $\Leftrightarrow$ are operators applied to the set T^* to produce a subset; it is not strictly necessary to treat $+$ and $\Leftrightarrow$ as operators — we might have defined T^* to be $T^+ \cup T^-$ — but the operator view simplifies the presentation of the meaning of membership ‘ $\in$ ’ below. However, we will often abbreviate T^{*+} and T^{*-} to, respectively, T^+ and T^- . Informally, T^+ and T^- may be interpreted as, respectively, the defined values of T and the undefined values of type T . It is preferable, however, to view elements of T^- not as undefined values, but as, say, “super values” of type T , since they may have various properties assigned to them; for example, if $f(0)$ is an “undefined” expression of type $\mathbb{A}$, and g is a function from $\mathbb{A}$ to $\mathbb{A}$, then we may have $g(f(0)) = 0$, without inconsistency arising.

Any set expression S introduced by a specification will have a type $\mathbb{P}T$, and will thus be assigned a subset of T^* in an interpretation. For example, $\{f(o)\}$, with type $\mathbb{P}nats$, will be assigned the set consisting of the value of $f(o)$ whether this value is in $nats^+$ or $nats^-$. If S is a non-type set expression whose type is the powerset of a given set G , and S^* is the set assigned to S in an interpretation, then the application of $+$ to S^* simply produces S^* itself; clearly, S^{*+} may or may not be a subset of G^{*+} . We also define $(\mathbb{P}\Pi)^+ = \mathbb{P}\Pi^+$ and $(\Pi_1 \times \dots \times \Pi_n)^+ = \Pi_1^+ \times \dots \times \Pi_n^+$ for any types or sets (in the models) $\Pi, \Pi_1, \dots, \Pi_n$, analogously to $*$. Similarly, for values $v_1, \dots, v_n$, $(v_1, \dots, v_n)^+ = (v_1^+, \dots, v_n^+)$. For any other interpretation value v we define $v^+ = v$.

In the following, we view any name n introduced by a declaration $n : S$ within a Z specification as a constant of the appropriate type, and thus assign a single meaning to it in any interpretation. Names introduced by declarations in sequents, on the other hand, are viewed as free variables, and for a sequent to be “valid” it must be “true” in every interpretation under all (well-typed) assignments to its free variables.

We are now able to define the meaning of our Z expressions in an interpretation I , under an assignment $[d]$ to any free-variable occurrences. In the following, let S, S_i denote set expressions, D_i denote a declaration, P, Q denote predicates, a_i denote a free variable or a constant, x, x_i denote free variables, c denote a constant, s, t, t_i denote terms, f denote a term of type $\mathbb{P}(T \times T')$ for some T, T' , and R denote a relation symbol. In the language of our models we use the same symbols $\mathbb{P}, \times$ and brackets $\{, \}, (,)$ as are used in Z , since no confusion should arise. However, because the relationship between $\in$ of Z and membership in the models is not straightforward, we use ϵ for membership in the model language. The symbol $=$ is our metalogical equality. (The total function App is defined as before.)

Declarations:

$$\begin{aligned} I(D_1; \dots; D_n)[d] &= I(D_1 \wedge \dots \wedge D_n)[d] \\ I(a_1, \dots, a_n : S)[d] &= I(a_1 \in S \wedge \dots \wedge a_n \in S)[d] \end{aligned}$$

Terms:

$$\begin{aligned} I(c)[d] &= c_I, \text{ an element of } T^*, \text{ where } T \text{ is the type of } c. \\ I(x_i)[d] &= d_i, \text{ the } i\text{th element of } [d] \\ I((t_1, \dots, t_n))[d] &= (I(t_1)[d], \dots, I(t_n)[d]) \\ I(t.i)[d] &= I(t_i)[d], \text{ where } t_i \text{ is the } i\text{th element of the tuple } t. \end{aligned}$$

$$\begin{aligned}
I(f \ t)[d] &= App(I(f)[d], I(t)[d]) \\
I(\{t_1, \dots, t_n\})[d] &= \{I(t_1)[d], \dots, I(t_n)[d]\} \\
I(\{x : S \mid P\})[d] &= \{y \mid y \in I(S)[d]^+, I(P)[d, y]\},
\end{aligned}$$

where y is a meta-language variable assigned to x .

$$I(\{D \mid P \bullet t\})[d] = \{I(t)[d, \mathbf{v}] \mid I(D)[d, \mathbf{v}], I(P)[d, \mathbf{v}]\},$$

where $\mathbf{v}$ is a sequence of (meta-language) variables $v_1, \dots, v_m$ assigned to the variables introduced by D .

$$\begin{aligned}
I(\mathbb{P} \ S)[d] &= \mathbb{P}(I(S)[d]) \\
I(S_1 \times \dots \times S_n)[d] &= I(S)[d] \times \dots \times I(S')[d]
\end{aligned}$$

Predicates:

$$\begin{aligned}
I(true)[d] &= True \\
I(false)[d] &= False \\
I(\neg P)[d] &= True \quad \text{iff} \quad I(P)[d] = False \\
I(P \vee Q)[d] &= True \quad \text{iff} \quad I(P)[d] = True \text{ or } I(Q)[d] = True \\
I(P \wedge Q)[d] &= True \quad \text{iff} \quad I(P)[d] = True \text{ and } I(Q)[d] = True \\
I(t \in S)[d] &= True \quad \text{iff} \quad I(t)[d]^+ \in I(S)[d]^+ \\
I(R \ t)[d] &= True \quad \text{iff} \quad I(t)[d]^+ \in I(R)[d] \\
I(s = t)[d] &= True \quad \text{iff} \quad I(s)[d]^+ = I(t)[d]^+ \\
I(\forall x : S \bullet P)[d] &= True \quad \text{iff} \quad I(P)[d, a] = True \text{ for all } a \in I(S)[d]^+
\end{aligned}$$

In the final equation, a is assigned to any occurrences of x in P . The meaning of more general universal quantifications (e.g. $\forall x, y : S; z : S' \bullet P$) is similar.

Example:

$$\begin{aligned}
I(\{x : \mathbb{N} \mid \neg x = 0 \bullet f(x)\}) &= \{f(y) \mid y \in \mathbb{N}^+, y \neq 0_I\}^+ \\
&= \{f(y) \mid y \in \mathcal{N}, y \neq \bigcirc\}
\end{aligned}$$

where, in the standard model, $\mathcal{N}$ is the set of natural numbers and $\bigcirc$ is zero.

$$I(f(0) = succ(f(0))) = (App(I(f), 0_I) = App(succ_I, App(I(f), 0_I))).$$

We may now define models and consequence in the usual way.

Definition: An interpretation I is said to be a *model* of a specification Σ if all the axioms in Σ are satisfied by I (i.e. have the value *True* in I).

In a model (if one exists) of a specification including

$$f : \mathbb{N} \rightarrow \mathbb{N} \text{ and } f(0) = succ(f(0)),$$

the value of $App(f_I, 0_I)$ will be non-numeric, but of type $\mathbb{A}$.

Definition: Let $X = (x_1, \dots, x_n)$ be the sequence of free variables occurring in the declarations D in the conjecture $D \mid, \vdash \Delta$. Then the conjecture is said to be a *consequence* of a specification Σ if every model of Σ satisfies $\wedge \Rightarrow \Delta^\vee$ under all assignments $[d_1, \dots, d_n]$ to X , where $\wedge$ is the conjunction of the members of $\wedge$, and $\Delta^\vee$ is the disjunction of the members of Δ .

For example, $(0, 0) \in f$ does not follow from the specification

$$[f : \mathbb{N} \rightarrow \mathbb{N} \mid f(0) = 0],$$

since $App(f_I, 0_I)$ must be equal to 0_I in any model of the specification, but $(0_I, 0_I) \in f_I$ need not hold.

7 Soundness of our Rules

In this section we demonstrate that our expansion and inference rules are consistent with the above semantics, via a soundness proof, and contrast this with the CADiZ rules. This should make sense of our semantics from the perspective of a typical Z/CADiZ user.

We say that an inference rule is *sound* if, whenever the hypotheses are a consequence of a specification, so is the conclusion. By an induction on proof structures we may show that soundness of our inference rules implies that our proof system is sound in the sense that, if a sequent X is proven from a specification Σ , then X is a consequence of Σ (cf. [3]). We thus concern ourselves merely with the soundness of our rules.

If we translate a conjectural sequent $D \mid , \vdash \Delta$, for non-empty $,$ and Δ , into the Z sentence $\forall D \bullet , \wedge \Rightarrow \Delta^\vee$, then the sequent is a consequence of a specification Σ iff the sentence is *True* in every model of Σ . (If $,$ and/or Δ is empty, we may replace them by, respectively, *true* and *false* in the corresponding sentence.) We may thus show that, for each inference rule, if the hypotheses translated into sentences are *True* in every model of Σ , then so is the conclusion. We signify that a sentence S is *True* in all models of a specification Σ by $\Sigma \models S$.

Theorem: Our inference rules are sound.

Proof:

Tautologies: The tautologies are clearly *True* in all models, and thus sound when viewed as axioms.

Normalisation: This rule is sound if $\forall x : S \bullet x \in S \wedge \gamma \Rightarrow \delta \models \forall x : S \bullet \gamma \Rightarrow \delta$, for arbitrary Z predicates γ and δ , and set or type expression S . With the assignment $[d]$ omitted for brevity, the meaning of the antecedent in an interpretation is

$$\forall y \bullet y \in I(S)^+ \Rightarrow (y \in I(S)^+ \wedge I(\gamma) \Rightarrow I(\delta));$$

but this simplifies to

$$\forall y \bullet y \in I(S)^+ \Rightarrow I(\gamma) \Rightarrow I(\delta),$$

which is the meaning of the consequent.

Function-application expansion: For a “function” $f \subseteq \mathbb{P}(T_1^* \times T_2^*)$, and argument $a \in T_1^*$,

$App(f, a) =$ the unique z such that $(a, z) \in f$ if such a z exists.

The first hypotheses of each inference rule for function application is concerned with showing that such a z exists for the function expression f and argument t . If it does exist, then we may assume its unique existence, and replace $P(f t)$ by its equivalent $\exists x : \tau(f t) \bullet F \wedge P(x/f t)$, which is what the second hypotheses of each inference rule does.

Equality reasoning: The inference rules of reflection, commutation, and Leibniz are clearly sound for the equality of non-type terms, since the meaning of such a term of type T is simply some element of T^* , and the meaning of equality over T in a model is the “identity” relation over T^* . Equality of types also satisfies these rules, since if $T_1^+ = T_2^+$, then $T_1^* = T_2^*$ by the disjointness of the model

types, and so the interpretation of equality is again the identity relation. The only difficult case is the equality of sets with types. If we have, for instance, $S = T$ in our specification, where S is a set expression and T is a type expression, then this implies that the meaning of S in a model is the same as T^+ , not T^* ; we do not have identity in the model. However, commutation clearly holds, and Leibniz holds because the meaning of $\in$, $=$, and so on, refer only to the sets S^+ for any set or type S , so the meaning of any predicate will be unchanged by the replacement of S by T (or vice-versa) if $I(S)^+$ and T^+ are the same.

Connectives: The soundness of these is straightforward.

$\forall$ quantifier: $\forall$ introduction is sound, since $\forall D \bullet \text{true} \Rightarrow P$ is equivalent to $\text{true} \Rightarrow \forall D \bullet P$. The soundness of $\forall$ substitution holds because

$$(Q \Rightarrow P'\{\mathbf{x} \rightarrow \mathbf{t}\}) \Rightarrow \delta \models (\forall D \bullet P) \Rightarrow \delta,$$

where $D, P, Q, \mathbf{x}$ and $\mathbf{t}$ are as in the inference rule, and δ is any predicate.

Membership Expansion: We justify only the expansion of powerset membership, leaving the other cases to the reader. Essentially, $S_1 \in \mathbb{P}S_2$ means $S_1^+ \in \mathbb{P}S_2^+$, while $\forall x : S_1 \bullet x \in S_2$ means $\forall x \bullet x \in S_1^+ \Rightarrow x \in S_2^+$. These are clearly equivalent according to the usual meanings of $\in$, $\mathbb{P}$, and so on.

Set-Equality Expansion: Essentially, $S_1 = S_2$ means $S_1^+ = S_2^+$, while the meaning of $\forall x : S_1 \bullet x \in S_2$ is as above, and similarly for $\forall x : S_2 \bullet x \in S_1$. Clearly, the meanings of the two sides of the expansion are equivalent. ■

The original CADiZ normalisation rule is unsound with respect to our semantics, since $t \in S \Rightarrow t \in \tau(t)$ may be false. The original CADiZ set-expansion rule is unsound with respect to our semantics, since S and S' , with type $\mathbb{P}T$, may have the same elements from T , and thus be the same according to the CADiZ rule, but may have different elements of type T ; for example, S might be $\{f o\}$ and S' might be $\{f(s o)\}$, where $I(f o) \neq I(f(s o))$ and neither $f o$ nor $f(s o)$ are in their type *nats*.

8 Properties of the Semantics

In our semantics, $t = t$ always holds, but a conjecture such as $\exists x : \mathbb{A} \bullet f(0) = x$, where f has the declaration $f : \mathbb{A} \rightarrow \mathbb{A}$, may not hold. Applying existential instantiation with x bound to $f(0)$ results in an identity, $f(0) = f(0)$, apparently proving the conjecture. However, there is a side condition to be proven, namely, $f(0) \in \mathbb{A}$, and this membership may not hold according to our semantics, since the fact that $f(0)$ has type $\mathbb{A}$ does not imply that $f(0) \in \mathbb{A}$.

As an example where “non-terminating” functions are involved, consider the following specification, based upon an example given by Smolka and Nutt [7]:

$$[f : \mathbb{N} \rightarrow \mathbb{N} \mid \forall n : \mathbb{N} \bullet f(n) = \text{succ}(f(n))]$$

If f had been declared to be a total function, then it would be possible to prove

$$\vdash f(0) \leq f(0) \wedge \neg f(0) \leq f(0),$$

a contradiction, but this is not surprising since a consequence of the axiom “defining” f is that f has no numeric value for any natural-number argument. This is a problem for Smolka and Nutt since they have only total functions; to

solve the problem they introduce an error element and make f total on the set $?\mathbb{N} = \mathbb{N} + \{\text{error}\}$. However, since f is defined as partial, no contradiction arises in our semantics. The reason the attempted proof of the contradiction fails is that again a membership constraint arises, $f(0) \in \mathbb{N}$.

These properties of our semantics seem reasonable, but there are other properties that may appear anomalous. Firstly, our “undefined” values may have properties, apparently contradicting their undefined status. For example, we might have $g(f(0)) = 0$ as an axiom, though $f(0)$ has no value in its range (i.e., is “undefined”). However, this holds for function applications that are undefined merely in the sense that they have not been explicitly assigned a value. For example, we might have $f(0) > 3$ as an axiom, though $f(0)$ has no explicit value according to the specification; that is, a model might assign any element of $\mathbb{N} \setminus \{0, 1, 2, 3\}$ to $f(0)$. We preempted this argument by pointing out that our “undefined” values should more accurately be referred to as “super” values.

A second, more disconcerting, aspect of our semantics is that our super values may have counterintuitive properties.

Example: Suppose we have a relation $_ > _ : \mathbb{P}(\mathbb{N} \times \mathbb{N})$, a function $f : \mathbb{N} \rightarrow \mathbb{N}$, and the usual axiom for $>$, $\forall x : \mathbb{N} \bullet \neg x > x$. No contradiction arises if we add to a specification with just these declarations and axiom, the properties $f(0) > 0$ and $f(0) < 0$, the reason being that, to contradict the given axiom, $f(0)$ must be assigned a value in $\mathbb{N}$ and there are models in which it is not assigned such a value.

Our first justification for this apparently anomalous result is that $f(0)$ has the unexpected properties only because it has been assigned them by the specification. Of course, even if the axioms $f(0) > 0$ and $f(0) < 0$ had not been added to the specification, there would still be models of the original specification in which they hold, but, in that case, we would not be able to prove that they hold as a consequence of the specification.

Secondly, we argue that the counter-intuitive nature of this result partly arises from the choice of the symbol $>$ for our relation. Consider a specification in which we have a relation $R : \mathbb{P}(\mathbb{A} \times \mathbb{A})$, and the axiom $\forall x : \mathbb{N} \bullet \neg R(x, x)$; clearly no contradiction arises (formally or intuitively) if we add $R(1 \div 2, 1 \div 2)$, where ‘ $\div$ ’ denotes division: a model for this specification is one in which R is assigned the relation $\forall x, y : \mathbb{A}^* \bullet x * y < x$. If R had been called $>$, however, $1 \div 2 > 1 \div 2$ would have been disconcerting. Of course, Z users will typically wish to use suggestive names and symbols (such as $>$) for their relation and function symbols; the point is that, since a relation is always declared over a set (of, for example, pairs), the specifier cannot expect all defined properties of the relation to hold over objects not in that set, and these objects include the super values in our semantics.

All these properties of our semantics have a counterpart in the alternative approach to partiality in which we modify, for instance, the concept of a free type within Z. For example, if we define $nats$ as a subset of the given type $nats^*$, and include the axiom $\forall x : nats \bullet \neg x > x$, then we may or may not have $fo > fo$ in any model in which fo is assigned some value from $nats^* \setminus nats$. Similarly, the reinstatement, in the type-change approach, of the property $\forall x : T \bullet \exists y : T' \bullet f x = y \Rightarrow (x, y) \in f$, where T, T' are types, and f has type $\mathbb{P}(T \times T')$, as suggested in Section 4.1, is also possible in the semantic approach.

The relationship between the two approaches to partiality may be formalised as follows. Let Σ be a specification with exactly the given types $T_1, \dots, T_n$, and in which there is no use of the names $T_i^*, T_i^\#$ or T_i^+ for any i . We derive from Σ the specification $\Sigma^\#$ by replacing the introduction of each given type T_i with the introduction of the given type $T_i^\#$, and by adding the declaration $T_i : \mathbb{P} T_i^\#$ for each T_i . We also modify our semantics I to J by deleting all occurrences of ‘+’ — our subset of Z thus becomes virtually its own semantics. If we represent logical consequence in our I semantics with $\models_I$ and logical consequence in the J semantics with $\models_J$, then for any Z predicate α in the language of Σ , we have

$$\Sigma \models_I \alpha \text{ iff } \Sigma \models_J \alpha.$$

The reason for this is that, since neither Σ nor α makes reference to either the $T_i^*, T_i^\#$ or T_i^+ , we may correlate each T_i^* with the J interpretation of each $T_i^\#$, and correlate, for each set expression S in Σ or α , the I -set S^{*+} with the J interpretation of S . We then have a correspondence between the two semantics for α and the expressions in Σ .

This does not mean, however, that the approaches are, for all practical purposes, identical. One major difference is the relative difficulty in defining certain concepts, such as $\cup$, even if we suppose they are defined via expansion laws. In the first approach, we may define the union of any two arbitrary set expressions S and S' by

$$\{x : T \mid x \in S \vee x \in S'\},$$

where $\mathbb{P}T$ is the type of S and S' . This definition will not always produce the full union in the second approach, but only the union of the “defined” subsets. On the other hand, for practical applications we may wish to construct only the union of these defined subsets. For example, in the process of refinement, where we wish to construct a function satisfying our specification, it is likely that the desired domain for our function will be the (maximal) set over which the function’s specification is total [10]. There seems to be no practical reason for constructing the full union of sets with undefined elements; the construction of such sets in this paper was purely for theoretical considerations.

One, as yet unresolved, difficulty with our semantic approach to partial-function application is that we cannot always simplify expressions involving undefined terms as we would like. For example, we cannot simplify the expression $(s \circ \div o) + o$ via the equation $\forall x : nats \bullet x + o = x$, because we cannot show that $s \circ \div o$ is in the type $nats$; the principal benefit of Arthan’s approach [2] is that such expressions may be simplified, since $s \circ \div o$ would be assumed to be in $nats$. One possible resolution to this problem is to allow untyped quantifiers³. For example, we might allow equations such as $\forall x \bullet x + o = x$. Any applied instance of predicates with untyped quantifiers must have appropriate types. For example, any applied instance, $t + 0 = t$, of the equation $\forall x \bullet x + o = x$, must be such that $t + 0$ and t are both well-typed *Terms* and have the same type. This constraint on the application of predicates with untyped quantifiers should ensure that untyped quantifiers do not introduce the sort of problems (e.g. paradoxes) typing is designed to eliminate [8]. Consequently, we are not advocating that all expressions be untyped.

³Rob Arthan, private communication

9 Conclusions

We have argued that it is advantageous to be able to include in our specifications functions applications that do not always return a value of the function's range, but have shown that this can make typical Z specifications inconsistent according to the usual (intuitive, if not yet formalised) semantics of Z. We have also presented two possible resolutions to this inconsistency problem, one involving a modification to our specifications, and the other a modification to our proof systems.

We have shown that the two approaches are closely related, and thus have some benefits in common. Neither imposes constraints on the meaning of a function itself; in particular, partial functions are not “totalised”. Also, we do not have a “bottom” object, and do not have the problem of deciding whether or not our functions should be strict: the value of a function application to a super value is simply an element of the set assigned to the type of the function application. Perhaps most importantly, partial functions are not total from the perspective of function application.

The major disadvantage of the specification-change approach is that it is an imposition on the specifier, even if automated tools make the necessary changes. The inference-change approach avoids this problem, for the most part, but deviates from the classical view of and approach to reasoning about Z [8, 2, 9]. If either of these approaches are to be used in practice, preference will probably depend upon the perceived utility and pervasiveness (in real specifications) of non-total function application.

Irrespective of the acceptability of non-total function application, it is hoped that this paper has clarified the meaning of application in general, including the application of relations that are neither total nor partial functions. With respect to partial functions, Arthan [2] and Valentine [10] have argued that we may assign some member of their range types to any application to objects outside their domains. In this paper, we have shown further that this possibility is implicitly built in to current reasoning systems for Z such as CADiZ: while it is not immediately apparent that $f o$ is an element of the type *nats* if f is a partial function from *nats* to *nats*, we have shown that it is possible to prove in CADiZ that $\{x : \text{nats} \mid \text{true} \bullet f x\}$ is a subset of *nats*. Thus, any other approach to partial-function application, such as the introduction of a bottom object $\perp$ or the one we have proposed, will entail modification of the inference rules of reasoners such as CADiZ.

There are several possible directions for future research. We have covered only a fairly small subset of Z, omitting, most prominently, schemas and generics; inclusion of these is paramount for the utility of our work to Z users. There is also more work to be done on the effects of non-total function application on the issues of refinement and inductive proof; recently, we have been developing an approach to proving inductive properties of partial functions that utilises the distinction our semantics makes between defined and undefined function applications. Completeness results for our proof rules with respect to the described semantics might also be developed: we should be able to show that our rules support the proof of all possible properties of partial functions, at least for subsets of Z. In addition, we need to consider further the problem of simplifying (rewriting) expressions involving undefined terms.

Acknowledgements

I would like to thank Ian Toyn for his detailed and helpful comments on successive versions of this paper, and to thank Ian, Rob Arthan, Robert Dormer, Alan Frisch, and Sam Valentine for our discussions of the ideas presented.

References

- [1] J-R. Abrial. *The B-Book: Assigning Meaning to Programs*. Cambridge University Press, 1996.
- [2] R.D. Arthan. Undefinedness in Z: issues for specification and proof. In *CADE-13 Workshop on Mechanisation of Partial Functions*. Springer, Rutgers University, New Brunswick, New Jersey, USA, 1996.
- [3] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [4] H.B. Enderton. *Elements of Set Theory*. Academic Press, 1977.
- [5] D.J. King and R.D. Arthan. Development of practical verification tools. *Ingenuity - the ICL Systems Journal*, 11, May 1996.
- [6] Z. Manna. *Mathematical Theory of Computation*. Prentice-Hall International Editions, 1974.
- [7] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-sorted equational computation. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*. Academic Press, 1989.
- [8] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice Hall, 1992.
- [9] I. Toyn. Proving conjectures using CADiZ. Available from the Department of Computer Science, University of York at the web address <http://www.cs.york.ac.uk/~ian/cadiz/tutorials.html>, 1997.
- [10] S. Valentine. Inconsistency and undefinedness in Z – a practical guide. To appear in *Proceedings 11th International Conference of Z User's, ZUM'98*, 1998.
- [11] C. Walther. Mathematic induction. In D.M. Gabbay, C.J. Hooger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 2: Deduction Methodologies*. Clarendon Press, Oxford, 1994.
- [12] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.