Jaza User Manual and Tutorial

Mark Utting
The University of Waikato
Hamilton, New Zealand
Email: marku@cs.waikato.ac.nz

June 15, 2005

Jaza stands for *'Just Another Z Animator'*. It is an open-source animator for the Z specification language. It is written in Haskell, which is a modern strongly-typed functional programming language.[1] The current version of Jaza has several major limitations:

- it supports only LaTeX Z notation for input and output. This is not very readable for Z novices, but good for Z experts who want to cut and paste between Jaza and the LaTeX source of their Z specifications. However, it might be nice to support the Z standard email syntax or provide a formatted graphical view of the output, as well as supporting LaTeX notation.

- it does not support all of Z (generics and axiomatic definitions are the main omissions). See Section 7 for more details.

- it does not have a graphical user interface!

The Community Z Tools (CZT) project is developing a new suite of Z tools, including an animator, that will fix these restrictions. See http://czt.sourceforge.net for details.

---

[1]See http://www.haskell.org.

1 Introduction and Motivation

The goal of writing a Z specification for a system is to give a precise description of some aspect of the system's behaviour. The specification can be checked to ensure that it contains no obvious errors (syntax errors, type errors, etc.). However, this does not guarantee that the specification describes the system that we really want. It is almost as easy to make mistakes or misunderstanding in specifications as it is in programs. There are four main ways of validating a specification to check that it correctly captures our requirements:

Inspection: Z experts, preferably together with domain experts, read the specification and manually cross-check it against the requirements.

Proof: Important requirements are formalized and then we prove that they follow from the specification. There are several good proof tools available to help with such proofs.[2]

Execution: Some Z specifications may happen to be written in a form that is executable. That is, given some input values for an operation, a tool such as Jaza can calculate the output values automatically. In this case, it is possible for users to interact with the specified system and try various sequences of operations, checking to see if they have the desired behaviour. However, not all Z specifications are executable in this fashion, and it is often better style to write specifications in a non-executable fashion [HJ89].

Testing: If values are given for all the inputs and outputs of a specified operation, then Jaza can usually evaluate the specification of that operation to either true or false. So a useful validation technique is to provide for each operation several positive test sets (which should cause the specification to evaluate to true) as well as several negative test sets (which should evaluate to false). This is a powerful validation technique, since it allows the behaviour of the operation to be bounded on both sides.

Ideally, all these techniques should be used together to validate a specification. Inspection is a cost-effective way of uncovering errors and is usually

---

[2]See `http://archive.comlab.ox.ac.uk/z.html` for a list of Z tools.

cheaper to apply than proof. Proof is the only technique that can give guarantees about all possible behaviours of the system. The usual caveats about testing apply to the last two techniques: they can show the presence of errors in the specification, but cannot guarantee the absence of all errors.

Jaza is aimed at supporting the last two validation techniques: execution and testing. It is designed to complement a type-checker and prover, such as Z/EVES [Saa97], by providing more efficient and convenient evaluation of schemas on ground data values.

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.). The following sections describe how Jaza supports each of these techniques.

2 Starting Jaza

Jaza can be used with either the Hugs interpreter, or the GHC compiler (see `http://www.haskell.org` to obtain these systems). Jaza will be much faster if you use GHC, but the Hugs version is quite usable for most Z specifications, though you will need to increase the heap size for most specifications.

NOTE: To set the heap size with Hugs, you can start Hugs with a `-h20m` flag, or set the HUGSFLAGS environment variable to `"-h20m"`. The 20m means 20 million *cells*—since cells are usually two integers each, this is about 160 megabytes on a 32 bit machine. On Windows systems, you can type `:set -h20m` within hugs (which saves that setting in the Windows registry), and then restart hugs for the new setting to take effect. With the binary version of Jaza, which is compiled with GHC, the heap will start small (which means lots of garbage collections) then grow as big as necessary. If you want to experiment with a large fixed-size heap (to reduce the initial garbage collections), then you can either set the environment variable `GHCRTS='-M500m'` (for 500 megabytes) or adding these flags to the command line `+RTS -M500m -RTS`.

If you have a version of Jaza that has been compiled with GHC, you can start Jaza by clicking on the `jaza.exe` program under Windows 95/NT, or just typing `jaza` on Linux or UNIX systems (you may need to set your

PATH environment variable to include the Jaza directory). For example, under Linux:

```
export PATH=$PATH:/Directory-Where-Jaza-Is-Installed
jaza
```

If you are using the Hugs interpreter under Windows 95/NT, you should be able to just double click on the `TextUI.hs` file in the Jaza directory. Under Linux/UNIX, the easiest way of starting Jaza is to use the command line version of Hugs:

```
export HUGSFLAGS="-h20m"
runhugs /Directory-Where-Jaza-Is-Installed/TextUI.hs
```

An alternative way is to start up the interactive version of Hugs, then load the `TextUI.hs` module, then type 'main'.

```
export HUGSFLAGS="-h20m"
hugs
Prelude> :load TextUI
Main> main
Welcome to Jaza, version ...
JAZA>
```

Whichever way you choose, you should see the `JAZA>` prompt. Typing `help` at this prompt will show a summary of the Jaza commands. To exit Jaza, just type `quit`. In this manual, all Jaza input and output will be shown in typewriter font.

3   Evaluating Expressions

To evaluate a Z expression, you use the `eval E` command. The expression E must be written in LaTeX notation. It can spread across multiple lines if the characters after the `eval` command contain more opening brackets ((, [, or {) than closing brackets. In this case, Jaza will prompt for additional lines of input, until the right number of closing brackets are found. (During the entry of multi-line input, the Jaza prompt changes to six spaces, so that the input lines look properly aligned, but can still be selected by dragging a

mouse over the whole paragraph. This makes it easy to cut and paste large expressions between Jaza and other documents.)

For example:

```
JAZA> eval 123456 * 1000000
123456000000
JAZA> eval \{ x : 0 \upto 10
            @ x*x \}
\{0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}
```

Similarly, to evaluate a Z predicate, you use the `evalp P` command:

```
JAZA> evalp 12 < 12
False
JAZA> evalp 2 \in \dom \langle 3, 5, 7 \rangle
True
```

Warning: Jaza currently does not perform Z typechecking on the Z constructs that you enter. If you enter incorrectly typed terms, it will either return with an error (not always a type error), or it will give a sensible result according to untyped set theory, even though that result may not have a legal Z type. For example:

```
JAZA> eval \{3,3,2\} \cup \{ (1,4) \}
\{2, 3, (1, 4)\}
JAZA> evalp 3 \in 5
Coercion Problem: cannot convert: ZInt 5, into finite set
```

4   Entering Specifications

You can enter Z specifications into Jaza in two ways:

- Type each Z paragraph directly into the Jaza prompt. Each paragraph can span several lines, but must start with \begin{...} and end with a matching \end{...}, where the ... can be any Z LaTeX environment, such as zed, schema, axdef or gendef. Syntax errors will not be detected until you have correctly entered the \end{...} command.

5

- Load a specification from a file. In this case, Jaza ignores all text in the file, except for the Z paragraphs such as \begin{zed} ... \end{zed}. Each load command simply adds more definitions onto the stack of definitions within Jaza.

The recommended style is to write your Z specification in one or more separate files, use a typechecker (such as Z/EVES) to check the specification, then the `load` command of Jaza to load the files in the right order (so that names are defined before they are used).

Jaza provides several commands for inspecting and undoing a specification that you have entered:

'show': This displays a summary of the whole specification, with one definition per line.

'show N': This displays the internal 'unfolded' form of the schema or defined name 'N'.

'pop': This removes the most recently entered Z definition.

'reset': This removes the entire specification and resets Jaza to its starting state.

For a running example in this manual, I use the standard BirthdayBook system from Spivey [Spi92].

Here is an example of entering the first two paragraphs directly into Jaza.

```
JAZA> \begin{zed}
         [NAME, DATE]
      \end{zed}
Added 2 definitions.
JAZA> \begin{schema}{BirthdayBook}
          known: \power NAME \\
          birthday: NAME \pfun DATE
      \where
          known=\dom birthday
      \end{schema}
Added 1 definition.
JAZA>
```

Tiring of this, let's load the whole specification from a file called `bbook.zed` (this is included in the Jaza distribution, either at the top level or in the userman directory). Since this file contains the above definitions (of *NAME*, *DATE* and *BirthdayBook*), we must do a `reset` command first – otherwise Jaza would complain that those names are being redefined.

```
JAZA> reset
Specification is now empty.
JAZA> load userman/bbook.zed
load userman/bbook.zed
Loading userman/bbook.zed...
Added 13 definitions.
JAZA>
```

We can now see the expanded forms of the schemas, like this:

```
JAZA> show InitBirthdayBook
[known:\power NAME; birthday:NAME \pfun DATE;
 known = \dom birthday; known = \{\}]
```

Advanced Topic: Note that schemas are displayed in horizontal form (`[...]`) and the contents are simply a list of declarations and predicate tests, with no clear separation between the declaration and predicate parts of the schema. Internally, Jaza uses a generalized schema notation which allows predicates and declarations to be intermixed, so that some predicate tests can be moved in amongst the declarations. Since schemas are evaluated from left to right, this is one technique that is used to improve evaluation efficiency. Schemas are 'optimized' just before being executed, so this output from the `show` command shows the schema in unoptimized form. It is sometimes interesting to see how a schema has been optimized – you can do this with the `why` command after the execution of the schema.
End of Advanced Topic.

5   Testing Schemas

After Spivey defines the *BirthdayBook* state schema, he gives some example values that satisfy the schema:

$$known = \{\,John, Mike, Susan\,\}$$

$$birthday = \{\, \text{John} \;\mapsto 25\text{--Mar},$$
$$\text{Mike} \;\mapsto 20\text{--Dec},$$
$$\text{Susan} \mapsto 20\text{--Dec}\,\}.$$

This is a useful validation check, because it ensures that the sample data satisfies the state invariant. However, the syntax shown here is somewhat informal, since *John* and *25–Mar etc.* are not legal Z values (no members of the *NAME* and *DATE* given sets have been declared).

Jaza provides a convenient way of creating members of any given set: simply use any string surrounded by double quotes!

To check the above example values, we could use the `do` command to evaluate a schema expression, like this:

```
JAZA> do [BirthdayBook |
      \t1      known = \{\,"John", "Mike", "Susan"\,\} \land \\
      \t1      birthday = \{\,"John"  \mapsto "25-Mar", \\
      \t4                     "Mike"  \mapsto "20-Dec", \\
      \t4                     "Susan" \mapsto "20-Dec"\,\} ]
\lblot birthday==\{("John", "25-Mar"), ("Mike", "20-Dec"),
               ("Susan", "20-Dec")\},
       known==\{"John", "Mike", "Susan"\} \rblot
```

This displays one possible binding that satisfies the schema (in this case, it is the only such binding). The command `do S` essentially returns the set of all bindings that satisfy the schema expression `S`. However, since that set of bindings is typically infinite, it displays only the first binding and leaves any remaining bindings unevaluated until you ask to see them. If a schema has several solutions, we can step through them using the `next` and `prev` commands.

To illustrate this, let us use another schema. It finds factors of x.

FactX
$x, top : \mathbb{N}$
$factors : \mathbb{P}\,\mathbb{N}$

$factors = \{f : 0 \mathinner{..} x \mid (\exists\, g : 0 \mathinner{..} x \bullet f * g = x)\}$
$x < top$

If we try to find a binding for *FactX*, with $x = 12$, using the command do [FactX|x=12], Jaza gives an error, because *top* is an arbitrary natural number, so the search space is too large (infinite, in this case).

```
JAZA> do [FactX | x=12]
Problem: cannot convert: \{x:\num | x \geq 13\}
                   into finite set
```

This illustrates how, to limit memory required for each evaluation and ensure quick response times, Jaza places limits on the maximum size of any set (`maxset`) and the maximum search space (`maxsearch`). When any of these limits are exceeded, a failure message is returned instead of a normal result. The current values of these limits can be viewed and changed via the `set` command.

To see more clearly why this schema was too difficult to animate, we can use the `why` command to display the optimised form of the schema which Jaza tried to execute.

```
JAZA> why
\begin{schema}{[FactX | x=12]}
1  x==12:\nat
2  factors==\{f:0 \upto x; g:0 \upto x; f * g = x @ f\}:\power \nat
3  top:\bigcap \{\nat, (_ ZGreaterThan x)\}
\end{schema}
The maximum number of true constraints was 2.
```

Jaza tries to execute this sequence from top to bottom (and left to right within each line), so the initial $x == 12 : \mathbb{N}$ and *factors* $== \ldots : \mathbb{P}\,\mathbb{N}$ lines mean that it has calculated unique solutions for these two variables (the : *Type* part checks that the solution has the correct type). However, line 3 is trying to choose a value of *top* from the set $\mathbb{N} \cap (12 \ldots \inf)$, which is an infinite set. This shows that we need to give more constraints on *top*, or specify its value. The last line of output tells us that Jaza managed to find solutions to the first two constraints, but not the third one (the *top* : ...).

For example, if we set top to 1000, the search succeeds.

```
JAZA> do [FactX | x=12 \land top=1000]
\lblot factors==\{0\}, top==1000, x==12 \rblot
```

9

A more interesting experiment is to try searching for the factors of an arbitrary x. For example, if we set top to 10, Jaza chooses an arbitrary x within the allowable range $(0 \mathinner{.\,.} 10)$ and displays its factors:

```
JAZA> do [FactX | top=10]
    \lblot factors==\{0\}, top==10, x==0 \rblot
```

Since this schema has more than one solution, we can step through them using `next` and `prev`, or use `state N` to jump to the N'th solution (if there are less than N solutions, it stops at the last solution).

```
    JAZA> next
    \lblot factors==\{1\}, top==10, x==1 \rblot
    JAZA> next
    \lblot factors==\{1, 2\}, top==10, x==2 \rblot
    JAZA> state 1000
    No more solutions
    JAZA> curr
    \lblot factors==\{1, 3, 9\}, top==10, x==9 \rblot
    JAZA> prev
    \lblot factors==\{1, 2, 4, 8\}, top==10, x==8 \rblot
```

All these solutions have the same *top* value: 10. Usually we do not want to see the values of fields that we have just given exact values for, so we use the Z/EVES *replacement* operator to set *top* to 10. The term $S[n := E]$ is equivalent to the schema expression

$$[S \mid n = E] \setminus (n).$$

```
    JAZA> do FactX[top := 10]
    \lblot factors==\{0\}, x==0 \rblot
    JAZA> state 8
    \lblot factors==\{1, 2, 4, 8\}, x==8 \rblot
```

Note that the `do` command treats all names equally (it does not depend upon the $x, x', x?, x!$ naming conventions of Z), so you can use it for a variety of purposes:

- to search for all solutions to a schema;

- to execute a schema 'forwards' (specify values for the inputs, then let it search for possible output values);

- to execute a schema 'backwards (specify values for the outputs, then let it search for possible input values);

- to test particular input and output values (specify values for every name, and see if those values satisfy the schema or not);

- to explore special cases of the schema (add predicates that restrict the set of solutions).

Here is a typical sequence of 'tests' for the *FactX* schema. The first five are positive tests that satisfy the schema predicate, so produce a single empty binding. The remainder are negative tests that contradict the schema predicate, so Jaza correctly reports that they have no solution.

```
JAZA> % Positive Tests for FactX
JAZA> do FactX[top := 1, x := 0, factors := \{0\}]
\lblot \rblot
JAZA> do FactX[top := 10, x := 0, factors := \{0\}]
\lblot \rblot
JAZA> do FactX[top := 10, x := 1, factors := \{1\}]
\lblot \rblot
JAZA> do FactX[top := 10, x := 5, factors := \{1,5\}]
\lblot \rblot
JAZA> do FactX[top := 10, x := 6, factors := \{1,2,3,6\}]
\lblot \rblot
JAZA> % Negative Tests for FactX
JAZA> do FactX[top := -3]
No solutions
JAZA> do FactX[top := 1, x := 1]
No solutions
JAZA> do FactX[top := 10, x := 6, factors := \{1,6\}]
No solutions
JAZA> do FactX[top := 10, x := 6, factors := \{\}]
No solutions
JAZA> do FactX[top := 10, x := 6, factors := \{1,2,3,4,5,6\}]
No solutions
```

This style of 'testing' schemas is the main purpose of Jaza. I suggest that every operation schema that you write should have several positive and negative tests to test its behaviour. It is also useful to write some positive and negative tests for your state schemas, to check that your state invariant is correct. Of course, your initialisation schema is one such positive test, but adding some negative tests can be a useful way of checking that the invariant is strong enough.

6 Executing Specifications

If a specification has an initialization schema (say *Init*) plus several operation schemas (say $Op1 \dots OpN$), and follows the usual Z conventions for sequential systems (x/x' for initial/final state variables, $x?$ for inputs and $x!$ for outputs), then we can try to 'execute' it like this:

```
JAZA> do Init'
JAZA> ; Op3
JAZA> ; Op1[in? := 4]
JAZA> ; Op3
```

How does this work? The essence of 'executing' a specification is that we have some current 'system state', and we apply various operation schemas to this system state to transform it into a new state. Some of the operations may take inputs (names ending with ?) and/or produce outputs (names ending with !), in addition to changing the current state of the system.

In Section 5 we saw that the `do` command displays just one binding at a time. This binding is treated as the *current system state* and is used as the input state whenever we apply an operation schema using the *sequential composition* command ';'. So the result of the `do Init'` command is a current system state that contains values for each of the primed variables in $Init'$. (If you forget the prime, the state will contain unprimed variables, which will not be usable as inputs of the next command. So when you type `; S` next, you will be prompted to enter values for all these state variables!)

The `; S` command sequentially composes the current system state with S, prompts the user for any remaining inputs (unprimed variables or variables that end with a ?), then searches for outputs that satisfy schema S. The first output binding that is found becomes the new system state (but alternative

12

output bindings can be explored using the `next` and `prev` commands as usual).

Let us see how this works using the birthday book specification. After loading the specification (see Section 4), we start by finding a solution to the *InitBirthdayBook′* schema (the prime ensures that we have an *output* state).

```
JAZA> do InitBirthdayBook'
\lblot birthday'==\{\}, known'==\{\} \rblot
```

Now we can use the ';' command to sequentially compose an operation schema with this current system state. If we do not specify the input values, the ';' command prompts for them. Alternatively, we can use `:=` to assign values for some or all inputs.

```
JAZA> ; RAddBirthday
    Input name? = "andy"
    Input date? = "1-Jan"
\lblot birthday'==\{("andy", "1-Jan")\}, known'==\{"andy"\},
        result!==ok \rblot
JAZA> ; RAddBirthday[name? := "beth"]
    Input date? = "31-Dec"
\lblot birthday'==\{("andy", "1-Jan"), ("beth", "31-Dec")\},
        known'==\{"andy", "beth"\}, result!==ok \rblot
JAZA> ; RAddBirthday[name? := "carl", date? := "31-Dec"]
\lblot birthday'==\{("andy", "1-Jan"), ("beth", "31-Dec"),
                ("carl", "31-Dec")\},
        known'==\{"andy", "beth", "carl"\}, result!==ok \rblot
```

After entering this data, we use the *FindBirthday* operation to check when various people have birthdays. As expected, we find that asking for `"andy"` returns the date `"1-Jan"`, while asking for `"mark"` causes the *FindBirthday* operation to fail because its precondition is false.

```
JAZA> ; FindBirthday
    Input name? = "andy"
\lblot birthday'==\{("andy", "1-Jan"), ("beth", "31-Dec"),
                ("carl", "31-Dec")\},
        date!=="1-Jan", known'==\{"andy", "beth", "carl"\} \rblot
```

```
JAZA> ; FindBirthday[name? := "mark"]
No solutions
```

When an operation returns no solutions, there is no 'current system state' any more, so we cannot continue composing more operations. To overcome this, we could start again (`do InitBirthdayBook'`), or we can use the `undo` command to step back over the unsuccessful call to *FindBirthday* and recover the old system state.

```
JAZA> undo
undone operation: FindBirthday[name? := "mark"]
JAZA> curr
\lblot birthday'==\{("andy", "1-Jan"), ("beth", "31-Dec"),
                ("carl", "31-Dec")\},
        date!=="1-Jan", known'==\{"andy", "beth", "carl"\} \rblot
```

If we use the (non-deterministic) *RemindOne* operation to find someone who has a birthday on the 31st of December, we get two solutions.

```
JAZA> ; RemindOne[today? := "31-Dec"]
\lblot birthday'==\{("andy", "1-Jan"), ("beth", "31-Dec"),
                ("carl", "31-Dec")\},
        card!=="beth", known'==\{"andy", "beth", "carl"\} \rblot
JAZA> next
\lblot birthday'==\{("andy", "1-Jan"), ("beth", "31-Dec"),
                ("carl", "31-Dec")\},
        card!=="carl", known'==\{"andy", "beth", "carl"\} \rblot
JAZA> next
No more solutions
```

Of course, if we use *RRemind* instead, we see the set of all people in the first (unique) solution of the operation. Note that the `next` command leaves the current system state unchanged when it fails, so the input state to this *RRemind* call is the last state shown above.

```
JAZA> ; RRemind[today? := "31-Dec"]
\lblot birthday'==\{("andy", "1-Jan"), ("beth", "31-Dec"),
                ("carl", "31-Dec")\},
        cards!==\{"beth", "carl"\},
        known'==\{"andy", "beth", "carl"\},
        result!==ok \rblot
```

7　Limitations of Jaza

Generally, Jaza supports most of Spivey Z (second edition), except for generics, axiomatic definitions and bags. In addition, it supports the Standard Z `\lblot...\rblot` notation for bindings, but does not support all the other features of Standard Z, such as sections. Here is a list of other limitations:

- Input is not type checked. (You should run specifications through an external type checker first).

- Transitive closure: R^+ is implemented, but R^* is not (because it requires type inference to determine the basetype of R).

- Signature compatibility of schemas is more strict in Jaza than in standard Z. The types in the two signatures must be written identically for those signatures to be compatible.

- Theta expressions are not allowed to include schema renaming.

- There is no way of declaring user-defined infix/prefix/postfix functions or relations.

- Jaza gives errors for axiomatic declaration. It is not clear what an animator should do with axiomatic declarations in general, because it means that your specification is really a parameterized family of specifications, and which member of the family should be animated? The usual way to avoid this problem is to replace the axiomatic declaration by a definition, while you are animating the specification.

Acknowledgements

References

[HJ89]   I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.

[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.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall International (UK) Ltd, second edition, 1992.

Appendix B: Summary of Jaza Commands

Note that all expressions, predicates and schema expressions must be input in LaTeX format. The syntax follows Spivey second edition closely, but adds the Z/EVES replacement operator: $S[x := E]$.

The available commands in Jaza include:

help: Display a summary of all the Jaza commands.

quit: Exit the animator.

\begin{...}...\end{...}: Enter a Z paragraph. The paragraph can spread over several lines, because Jaza will continue prompting for input lines until the end command is seen.

load filename: Load a Z specification from a file.

show: Display a summary of all the definitions in the specification.

show N: Display an unfolded definition of the name N.

pop: Delete the last Z paragraph entered.

reset: Reset the whole specification, clearing all previously entered paragraphs.

do S: Execute schema expression S. This displays the a solution to the schema, if one can be found or 'No solutions' if there are no bindings that satisfy the schema. It may also return an error if S could not be evaluated. If a solution is displayed, it becomes the 'current state' of the animator.

next (or 'n'): Show the next state of current schema.

curr: Reshow the current state.

prev (or 'p'): Show the previous state.

state N: Show the N'th state.

; S: Compose schema S with the current state.

undo: Undo the current operation. This removes the 'current state', and returns to the previous 'current state' if there was one. You can type `'curr'` to see that state. Only one level of undo is supported by default, but you can increase this by setting the `undo` parameter higher via the `set` command.

eval E: Evaluate expression E.

evalp P: Evaluate predicate P.

why: Show the optimized form of the current schema operation or the preceeding expression or predicate evaluation. Given a sequence of declarations, $x_1 : T_1; \ldots x_n : T_n$ (probably interspersed with predicates), evaluation is done left-to-right, so the maximum size of the search space is the cross-product of the T_i sets. The optimization attempts to reduce the size of each T_i set, and reorder them so that smaller sets come first. This command allows you to see how successful the optimization was, and perhaps see why a schema or expression cannot be evaluated. In this mode, false and true predicates usually have explanations attached, to show what predicate it was that evaluated to false or true.

checktrue P: Check that P is true, else give an error. This is intended mainly for automated regression testing of Jaza itself.

checkundef P: Check that P is undefined, else give an error. This is intended mainly for automated regression testing of Jaza itself.

set: Show the current settings of Jaza, such as the maximum size of any finite set and the maximum search space size.

set *Param NNN*: Set the parameter called *Param* to the number *NNN*. Current parameters include:

- `undo` (the number of undo levels),
- `maxset` (the maximum size of any finite set), and
- `maxsearch` (the maximum search space).

For example, $\{x, y, z : 1..10 \bullet x+y+z\}$ has a search space of 1000, but a set size of less than 30. These limits are intended to prevent fruitless infinite searches. The default limits are quite small, so you may want to increase them if you have a fast machine or lots of memory. If you increase the limits, you may also need to increase the Haskell heap size with the `-hNNN` flag.

Evaluations that require more resources than these limits allow will return an error message instead of a normal result. The internal engine is designed so that these limits can affect completeness (low limits may make it impossible to evaluate some expressions), but not correctness (for example, reducing the search space will never cause a $\forall$ or $\exists$ to return *true* instead of *false*, or vice versa).

echo msg...: Echo msg (the rest of the line). This is useful for displaying progress messages within Jaza scripts.

% comment: All lines that start with a percent are ignored. This allows you to put comments in your Jaza scripts.

Note that commands like `eval` and `evalp` that take an expression or a predicate as input can be spread over multiple input lines. If the first line contains more opening brackets than closing, then Jaza continues reading input lines until the number of opening and closing brackets match.