Data types

Declaration of basic global data types:

$$[\, NAME, PHONE \,]$$

More complex types can be created using constructors: subset relation, power set, cartesian product, union, functions (mappings), other relations, etc. We will use them later.

The basic types *NAME* and *PHONE* were given "from outside", as their structures and elements of the corresponding sets are not defined explicitly. A data type is declared as basic, when it is not really important what are the objects of the type. We can say that it is a kind of abstraction.

In the case of smaller types (sets), we can use enumeration to define them.

$$Cities = \{Praha, Brno, Plzen, Ostrava, Kladno\}$$

A set can be defined also by its characteristic predicate.

$$Pos = \{x : Int \mid x > 0\}$$

How to declare a function data type:

$$length : NAME \to \mathbb{N}$$

Restricting function values through predicates (constraints):

$$\begin{array}{|l} pow2 : \mathbb{R} \to \mathbb{R} \\ \hline \forall\, x : \mathbb{R} \bullet pow2(x) \geqslant 0 \end{array}$$

Let R is a relation between X and Y, defined as $X \leftrightarrow Y$. Its domain and range are denoted by $\operatorname{dom} R$ and $\operatorname{ran} R$, respectively.

The Z language also supports the following operations:

- Domain restriction of R to A (i.e., the first element of a pair must belong to A): $A \lhd R$.
- Range restriction: $R \rhd B$.
- Domain subtraction (first element must not belong to A): $A \lhd\!\!\!\cdot\, R$.
- Range subtraction: $R \rhd\!\!\!\cdot\, B$.

Transitive closure of the relation R is written as R^+.

Alternative syntax for defining new types looks like this:

$$T ::= c_1 \mid \ldots \mid c_m \mid d_1 \langle\langle E_1 \rangle\rangle \mid \ldots \mid d_n \langle\langle E_n \rangle\rangle$$

where $c_1, \ldots, c_m$ are constants, $d_1, \ldots, d_n$ are constructors, and $E_1, \ldots, E_n$ are set expressions (they may contain already defined types).

Example:

$$nat ::= zero \mid succ\langle\langle nat \rangle\rangle$$

The same type can be defined also in this way (i.e., using this syntax):

$$\begin{array}{|l} zero : nat \\ succ : nat \pfun nat \\ \hline \forall\, n : nat \bullet n = zero \lor \exists\, m : nat \bullet n = succ(m) \end{array}$$

For example, we can define a type of binary trees where leafs are labeled with natural numbers.

$$Tree ::= leaf\langle\langle \mathbb{N} \rangle\rangle \mid branch\langle\langle Tree \times Tree \rangle\rangle$$

Label for a particular node captures the whole subtree rooted by the node (recursively).

However, it is not really clear (established), or at least I do not know, which syntax for declaring types is more common or preferable in the Z language community.

Schemas

Syntax (template):

name

declarations
---
constraints

Schemas are used to define (1) state and (2) operations.

Example of a schema that defines state:

AddressBook

$persons : \mathbb{P}\, NAME$
$contacts : NAME \pfun PHONE$

$persons = \mathrm{dom}\, contacts$

Alternative syntax (linear, usable for tools):

$$AddressBook \mathrel{\widehat=} [\, persons : \mathbb{P}\, NAME, contacts : NAME \pfun PHONE \mid persons = \mathrm{dom}\, contacts \,]$$

Example of a schema that specifies an update operation:

AddContact

$\Delta AddressBook$
$name? : NAME$
$phone? : PHONE$

$name? \notin persons$
$contacts' = contacts \oplus \{name? \mapsto phone?\}$

The Δ symbol indicates (means) that the operation updates the state of the module "AddressBook". Variables having the suffix "?" represent input (arguments of this operation). Variables and terms with the suffix ' (apostrophe) represent the post-state of the operation, i.e. values of state variables after the operation finishes all changes (steps). Variables without the suffix ' represent the pre-state, i.e. values of state variables before the execution of this operation started (values how they were at the time when the operation was called). The map (function) "contacts" is extended with another pair of values.

For operations that just read the current state, we can use the following syntax:

GetPhoneNumber

$$\Xi AddressBook$$
$$name? : NAME$$
$$phone! : PHONE$$
$$\hline$$
$$phone! = contacts(name)$$

The symbol Ξ means that $s' = s$ for every state variable s in the module, i.e. the state is not changed at all by this operation. Suffix "!" indicates output variables (return values).

Inference rules

Template for inference rules:

$$\frac{premisses}{conclusion} \quad [\ name\ ]$$

$$side\ condition$$

Each rule consists of premisses and conclusion. When all the given premisses are satisfied, then the conclusion must also hold. Sometimes the side condition must also hold - it means that an inference rule can be used only when this side condition is true.

Several examples of inference rules follow. They are usable in proving claims about specifications in Z.

Introducing conjunction:

$$\frac{p \qquad q}{p \land q} \quad [\ \land -intro\ ]$$

Elimination of a conjunction:

$$\frac{p \land q}{p} \quad [\ \land -elim1\ ]$$

Note that inference rules can be nested, i.e. they can make a tree that represents a proof derivation.

The Z language and inference engine also supports quantifiers and term substitution.

Universal quantifier:

$$\frac{t \in a \qquad \forall x : a \bullet p}{p[t/x]} \quad [\ \forall -elim\ ]$$

Existential quantifier:

$$\frac{t \in a \qquad p[t/x]}{\exists\, x : a \bullet p} \quad [\, \exists - intro \,]$$

Equivalent objects can be substituted:

$$\frac{s = t \qquad p[t/x]}{p[s/x]} \quad [\, eq - sub \,]$$

To verify some claim, you may also need to use axioms over data types and variables of the given types.

Definition of an axiom looks like this:

$$\begin{array}{|l} x : S \\ \hline p \end{array}$$

Alternative notation (syntax) for axioms:

$$\frac{}{x \in S \land p} \quad [\, axiom \,]$$

Example:

$$\begin{array}{|l} maxsize : \mathbb{N} \\ \hline maxsize > 0 \end{array}$$

Here, *maxsize* is just a variable name.

Global variables and their values can be defined through axioms (e.g., the *maxsize* variable), and then used in any schema.

Other language constructs

Generic schemas (parameterizable) are defined in this way:

$$\begin{array}{|l} \hline name[T] \\ x : S \\ \hline p \\ \hline \end{array}$$

Notice the generic parameter T.

Example (generic stack):

$$\begin{array}{|l} \hline Stack[T] \\ data : LIST[T] \\ top : INT \\ \hline top \geqslant 0 \\ \hline \end{array}$$

In addition, you can also define generic functions and relations.

Example (generic specification of a subset relation over the set X):

$$[X]$$
$$\_ \subseteq \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$$

$$\forall\, s, t : \mathbb{P}\, X \bullet$$
$$\quad s \subseteq t \Leftrightarrow \forall\, x : X \bullet x \in s \Rightarrow x \in t$$

You can use multiple generic formal parameters. Example (function that returns the first element of a tuple):

$$[X, Y]$$
$$\mathit{first} : X \times Y \to X$$

$$\forall\, x : X;\ y : Y \bullet$$
$$\quad \mathit{first}(x, y) = x$$

Unnamed generic schemas (constants, definitions) are often used too. They can be included or embedded inside more complex schemas (with proper instantiation). For example, the schema above defines a large family (collection) of functions named *first*, one for every possible tuple of concrete values of the formal parameters. Concrete return values of these functions are specified by the constraint formula.

Selecting one element of a tuple:

$$(x_1, x_2, \ldots, x_n).m = x_m$$

The tuple can be an element of a data type created using the cartesian product operator.

The Z language also supports the following notation for sequences.
Declaration: $\langle 1, 2, 3, 4 \rangle$.
Concatenation: $\langle 1, 2, 3 \rangle \frown \langle 4, 5 \rangle$.
Filtering (projection): $\langle 1, 2, 3, 4, 1, 2, 3, 4, 5 \rangle \upharpoonright \{1, 2, 5\} = \langle 1, 2, 1, 2, 5 \rangle$.
Length: $\#\langle 1, 2, 3, 1, 2, 3 \rangle = 6$.
Note that a sequence of elements from the set X can be modeled as a function from $\mathbb{N}$ to X.