

# Data Structures for Z Testing Tools

Mark Utting

Waikato University, Hamilton, New Zealand

Email: `marku@cs.waikato.ac.nz`

**Abstract:** This paper describes some of the difficulties and challenges that arise during the design of tools for validating Z specifications by testing and animation. We address three issues: handling undefined terms, simplification versus enumeration, and the representation of sets, and show how each issue has been handled in the Jaza tool. Finally, we report on a brief experimental comparison of existing Z animators and conclude that while the state of the art is improving, more work is needed to ensure that the tools are robust and respect the Z semantics.

## 1 Introduction

Tools for analyzing Z specifications can be classified into two main groups, based on whether they attempt to show universal or existential properties:

**Proof-like Tools:** This includes type checkers and theorem provers. The goal of using these tools is to show that the specification has some desirable property *for all possible inputs and outputs*. The main emphasis of these tools is symbolic formula manipulation.

**Testing Tools:** This includes test-case checkers and animators. The goal of these tools is to check some property of a given set of test data, or to generate some data values that satisfy the specification (or contradict it, in the case of model checkers). Although these tools must perform some symbolic formula manipulation, their main emphasis is on manipulating or evaluating specific ground terms (i.e., terms that contain no variables).

The two classes of tools are complementary. Testing tools are best used early in the system life cycle, to validate specifications against example input-output test sets, or to execute specifications to detect errors (though not all specifications are executable [HJ89]).

Implementation techniques for the proof-like tools are similar to other theorem-proving applications, and there is much literature about theorem proving, type-checking etc. The fundamental data structure of these tools is an abstract syntax tree of the specification, and this is transformed in various ways (e.g., rewriting) to prove the desired results.

Testing tools require different implementation techniques. This paper describes some of the issues that are important for Z testing tools: handling undefined terms (Section 2), managing the balance between general symbolic formula manipulation and enumeration and evaluation of ground data values (Section 3) and a variety of data structures for representing sets (Section 4)

The techniques described here have been implemented in the *Jaza* tool. *Jaza* is a new, open-source, animator for the Z specification language that is written in Haskell. It currently provides only a text-based interface (a GUI is planned), but handles most of Spivey Z [Spi92] except for generics and bags. The decision to develop *Jaza* came from our difficulties with using other animators, such as poor support for quantifiers or various less-often-used Z constructs ( $\mu, \lambda, \theta$  terms), unpredictable performance characteristics, and inability to handle non-deterministic schemas.

*Jaza* is primarily aimed at supporting two validation techniques: testing and execution (including execution of non-deterministic operations). It is designed to complement proof-like tools, such as Z/EVES [Saa97], by providing more efficient and convenient evaluation of schemas on ground data values, and the ability to search for example solutions of a schema or predicate.

*Jaza* can also be used as a *desk-calculator* for Z expressions and predicates. This is useful for students who are learning the Z toolkit and wish to experiment with its operators and data types (sets, relations, functions, sequences etc.).

## 2 Undefinedness

Z uses a 'loose' approach to handling undefined terms [Spi92, Val98]. There is no specific *undefined* value. Instead, the result of applying a partial function to a value outside its domain is simply any arbitrary value of the correct type. The logic of Z is two-valued, so  $3/0 = 1 \vee 3/0 \neq 1$  is always true, as is  $1/0 = 1/0$ . However, little else can be deduced about an undefined term. For example, it is not possible to prove whether the set  $\{x : 0..3 \bullet 6/x\}$  has three or four elements. and requiring  $\#s = 3$  does not guarantee that  $s$  is finite, since  $\#$  may return 3 when applied to an infinite set!

This approach is simple, though it has some strange consequences for singleton types [Jon95], and easily leads to subtle errors in specifications. Z/EVES provides a *domain check* facility that generates a proof obligation to ensure that a schema does not rely on the result of any partial function outside its domain, and when they analyzed 400 published Z specifications, every one failed this test [SDG99]!

Provers can support this loose approach to undefinedness by simply not providing any inference rules for reasoning about undefined terms, other than basic logical laws such as  $x = x$ . However, for testing tools, we want a more precise solution, so that we can keep track of exactly which terms are defined and which are not. For example, this is important if we want to be able to evaluate a ground term in a single bottom-up pass.

One possible approach would be to assign a specific return value to each partial function. For example, decide that division by zero will always return 42. However, this would mean that the value returned from evaluating  $3/0$  would be more specific than in Z, which does not commit to a specific value. This could lead to returning true for a predicate that other Z tools evaluate to false (or vice versa), which is undesirable.

A better approach is to add an explicit *undefined* value to each type, so that we can carefully define how functions deal with undefined inputs. Most functions

need to be strict on undefined inputs, but logical operators have more freedom, since  $true \vee X$  is  $true$  regardless of  $X$ .

Jaza implements this approach by defining an *ErrorOr*  $t$  type that wraps exception handling around any type of term  $t$ . The *ErrorOr* type contains an *Ok* alternative, which contains a defined value of type  $t$ , plus an *Undef* alternative, which means that the result is undefined. It also contains other alternatives which are used for various classes of errors, such as syntax and type errors, ‘search space too large’, ‘undecidable equality’, etc. This *ErrorOr* type is wrapped around the evaluation results of all Z schemas, expressions *and predicates*. So Jaza effectively uses a three-valued logic. This is slightly more strict than Z requires, but has the benefit that it is possible to report if a predicate like  $P \vee \neg P$  depends upon undefined terms.

This makes it easy to define non-strict functions as well as strict ones. For example, logical conjunction  $\wedge$  is defined as:

```
zand :: ErrorOr Bool -> ErrorOr Bool -> ErrorOr Bool
zand (Ok False) _           = Ok False
zand _ (Ok False)          = Ok False
zand (Ok True) (Ok True)   = Ok True
zand err1 err2             = mergeErrors err1 err2
```

Note that the lazy evaluation of Haskell means that  $A \wedge B$  will not evaluate  $B$  at all if  $A$  happens to be false. If  $A$  is true, then it is necessary to evaluate  $B$ . If  $A$  is undefined, there is the danger that the evaluator will spend forever trying to evaluate it, even though  $B$  happens to be false. The Jaza execution engine is designed to avoid this problem by always returning some result reasonably quickly – it performs a limited search for solutions, and returns an error if the search space is too big. The philosophy is that it should always be safe (and reasonably quick) to evaluate any term, whether defined or not.

The *ErrorOr* type is defined as a monad [Wad95], so the Haskell ‘do’ notation (which looks like imperative programming) hides all the exception handling and allows the programmer to specify just the non-error case. This has the effect of always returning the first exception that is raised. In a few places in Jaza, more sophisticated exception handling is useful, and this is easily done by manipulating the *ErrorOr* values directly and checking for particular kinds of errors. For example, during the evaluation of  $(\exists x : 0..100 \bullet P)$  we try evaluating  $P$  with  $x$  set to successive values,  $0..100$ . We ignore some undefined results from  $P$  in the hope that a later value for  $x$  will make  $P$  true. The whole predicate returns *OkTrue* if we find any choice of  $x$  that makes  $P$  true, *OkFalse* is returned when *every* choice of  $x$  makes  $P$  false, and some kind of error is returned otherwise.

### 3 An Animation Architecture

It is tempting to take a theorem-proving approach to evaluating schema on test cases — repeatedly apply simplification rules until no further simplification is possible. However, this *symbolic rewriting approach* is inefficient compared to how compiled programming languages evaluate expressions (a single bottom-up pass).

Jaza attempts to get the best of both approaches by applying a fixed sequence of steps to each schema or expression evaluated. The philosophy is to perform most of the ‘expensive’ symbolic analysis (simplifying terms, determining computation order, reasoning about the free variables of each expression etc.) at the beginning, so that later phases can operate in a more predetermined style with a fixed evaluation order like a programming language.

Furthermore, a key technique that testing tools need to use is *enumeration* — for example, turning a symbolic set like  $\{x : 0 \dots y \mid y \text{ div } x = 0\}$  into a set of explicit integers. Proof-like tools are generally reluctant to do this because it can lead to large case analyses, but it is essential in testing tools, where the goal is often to find an instance of a schema or set. The following architecture does an enumeration phase after all the symbolic formula manipulation phases.

The phases are:

1. **Parse:** The LaTeX input is parsed into an abstract syntax tree (AST). The Z grammar is ambiguous, so some ambiguities are left in the AST.
2. **Unfold:** This phase uses context to resolve all ambiguities, then expands all schema calculus operators, and unfolds all references to other schemas and global definitions. Several Z constructs (e.g.,  $\lambda, \theta$  and schemas-as-types) are translated into set comprehensions. The resulting term is saved in the specification database until the user calls it.
3. **Substitute Values:** When the user invokes an operation schema, and supplies values for some of the inputs and/or outputs, the term corresponding to that schema is retrieved from the database and the given values are substituted into it.
4. **Simplify:** Simplification rules, such as one-point rules, are applied in a bottom-up pass through the term, so that the schema is specialized to the given input/output values. (If all inputs are supplied, this typically fully evaluates the precondition and narrows a disjunctive schema down to one case). A cute feature is that the ‘safe to evaluate any term’ philosophy allows this pass to call the *eval* function on each atomic predicate. If it does not return an error, then the predicate does not depend upon other variables, and can be replaced by the result of *eval* (*True* or *False*).
5. **Optimize:** This reorders declarations and predicates to minimize searching. This phase performs extensive analysis of the free variables of each predicate, and looks for useful consequences of predicates that can be used as constraints to narrow the search space. For example, from the predicate  $a \wedge b = c$  we can deduce three facts:  $c = a \text{ \texttt{cat} } b$  (which determines a unique value for  $c$  once  $a$  and  $b$  are known);  $a \text{ \texttt{prefix} } c$  (which narrows the search for  $a$  to  $\#c + 1$  possibilities once  $c$  is known); and  $b \text{ \texttt{suffix} } c$  (which similarly narrows the search for  $b$  once  $c$  is known).

The result of this phase is a sequence of intermixed declarations and predicates, where each declaration has the form  $x : \bigcap\{C_1, C_2 \dots C_n, T\}$  where  $C_i$  are the constraints on  $x$  that have been deduced from predicates and  $T$  is the original type of  $x$ . This narrows the search space for  $x$ , often to a single

value if there are equalities involving  $x$ . Each predicate appears as early as possible in the sequence, immediately after all its free variables have been declared. The goal is to push each filter predicate as early as possible, so that unsuccessful search paths are pruned as early as possible.

6. **Search:** This phase uses the common backtracking (or 'generate-and-filter') approach to search for a solution of the optimized declaration sequence  $D$ . Each  $x : T$  in  $D$  is executed by assigning each member of  $T$  in turn to  $x$ , with the rest of  $D$  being evaluated to determine whether the chosen values are acceptable. When each predicate is evaluated, all its free variables have ground values, so evaluation can be done efficiently in a single bottom-up pass, just like in programming languages.

This process returns a (lazy) list of all possible bindings that satisfy the schema. If  $D$  is not in a context like  $(\forall D \bullet P)$  or  $\#\{D \bullet E\}$ , which requires all solutions to be known, then it is often only necessary to fully evaluate the first one or two solutions. At the top level, Jaza provides user-level commands for stepping through the solutions of a non-deterministic schema.

If the schema is obviously deterministic and the optimization phase detected this, then no searching will be needed. To ensure reasonable response times for non-deterministic schemas, we limit the search space. The depth of the search is bounded by the length of the declaration sequence and the width is bounded by a user-settable parameter. A useful trick allows us to detect large search spaces in constant time without actually performing the whole search: if we are searching  $[a : T_a, b : T_b, c : T_c \mid P]$  with a maximum width of 100,000, and each of the types contain 100 values, we choose the first value of  $a$  and note that there are 100 alternatives to try, then choose the first value of  $b$  and note that there are now  $100 * 100$  alternatives, then proceed to  $c$  and immediately raise an exception, because the search space now contains  $100 * 100 * 100$  alternatives.

This description of the architecture is slightly idealized. In practice, when the 'search' phase fails to evaluate large sets, it sometimes returns them in symbolic form, in case the context can narrow the search. For example, when evaluating  $\{x : \mathbb{N} \bullet (x, x + 10)\} 3$ , the set comprehension is returned unevaluated, then the function application unifies 3 with  $x$ , giving  $\mu x : \mathbb{N} \mid x = 3 \bullet x + 10$ , which evaluates to 13.

Another advantage of this architecture is that it would be quite easy to add a 'compilation' stage in just before the evaluation phase, generating Haskell or C code to perform the final 'evaluate' phase. This would probably provide an extra 10-fold or more speedup, but we have not yet found this necessary.

## 4 Multiple Set Representations

Breuer and Bowen [BB94] classify Z animators by how they represent sets:

- (a) sets must be finite and are modelled by finite arrays/lists;
- (b) sets may be countably infinite, are modelled by an enumeration algorithm;

(c) sets are cardinally unbounded and modelled by their characteristic function.

For example: ZANS [Jia] and ZAL [MSHB98] use approach (a) only; Breuer and Bowen’s animator uses (b) only; Prolog-based Z animators typically use approach (c) [DKC89, DN91] and Z<sub>-</sub> supports both ‘passive’ sets (a) and ‘active’ sets (c) [Val91].

However, each of these approaches is good for some kinds of sets, but disastrous for others. Approach (a) does not allow infinite sets, which are widely used in Z, but has the advantage that efficient algorithms and data structures can be used for the finite sets. Approach (b) amounts to programming with (potentially infinite) lazy lists (‘streams’), so binary operators must address fairness issues and typically suffer from termination problems. Approach (c) is good for membership queries, but makes it difficult to enumerate the members of a set or calculate its cardinality. Note that proof-like tools tend to use only a symbolic representation of sets, which is closest to approach (c). This makes manipulation of finite, enumerated sets awkward and inefficient.

The best solution is to use *all* the above representations for sets, plus other representations that are customized to specific kinds of sets. Jaza supports at least 12 different representations of sets! Each set is kept in its optimal representation, and translated into another representation only when an operator requests it. A typical implementation of an operator handles several representations using special algorithms, then translates any remaining representations into a standard form so that it can apply a standard algorithm. (An error will be raised if the translation is impractical). Let us look at some of Jaza’s most important set representations.

**ZFSet [ZValue]:** A finite set of values that are all in ‘canonical’ form<sup>1</sup> is represented by a sorted list of values (without duplicates). This makes the common set operations ( $\cap, \cup, \setminus, \#, \in$ ) linear in the size of the sets.

**ZSetDisplay [ZExpr]:** Finite sets of arbitrary terms are also represented by lists, but slower  $O(N^2)$  algorithms must be used, because the list may contain two elements that are syntactically distinct, yet semantically equal.

**ZSetComp [ZGenFilt] ZExpr:** Set comprehensions contain a sequence of intermixed declarations and predicates that are optimized as described in Section 3, plus an expression for the outputs of the set. This representation can sometimes be executed to produce a finite set, but if the search space is too large for that, it is left in this ZSetComp form. Membership tests and function applications ( $\lambda$  expressions are always translated into ZSetComp form) can often be processed without enumerating the set. Instead, the alleged member is unified with the output expression of the set and this often narrows the search space. For example:  $E = \{x : \mathbb{N} \bullet x * 2\}$  is infinite so cannot be enumerated, but if it appears in the predicate  $4 \in E$ , this can be

---

<sup>1</sup> A value is in canonical form iff it is entirely constructed from basic values (integers and members of given sets) and the tuple, binding, free-type and ZFSet constructors. Canonical values have useful properties such as: no further evaluation is possible, no undefinedness can arise, and semantic equality is the same as syntactic equality.

transformed to  $(\exists x : \mathbb{N} \bullet 4 = x * 2)$  which can be evaluated to true without infinite search.

**ZIntSet (Maybe Integer) (Maybe Integer):** This represents a contiguous range of integers, with optional upper and lower bounds. For example, if both bounds are missing, it means  $\mathbb{Z}$ ; if the lower bound is 0 it means  $\mathbb{N}$  and if both bounds are given, it means  $a..b$ . Membership, cardinality and intersection tests can be done in constant time with this representation, and it is very useful for narrowing search spaces by intersecting integer inequalities.

**ZFuncSet:** This data structure represents function/relation spaces. It stores the domain and range sets, plus seven boolean flags that determine what kind of relational space it is (total, onto, functional, etc.). Typically, this is used as a type, and the most common operation is testing a given set of pairs for membership in the type. However, calculating the intersection of two function spaces is easy with this representation. The predicate  $\text{dom } f = s$  can be represented as  $f \in \text{ZFuncSet}\{\text{domset} = s\}$  and this is an elegant way of narrowing the search space for a function value. Similar data structures are used for power sets and cross products.

**ZFreeType Name [ZBranch]:** Free types are represented by a data structure that specifies the shape of all the possible recursive tree values that are members of the free type. It is easy to answer membership tests with this data structure, but it is also possible to enumerate all members of a non-recursive free type if its domain sets are finite.

Relating these to the Breuer/Bowen classification, ZFSet and ZSetDisplay are approach (a), while ZSetComp and ZFuncSet are approach (c). Jaza does represent any sets directly using approach (b), but provides coercion functions that turn the other set representations into a stream of values.

The danger of having multiple set representations is that it can lead to a combinatorial explosion in the algorithms that are needed for each binary operator ( $12*12$  possible input formats). However, our experience so far in Jaza shows that this problem can be overcome by defining a comprehensive set of coercion functions (which raise exceptions when necessary), and by providing query functions that return hints about the size and nature of sets.

## 5 Experimental Evaluation

This section describes a small evaluation of the coverage and correctness of the most commonly available Z animation tools. The goal is to measure the state of the art, by seeing how well the tools support the less commonly used Z features (presumably all tools handle  $\cup$  correctly, but how many implement  $\theta$  correctly?) and how closely they follow the Z semantics.

Our method is to take a wide selection of the Z constructs and toolkit operators and see how well each animator handles each construct. We ensure that all variables range over small integer ranges, so that the size of the search space will not affect the results. For each construct, we typically try a simple case,

plus several more subtle cases, in order to get some idea of how closely the tool respects the Z semantics. The second half of the table involves some infinite sets, combined with finite sets in ways that should be feasible to evaluate. The set of test expressions is shown in Appendix A, with the correct answers.

Figure 1 shows the results. The animators that were evaluated are all available from the Oxford Z page: <http://archive.comlab.ox.ac.uk/z.html>:

- ZANS (associated with ZTC). The latest version is version 0.3.1a, built on Apr 12 1996.
- Possum 1.0, from the Software Verification Research Centre in Australia.
- ZAP version 2.0, which is part of the ZETA environment.
- Jaza version 0.82.
- Z/EVES version 2.1. This is a prover rather than an animator, though one of its design goals is to be capable of evaluating most ground terms. We include it to show how well proof-like techniques work for animation purposes, using just the fully automatic proof command ‘**prove by reduce**’, with no interactive proof.

Z/EVES has the only perfect correctness record (it did not produce any wrong results or ever fail with an error message) but its automatic proof was surprisingly ineffective at evaluating these kinds of terms (presumably the desired results could be proven interactively). This shows that enumeration is an important technique for animation tools. Z/EVES avoids enumeration, preferring to stick with symbolic formulae in order to avoid excessive case splits within proofs. In contrast, the other tools use enumeration quite frequently, so are more effective at evaluating these terms.

Several tools had problems with declarations that consist of just a schema name, as in  $\{S \mid y < 3\}$ . Possum gave a syntax error, ZANS dumped core, and Z/EVES seemed unable to unfold  $S$ . For these tools, I had to manually unfold  $S$  in some of the later tests.

ZAP ignores type constraints in the signature part of schemas, so this required rewriting  $S$  to move the  $\in 1..10$  constraints into the predicate part. The  $\mu$ ,  $\lambda$  and  $\exists_1$  tests had to be rewritten in a similar way. This is inconvenient if you are using existing Z specifications, but if you were writing specifications with ZAP in mind, it would be relatively easy to ensure that all types in declarations are base types.

Overall, the results show that these animators are all fairly comprehensive in their coverage of Z constructs, but they all (except Z/EVES) have areas where they sometimes return wrong results (Jaza did not actually return a wrong result, but the  $\{x : S \mid \dots\}$  case was expanded incorrectly, and this could cause a wrong result to be returned, even though in this case it produced an error message instead). It was good to see that Possum, ZAP and Jaza all handle undefined expressions correctly (Z/EVES does too, but it takes the ‘cannot prove anything about undefined expressions’ approach rather than representing undefinedness explicitly).

Z Construct	ZANS	Possum	ZAP	Jaza	Z/EVES
Find instance of S	—	✓	—	✓	—
$(\exists S \bullet \dots)$	crash	—	✓	✓	uneval
$\{S \mid \dots\}$	crash	—	✓	✓	uneval
$\{x : S \mid \dots\}$	fail	✓	crash	fail	uneval
$\mu$	✓,X	✓,X	✓	✓	uneval
$\lambda$	✓,X	✓	✓	✓,fail	uneval
$\theta$	crash,X	✓,X	✓,fail	✓	partial
$\mathbb{P}_1$	✓	✓	fail	✓	uneval
$\exists_1$	✓,X	✓	✓	✓	✓,partial
<i>partition</i>	✓	✓,X	✓	fail	partial
<i>disjoint</i>	✓	✓	✓	fail	uneval
$\uparrow$	✓	✓	✓	✓	✓
<i>squash</i>	X	✓,X	✓,X	✓	uneval
$\cap /$	✓	✓	✓	✓	✓
$\mathbb{N} \cap \text{finiteset}$	✓	✓,fail	✓	✓	✓
$\text{finiteset} \in \mathbb{P}\mathbb{N}$	✓	✓	✓	✓	✓
$\dots \in (1..2) \mapsto \mathbb{N}$	fail	✓	✓	✓	uneval
$n \in (\mathbb{N} \setminus \text{finiteset})$	fail	✓	✓	fail	✓
$n \in (\text{finiteset} \setminus \mathbb{N})$	✓	✓	✓	fail	✓
✓-only score	8	12	14	13	6

Table 1: Comparison of the Coverage and Correctness of Z Animators. Key: ✓=correct result; X=incorrect result; fail=failed with an error message; crash=crashed the animator; —=construct not supported; partial=result was not fully evaluated; uneval=result was not evaluated at all; S is the schema  $[x, y : 1..10 \mid x < y]$ .

ZANS has not been updated since 1996 and has quite a low score, but it is good to see that the more recently developed animators (Possum, ZAP and Jaza) are getting higher scores. These numbers agree with our informal observation that the state of the art in Z animation is improving, but the number of Xentries in the table shows that more work is needed before the tools are robust and reliable.

Other experimental evaluations would be useful additions to the above comparison. For example, it would be useful to explore many more kinds of infinite set expressions to determine what kinds of sets each tool can handle. It would also be interesting to compare the evaluation speed of the tools, and how well they scale up to handle large schemas, or large data sets. Another good way to evaluate Z animators would be to take a large set of existing Z specifications and apply them to all the animators, to see what percentage of specifications

each animator handles. Some schemas would not be executable of course, but it would be interesting to see how many of those are usable as test oracles (that is, can be evaluated when all input and output values are supplied). However, this would be a large project, and our experience suggests that much tinkering with the specifications would be necessary to realistically evaluate each animator.

## 6 Conclusions

One outcome of this work is the conclusions that testing tools need to support a wider variety of data structures for representing sets than proof-like tools. For proof-like tools, a single abstract syntax tree representation is sufficient, but testing tools place more emphasis on enumeration of sets and typically handle larger ground sets than provers, so it is preferable to provide data structures tailored to those various kinds of sets.

Our experience with developing Jaza has demonstrated that the multiple set-representation approach described in this paper is a viable technique for Z testing tools, and that it can allow coverage of a wide class of Z constructs.

Most of the Jaza problems found during the comparison were simply errors or small omissions and there seems to be no reason why Jaza could not obtain a perfect score in the above evaluation. Probably the same is true for Possum and ZAP. ZANS seems to be more aimed at finite sets, and may not be able to pass our second group of tests without major changes. Z/EVES is primarily a prover rather than an animator, and it is interesting to see that its automatic proof commands fail to evaluate most of our tests. It would need to be extended with enumeration rules and heuristics to be more effective in this application domain.

Our comparison of Z animators suggests that the state of the art is improving, but that more work is needed before the tools are robust and reliably correct.

## Acknowledgements

Thanks to Greg Reeve and Steve Reeves for many discussions about Z and animation, and to Greg for implementing the Jaza parser and pretty-printer. This research is partially funded by the New Zealand Foundation for Research, Science and Technology under the ISuRF (Improving Software using Requirements Formalization) project.

## References

- [BB94] Peter T. Breuer and Jonathan P. Bowen. Towards correct executable semantics for Z. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 185–209. Springer-Verlag, 1994.
- [DKC89] A. J. J. Dick, P. J. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 71–85, Berlin, Germany, 1989. Springer-Verlag.

- [DN91] Veronika Doma and Robin Nicholl. EZ: A system for automatic prototyping of Z specifications. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 189–203. Springer-Verlag, 1991.
- [HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.
- [Jia] Xiaoping Jia. An approach to animating Z specifications. Available from [ise.cs.depaul.edu](http://ise.cs.depaul.edu) with the ZTC and ZANS tools.
- [Jon95] C. B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.
- [MSHB98] Ian Morrey, Jawed Siddiqi, Richard Hibberd, and Graham Buckberry. A toolset to support the construction and animation of formal specifications. *The Journal of Systems and Software*, 41(3):147–160, 1998.
- [Saa97] M. Saaltink. The Z/EVES system. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–88, Reading, UK, April 1997. Springer-Verlag, Berlin.
- [SDG99] Bill Stoddart, Steve Dunne, and Andy Galloway. Undefined expressions and logic in Z and B. *Formal Methods in System Design*, 15(3):201–215, November 1999.
- [Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall International (UK) Ltd, second edition, 1992.
- [Val91] Samuel H. Valentine. Z--, an executable subset of Z. In J.E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 157–187, Berlin, Germany, 1991. Springer-Verlag. Proceedings of the Sixth Annual Z User Meeting, York, 16-17 December 1991.
- [Val98] S. H. Valentine. Inconsistency and undefinedness in Z – a practical guide. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 233–249. Springer-Verlag, 1998.
- [Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Baastad, Sweden*, number 925 in *LNCS*, pages 24–52. Springer-Verlag, 1995.

## A Test Inputs for Animator Comparison

<i>Test</i>	<i>Inputs</i>	<i>CorrectResults</i>
<i>Find instance</i>	<i>Jaza</i> : <i>do S</i> <i>Poosum</i> : <i>S</i> <i>Z/EVES</i> : <i>z</i> ∈ <i>S</i>	$\langle x == 1, y == 2 \rangle$
$\exists S$	$(\exists S \bullet y < 3)$	<i>true</i>
$\{S \mid \dots\}$	$\{S \mid y < 3\}$	$\{\langle x == 1, y == 2 \rangle\}$
$\{x : S \mid \dots\}$	$\{x : S \mid x.y < 2\}$	$\{\langle x == 1, y == 2 \rangle\}$
$\mu$	$(\mu x : 1..100 \mid x * x \in 81..90)$ $(\mu x : 1..100 \mid x * x \in 82..90)$ $(\mu x : 1..100 \mid x * x \in 81..100)$	9 <i>Undefined</i> <i>Undefined</i>
$\lambda$	$(\lambda x : 1..100 \bullet x * x) 9$ $(\lambda x : 1..100 \bullet x * x) 0$ $(\lambda s : S; x : 1..100 \bullet s.x + x)$ $((\mu S \mid y = 2), 30)$	81 <i>Undefined</i> 31
$\theta$	$\{S \mid y < 3 \bullet \theta S\}$ $\{S; S' \mid y = 2 \wedge x' = 9 \bullet \theta S'\}$	$\{\langle x == 1, y == 2 \rangle\}$ $\{\langle x == 9, y == 10 \rangle\}$
$\mathbb{P}_1$	$\mathbb{P}_1(1..2)$ $\mathbb{P}_1(1..0)$	$\{\{1\}, \{1, 2\}, \{2\}\}$ $\{\}$
$\exists_1$	$(\exists_1 x : 1..10 \bullet x = 3)$ $(\exists_1 x : 1..10 \bullet x < 3)$	<i>true</i> <i>false</i>
<i>partition</i>	$\langle 1..4, 7..10, 5..6 \rangle$ <i>partition</i> (1..10) $\langle 1..4, 8..10, 5..6 \rangle$ <i>partition</i> (1..10)	<i>true</i> <i>false</i>
<i>disjoint</i>	<i>disjoint</i> $\langle 1..4, 7..9, 5..6 \rangle$ <i>disjoint</i> $\langle 1..4, 7..9, 5..7 \rangle$	<i>true</i> <i>false</i>
$\uparrow$	$\langle 1, 3, 5, 7 \rangle \uparrow \langle 3..6 \rangle$	$\langle 3, 5 \rangle$
<i>squash</i>	<i>squash</i> $\{(2, 1), (5, 2)\}$ <i>squash</i> $\{(2, 1), (2, 9), (5, 2)\}$	$\langle 1, 2 \rangle$ <i>Undefined</i>
$\frown$	$\frown / \langle \rangle$ $\frown / \langle \langle 1, 3 \rangle, \langle 2, 4 \rangle \rangle$	$\langle \rangle$ $\langle 1, 3, 2, 4 \rangle$
$\mathbb{N} \cap \text{finiteset}$	$\mathbb{N} \cap \{2, 0, -2\}$ $\{2, 0, -2\} \cap \mathbb{N}$	$\{0, 2\}$ $\{0, 2\}$
<i>finiteset</i> ∈ $\mathbb{PN}$	$\{0, 3\} \in \mathbb{PN}$ $\{0, -3\} \in \mathbb{PN}$	<i>true</i> <i>false</i>
$\dots \in (1..2) \rightsquigarrow \mathbb{N}$	$\{(1, 11), (2, 12)\} \in ((1..2) \rightarrow \mathbb{N})$ $\{(1, 11), (2, 12)\} \in ((1..3) \rightarrow \mathbb{N})$ $\{(1, 11), (2, 11)\} \in ((1..2) \rightsquigarrow \mathbb{N})$	<i>true</i> <i>false</i> <i>false</i>
$n \in (\mathbb{N} \setminus \text{finiteset})$	$101 \in (\mathbb{N} \setminus (-100..100))$ $100 \in (\mathbb{N} \setminus (-100..100))$	<i>true</i> <i>false</i>
$n \in (\text{finiteset} \setminus \mathbb{N})$	$-100 \in ((-100..100) \setminus \mathbb{N})$ $100 \in ((-100..100) \setminus \mathbb{N})$	<i>true</i> <i>false</i>