Z Notation

December 1, 2023

1 Definitions

There are many different ways to introduce an object in a Z specification: declarations, abbreviations, axiomatic definitions, and free types. Keep in mind that the only built-type in Z is $\mathbb{Z}$. Other types such as $\mathbb{N}$ are defined based on $\mathbb{Z}$.

Given a type X, we may introduce an element of such type as $x : X$, which is called a signature. We may also supply constraints to a signature: $x : X \mid x \in \mathbb{N} \land x < 5$.

1.1 Declarations

The declaration $[\mathit{Type}]$ introduces a new basic type called *Type*, which represents a set. We may declare multiple types by separating them by comma: $[\mathit{Type}1, \mathit{Type}2]$.

1.2 Abbreviations

An abbreviation introduces a new type in terms of equality of another existing entity. We may replace the abbreviation with the actual value anywhere in the specification.

For example: $\mathit{Numbers} == \mathbb{N}$, or $\mathit{Primary} == \{\mathit{red}, \mathit{green}, \mathit{blue}\}$. In the last example, *red*, *green*, and *blue* must have been defined elsewhere. If they are elements of set *Colours*, then $\mathit{Primary} : \mathbb{P}\, \mathit{Colours}$. Keep in mind that abbreviations should not introduce recursive definitions.

More complex abbreviations can be introduced using set comprehensions:

$$\mathit{Even} == \{n : \mathbb{N} \mid n \bmod 2 = 0\}$$

Abbreviations may be generic, in which case there will be a set of parameters following the defined symbol. We can define the empty set using a generic abbreviation:

$$\emptyset[S] == \{x : S \mid \mathit{false}\}$$

And then we can talk about the empty set of natural numbers as $\emptyset[\mathbb{N}]$. Notice that the square brackets are optional (so $\emptyset\mathbb{N}$ is valid) but may be used for readability purposes.
Z also allows syntax sugar to define infix generic abbreviations. For example: $s \mathit{\ rel\ } t == \mathbb{P}(s \times t)$ is equivalent to: $\mathit{rel}[s, t] == \mathbb{P}(s \times t)$.

1.3 Axiomatic Definitions

These types of definitions come with a constraint predicate. Such definitions are said to be axiomatic, given the predicate must hold whenever the symbol is used. For example, we can define natural numbers as:

$$\mathbb{N} : \mathbb{P}\,\mathbb{Z}$$
$$\hline$$
$$\forall z : \mathbb{Z} \bullet z \in \mathbb{N} \Leftrightarrow z \geq 0$$

The default constraint of an axiomatic definition is *true*, so:

$$x : S$$

Is the same as:

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

An axiomatic definition can be proved to be consistent with a proof of existence. Given the axiomatic definition $\mathbb{N} : \mathbb{P}\,\mathbb{Z}$ from before, we must prove that $\exists\, \mathbb{N} : \mathbb{P}\,\mathbb{Z} \bullet \forall\, z : \mathbb{Z} \bullet z \in \mathbb{N} \Leftrightarrow z \geq 0$. That it, that there exists an element that adheres to the definition.

Notice that we can define a boolean functions using an axiomatic definition, in which case making use of the operation involves using the existential operator.

$$\begin{array}{|l} crowds : \mathbb{P}(\mathbb{P}\, Person) \\ \hline crowds = \{p : \mathbb{P}\, Person \mid \#s \geq 3\} \end{array}$$

In this case, $\{John, Jane, Jack\} \in crowds$. However, Z supports syntax sugar to express that an axiomatic definition takes an argument:

$$\begin{array}{|l} crowds\_ : \mathbb{P}(\mathbb{P}\, Person) \\ \hline crowds = \{p : \mathbb{P}\, Person \mid \#s \geq 3\} \end{array}$$

And in that case, we can say $crowds(\{John, Jane, Jack\})$.

Given an axiomatic definition, if more than one set satisfies the constraints, then the definition is *loose*.

Axiomatic definitions may be generic, as long as the generic parameters are sets. For example:

$$\begin{array}{|l} [X] \\ \hline \mathbb{P}_1 : \mathbb{P}(\mathbb{P}\, X) \\ \hline \mathbb{P}_1 = \{s : \mathbb{P}\, X \mid s \neq \emptyset\} \end{array}$$

We may also define an infix operator as a generic axiomatic definition:

$$\begin{array}{|l} [X] \\ \hline \_ \subseteq \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X \\ \hline \forall\, s, t : \mathbb{P}\, X \bullet s \subseteq t \Leftrightarrow x \in s \Rightarrow y \in s \end{array}$$

Since the subset symbol is a generic axiomatic definition, given $X, Y : \mathbb{P}\,\mathbb{N}$, we can say $X \subseteq [\mathbb{N}]\, Y$, even though the generic parameters are usually obvious from the context.

1.4 Free Types

Free types are used to model more complex, and potentially recursive, data structures.

Take the following simple example:

$$Colours ::= red \mid green \mid blue$$

This free type defines both *Colours* and each of its elements as *different* elements (so $red \neq blue$, $blue \neq green$, etc). The ordering of the members in the definition not important.

Free types may use constructor functions. These are injective functions that map from a set of constructor arguments to the free type itself. For example:

$$MaybeInt ::= nothing \mid some\langle\langle \mathbb{Z} \rangle\rangle$$

In this case, *some* is a constructor function that maps $\mathbb{Z}$ to *MaybeInt*, and we may define elements such as $some(3) \in MaybeInt$, or $nothing \in MaybeInt$.

Free types can also be recursive by having constructors that take the free type as an argument. For example:

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

A free type may have more than one constructor. For example:

$$Tree ::= leaf \mid branch\langle\langle Tree \times Tree \rangle\rangle$$

A free type is *consistent* if each of its constructors involves Cartesian products, finite power sets, finite functions, and finite sequences. The following free type is not considered to be consistent:

$$T ::= c\langle\langle \mathbb{P}\ T \rangle\rangle$$

2 Functions

Z functions fall into the following categories:

- $\pfun$ Partial
- $\fun$ Total
- $\pinj$ Partial, Injective
- $\inj$ Total, Injective
- $\psurj$ Partial, Surjective
- $\surj$ Total, Surjective
- $\bij$ Partial, Bijective
- $\finj$ Total, Bijective

2.1 Defining Functions

We may define a function using an axiomatic definition and lambda notation:

$$\begin{array}{|l} double : \mathbb{Z} \pfun \mathbb{N} \\ \hline double = \lambda\, m : \mathbb{Z} \mid m \in \mathbb{N} \bullet m + m \end{array}$$

We can also define functions using axiomatic definitions and equivalences:

$$\begin{array}{|l} [X] \\ \hline \# : \mathbb{F} X \fun \mathbb{N} \\ \hline \forall\, s : \mathbb{F} X;\ n : \mathbb{N} \bullet \# s = n \Leftrightarrow \exists\, f : (1..n) \finj s \bullet true \end{array}$$

We can define functions that take more than one arguments by using cartesian products:

$$\begin{array}{|l} max : (\mathbb{N} \times \mathbb{N}) \fun \mathbb{N} \\ \hline \forall\, x, y : \mathbb{N} \bullet (x < y \Rightarrow max(x, y) = y) \land (x \geq y \Rightarrow max(x, y) = x) \end{array}$$

We can define recursive functions as follows:

$$[X]$$
$$reverse : \operatorname{seq} X \to \operatorname{seq} X$$
$$\forall x : \operatorname{seq} X \bullet$$
$$reverse\langle\rangle = \langle\rangle \land$$
$$reverse(\langle x \rangle \frown s) = (reverse(s)) \frown \langle x \rangle$$

2.2 Function Sets

We may define injections as abbreviations:

$$A \rightarrowtail B == \{f : A \pfun B \mid \forall x, y : dom(f) \bullet f(x) = f(y) \Rightarrow x = y\}$$
$$A \inj B == (A \to B) \cap (A \rightarrowtail B)$$

We may also define surjections the same way:

$$A \twoheadrightarrow B == \{f : A \pfun B \mid ran(f) = B\}$$
$$A \surj B == (A \to B) \cap (A \twoheadrightarrow B)$$

Finally, we can also define bijections using injections and surjections:

$$A \bij B == (A \rightarrowtail B) \cap (A \twoheadrightarrow B)$$
$$A \fun B == (A \to B) \cap (A \bij B)$$

The set of all finite functions from A to B is denoted $A \ffun B$. The set of all finite injections from A to B is denoted as $A \finj B$, which can be defined as $A \ffun B \cap A \rightarrowtail B$.

3 Schemas

A schema can be defined inline as $MySchema \mathrel{\widehat=} [declaration \mid predicate]$ or as:

MySchema
declaration
---
predicate

Omitting the predicate has the same result as providing a *true* predicate. Adding more than one predicate in different lines assumes a conjunction between the predicates.

We may use schemas to define composite data types, similar to structs in certain programming languages. Take the following schema as an example:

MySchema
$a : \mathbb{Z}$
$c : \mathbb{P}\,\mathbb{N}$

The type *MySchema* is the set of all possible combinations of the members a and c. Given an instance of *MySchema* called *MyInstance*, we can access its members using dot notation: $MyInstance.a$ and $MyInstance.c$.

Schemas may be generic:

MyGenericSchema$[X, Y]$
$a : X$
$c : \mathbb{P}\, Y$

And could have then defined *MySchema* as $MySchema \mathrel{\widehat=} MyGenericSchema[\mathbb{Z}, \mathbb{N}]$.

Schemas may impose constraints. For example:

MySchema

$$a : \mathbb{Z}$$
$$c : \mathbb{P}\,\mathbb{N}$$
$$a > 5$$

In this case, we can only instantiate *MySchema* if its member a is greater than 5.

A schema may be used as the declaration part of a lambda expression:

$$foo == \lambda\, MySchema \bullet a^2$$

3.1 Set Comprehensions

Using a schema in a set comprehension introduces all its members into the scope. Given a schema *MySchema* with members a and c, we can write the following comprehension to return the set of values of c where the instance had $a = 0$:

$$\{MySchema \mid a = 0 \bullet c\}$$

This is logically the same as:

$$\{x : MySchema \mid x.a = 0 \bullet x.c\}$$

3.2 Bindings

We can instantiate a schema using the concept of bindings. Given the following schema:

MySchema

$$a : \mathbb{N}$$
$$b : \mathbb{N}$$
$$a < b$$

We can instantiate it as:

$$instance : MySchema$$
$$instance == \langle\!|\, a \rightsquigarrow 5, b \rightsquigarrow 7 \,|\!\rangle$$

The bound variables should match the schema declaration, and the chosen values must obey the schema constraints, if any, otherwise the instantiation is undefined. We say that a is bound to 5, and b is bound to 7.

Given a schema *MySchema*, $\theta MySchema$ represents the *characteristic binding* of *MySchema*. If *MySchema* contains two declarations: a and b, then $\theta MySchema = \langle\!|\, a \rightsquigarrow a, b \rightsquigarrow b \,|\!\rangle$. Basically, we create an instance of *MySchema* where its a and b obtain the value of the a and b variables in the current scope. If we have $a = 3$ and $b = 5$ on the current scope, then $\theta MySchema = \langle\!|\, a \rightsquigarrow 3, b \rightsquigarrow 5 \,|\!\rangle$.

Notice that all possible instances of *MySchema* can be expressed as $\{a : \mathbb{N};\ b : \mathbb{N} \bullet \langle\!|\, a \rightsquigarrow a, b \rightsquigarrow b \,|\!\rangle\}$, which is of course the same as $\{a : \mathbb{N};\ b : \mathbb{N} \bullet \theta MySchema\}$, which is in turn equal to $\{MySchema \bullet \theta MySchema\}$. This means that a signature such as $p : MySchema$ is just a shortcut for $p \in \{MySchema \bullet \theta MySchema\}$. Thus we can say p is a binding that happens to match *MySchema*.

3.3 Operations

We can use schemas to model operations. This is done by defining the properties of a state before and after the operation. Consider the following simple schema that defines the state of our system:

SystemState

$$x : \mathbb{Z}$$

An operation schema adds the state in the declaration section. A trailing quote describes the state after the operation, and it introduces members with a trailing quote into the scope:

Multiply

$$SystemState$$
$$SystemState'$$
$$\hline$$
$$x' = x + x$$

Its considered best practice to also introduce a schema that defines the initial state of the system, by adding constraints over the after state:

SystemStateInit

$$SystemState'$$
$$\hline$$
$$x' = 0$$

Z includes a couple of shorthands to describe state transformations. $\Delta SystemState$, used to describe state mutations, results in the following schema:

$\Delta SystemState$

$$SystemState$$
$$SystemState'$$

Therefore we can re-write our *Multiply* operation as:

Multiply

$$\Delta SystemState$$
$$\hline$$
$$x' = x + x$$

Schema operations can define inputs and outputs in the declaration part. Declarations ending with a question sign and with an exclamation sign are considered inputs and outputs, respectively. For example:

Add

$$\Delta SystemState$$
$$n? : \mathbb{Z}$$
$$r! : \mathbb{Z}$$
$$\hline$$
$$x' = x + n?$$
$$r! = x'$$

Given *SystemState* in the scope, we may use $\theta SystemState$ and $\theta SystemState'$ to refer to the combination of state values before or after the operation. We can exploit this to define $\Xi SystemState$, which looks like this:

$\Xi SystemState$

$$\Delta SystemState$$
$$\hline$$
$$\theta SystemState' = \theta SystemState$$

This is useful to define operations that don't mutate the system state, as it concisely states that all the members of the state schema should remain the same after applying the operation. For example:

GetValue

$\Xi SystemState$
$r! : \mathbb{Z}$

$r! = x$

If an operation schema doesn't constraint the states it can be applied to, then its considered to be a total operation. Otherwise its considered to be a partial operation.

3.4 Normalization

Normalization is the process of rewriting a schema such that all the constraint information appears in the predicate part. This might require recursively normalizing other required schemas. For example, consider the following schemas:

S

$a : \mathbb{N}$
$b : \mathbb{N}$

$a \neq b$

T

S
$c : \mathbb{N}$

$b \neq c$

Increment

ΔT
$in? : \mathbb{N}$
$out! : \mathbb{N}$

$a' = a + in?$
$b' = b$
$c' = c$
$out! = c$
$b \neq c$

The normalized form of *Increment* looks like this:

Increment

$a, b, c, a', b', c' : \mathbb{Z}$
$in? : \mathbb{Z}$
$out! : \mathbb{Z}$

$a \geq 0 \land b \geq 0 \land c \geq 0$
$a' \geq 0 \land b' \geq 0 \land c' \geq 0$
$in? \geq 0$
$out! \geq 0$
$a' = a + in?$
$b' = b$
$c' = c$
$out! = c$
$a \neq b$
$a' \neq b'$
$b \neq c$
$b' \neq c'$

Notice we replaced $\mathbb{N}$ with $\mathbb{Z}$ as the latter is the only built-in number type in Z.

3.5 Schema Calculus

We can manipulate schemas in different ways using logical operators. Keep in mind that is always necessary to normalize the schemas before attempting any of these operations.

A conjunction between two schemas is calculated by merging the declarations and making a conjunction of the predicates. Conjunction is undefined if there a conflict in the declarations part. For example, given the following schemas:

Foo

$a : \mathbb{Z}$

$a > 5$

Bar

$a : \mathbb{Z}$
$b : \mathbb{Z}$

$a < 30 \lor b = 4$

The conjunction $Foo \land Bar$ equals:

$a : \mathbb{Z}$
$b : \mathbb{Z}$

$(a > 5) \land (a < 30 \lor b = 4)$

Disjunctions work the same way, except that we join the predicates using a disjunction. Given the above schemas, $Foo \lor Bar$ equals:

$a : \mathbb{Z}$
$b : \mathbb{Z}$

$a > 5 \lor a < 30 \lor b = 4$

Negation works by simply negating the normalized predicate. $\neg\ Bar$ equals:

$$
\begin{array}{|l}
\hline
a : \mathbb{Z} \\
b : \mathbb{Z} \\
\hline
\neg\ (a < 30 \lor b = 4) \\
\hline
\end{array}
$$

3.6 Quantification

Consider the following schema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MySchema}} \\
\hline
p : \mathbb{N} \\
q : \mathbb{P}\,\mathbb{N} \\
\hline
p \in q \\
\hline
\end{array}
$$

Given using a schema introduces its members to the current scope, we can use this schema in quantifier expressions as follows:

$$\exists\, \textit{MySchema} \bullet b = 4$$
$$\forall\, \textit{MySchema} \bullet c > a$$

We can also use it as a quantifier constraint:

$$\forall\, p : \mathbb{N};\ q : \mathbb{P}\,\mathbb{N} \mid \textit{MySchema}$$

The above example will be true if all possible combinations of p and q match the constraints imposed by *MySchema*.

Using a schema as a partial quantifier constraint results in a new schema. Given *MySchema*, $\forall\, q : \mathbb{P}\,\mathbb{N} \bullet \textit{MySchema}$ is the same as:

$$
\begin{array}{|l}
\hline
p : \mathbb{N} \\
\hline
\forall\, q : \mathbb{P}\,\mathbb{N} \bullet p \in q \\
\hline
\end{array}
$$

And $\exists\, q : \mathbb{P}\,\mathbb{N} \bullet \textit{MySchema}$ is the same as:

$$
\begin{array}{|l}
\hline
p : \mathbb{N} \\
\hline
\exists\, q : \mathbb{P}\,\mathbb{N} \bullet p \in q \\
\hline
\end{array}
$$

Notice we keep every non-quantified declaration, and move the quantification to the constraint area, replacing the mention of *MySchema* with any actual constraint.

Keep in mind that a lot of times an existential quantifier can be eliminated by using the one-point rule.

3.7 Renaming

This is a useful feature, based on substitutions, to rename properties of schemas. Take the following schema as an example:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MySchema}} \\
\hline
p : \mathbb{N} \\
q : \mathbb{P}\,\mathbb{N} \\
\hline
p \in q \\
\hline
\end{array}
$$

We can rename p to x and q to y by saying $MyNewSchema \mathrel{\widehat=} MySchema[x/p, y/q]$, which results in the following schema:

MyNewSchema

$$x : \mathbb{N}$$
$$y : \mathbb{P}\,\mathbb{N}$$

$$x \in y$$

Notice the resulting schema, even though only renaming took place, is considered to be different, so comparing *MySchema* and *MyNewSchema* is an undefined operation.

3.8 Hiding

Hiding is a powerful mechanism for schema abstraction. We can use it to "hide" elements of the declaration part that are not required at the current level of specification.

For example, consider the following schema:

MySchema

$$a : \mathbb{N}$$
$$b : \mathbb{N}$$

$$P$$

We can hide a as $MySchema \setminus \{a\}$, which is equal to $\exists\, a : \mathbb{N} \bullet MySchema$:

$$b : \mathbb{N}$$

$$\exists\, a : \mathbb{N} \bullet P$$

Note the close relationship between hiding and schema existential quantification.

3.9 Composition

We can think of schema operations as relations between states of the system. Given a state *State* and two operations *Operation*1 and *Operation*2 that operate on *State*, we can introduce a temporary version of the state called $State''$. Then, if *Operation*1 and *Operation*2 are composed, the result is an operation that maps from *State* to $State''$, and then from $State''$ to $State'$, where $State''$ is hidden.

More formally, if our operations mutate a state containing a and b, then:

$$Operation1 \fatsemi Operation2 =$$
$$(Operation1[a''/a', b''/b'] \land Operation2[a''/a, b''/b]) \setminus \{a'', b''\}$$

Basically, we rename the *after* state of *Operation*1 to be the intermediary state, then rename the *before* state of *Operation*2 to be the intermediary state, conjunct both schemas, and then hide the intermediary state.

3.10 Promotion

Promotion, or framing, is a structuring technique to reduce the specification complexity when modeling operations on a single data type, that take a composite type as an input. For example, given a sequence of complex types and an operation on a single complex type, promotion allows us to define the same operation that acts on a single complex type taking the sequence as an input.

Consider the following schemas:

$$
\begin{array}{|l}
\hline
Data \\
value : Value \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
Array \\
array : \operatorname{seq} Data \\
\hline
\end{array}
$$

And an operation on $Data$:

$$
\begin{array}{|l}
\hline
AssignData \\
\Delta Data \\
new? : Value \\
\hline
value' = new? \\
\hline
\end{array}
$$

We can write the following schema to *promote* $AssignData$ to work on $Array$:

$$
\begin{array}{|l}
\hline
AssignDataPromote \\
\Delta Array \\
\Delta Data \\
index? : \mathbb{N} \\
\hline
index? \in dom(array) \\
\{index?\} \lhd array = \{index?\} \lhd array' \\
array(index?) = \theta Data \\
array'(index?) = \theta Data' \\
\hline
\end{array}
$$

The first constraint ensures that the given index is valid, and exists in the array. The second constraint ensures the elements on the array other than the specified one will remain the same. The remaining constraints state that the element at $index?$ will follow the transformations made to $\Delta Data$.

Notice the promotion schema only specifies the relation between the transformation of a single data element and the transformation of an array of such data elements. The idea is to then calculate $AssignDataPromote \land AssignData$:

$$
\begin{array}{|l}
\hline
\Delta Array \\
\Delta Data \\
index? : \mathbb{N} \\
new? : Value \\
\hline
index? \in dom(array) \\
\{index?\} \lhd array = \{index?\} \lhd array' \\
array(index?) = \theta Data \\
array'(index?) = \theta Data' \\
value' = new? \\
\hline
\end{array}
$$

And then hide $\Delta Data$, as the user is not expected to pass it directly:

$$
\begin{array}{|l}
\hline
\Delta Array \\
index? : \mathbb{N} \\
new? : Value \\
\hline
\exists \Delta Data \bullet \\
\quad index? \in dom(array) \land \\
\quad \{index?\} \lhd array = \{index?\} \lhd array' \land \\
\quad array(index?) = \theta Data \land \\
\quad array'(index?) = \theta Data' \land \\
\quad value' = new? \\
\hline
\end{array}
$$

The resulting schema is effectively a promoted version of *AssignData*.

In conclusion, a promoted operation is defined as $\exists \Delta Local \bullet LocalOperation \land Promote$.

Promotions might be *free* or *constrained*. A promotion is said to be free if the promotion schema satisfies the following statement:

$$(\exists Local' \bullet \exists Global' \bullet Promote) \Rightarrow (\forall Local' \bullet \exists Global' \bullet Promote)$$

Which basically means: "given that the update is possible at all, it is possible for all outcomes of the local state". This is satisfied if neither the promotion schema nor the global state definitions place any additional constraint upon the component variables of the local state schema.

3.11 Preconditions

Given an operation *MyOperation*, pre *MyOperation* refers to the (different) schema that consists of *MyOperation* minus any outputs and components that correspond to the state after the operation:

$$\text{pre } MyOperation = \exists State' \bullet MyOperation \setminus outputs$$

The recipe to calculate these precondition schemas is:

- Expand any required schemas, Δ, Ξ, etc from the schema declaration
- Divide the declaration into 3 parts: before (including inputs), after (including outputs), and everything else

The precondition is then:

Before
$\exists After \bullet Predicate$

Given the following schemas:

S

$a, b : \mathbb{N}$
$a \neq b$

T

S
$c : \mathbb{N}$
$b \neq c$

Increment

ΔT
$in?, out! : \mathbb{N}$
$a' = a + in?$
$b' = b$
$c' = c$
$out! = c$
$b \neq c$

We first need to expand *Increment*:

Increment

$a, b, c, a', b', c', in?, out! : \mathbb{N}$

$a' = a + in?$
$b' = b$
$c' = c$
$out! = c$
$a \neq b \land a' \neq b'$
$b \neq c \land b' \neq c'$

Then we should divide the declaration into 3 parts (2 in this case):

$a, b, c, in? : \mathbb{N}$

$a', b', c', out! : \mathbb{N}$

Then we should hide the *after* part of the schema:

$a, b, c, in? : \mathbb{N}$

$\exists\, a', b', c', out! \in \mathbb{N} \bullet$
$a' = a + in?$
$b' = b$
$c' = c$
$out! = c$
$a \neq b \land a' \neq b'$
$b \neq c \land b' \neq c'$

We can now simplify the quantified definition using the one-point rule:

$a, b, c, in? : \mathbb{N}$

$b \neq a + in?$
$b \neq c$

And we can finally reintroduce T into the declaration, and we will get the final precondition schema:

T
$in? : \mathbb{N}$

$b \neq a + in?$

Its common practice to create tables to store the preconditions of every operation in a specification, for convenience.

Notice the precondition operator distributes over $\lor$:

$$Op \mathrel{\widehat=} Op_1 \lor Op_2 \Rightarrow \text{pre } Op = \text{pre } Op_1 \lor \text{pre } Op_2$$

But it does not distribute over $\land$.

Under a *free promotion*, the precondition of a global operation is the conjunction of the precondition of the local operation and the precondition of the promotion:

$$\text{pre } Global = \text{pre } Local \land \text{pre } Promotion$$

In the case of a *constrained promotion*, the precondition of a global operation is the precondition of the conjunction of the local operation and the promotion:

$$\text{pre } Global = \text{pre } (Local \land Promotion)$$

3.12 Consistency

Every time we define a particular state, such as *SystemStateInit*, the starting point of *SystemState*, we must prove its consistency by showing that there is a set of bindings that adhere to the schema. This is called the *initialization theorem* for a data type. In this case, we must prove that:

$$\exists\, SystemState' \bullet SystemStateInit$$

In order to prove that a specification is consistent, we must prove the initialization theorem for each data type, and calculate the preconditions of each operation.

4 Refinement

Refinement is the process of developing a specification in such a way that it leads us towards a suitable implementation.

This is done by starting with a high-level specification, and then re-writing the schemas to use data structures that start resembling data structures used in real programming languages, checking that the new schemas are consistent with their previous high-level versions. Several refinement steps may be performed to approach executable code.

For example, we may start defining our system state using mathematical sets. After a refinement round, we might switch to sequences (and therefore reimplement all operations based on this new data structure), that more closely resemble arrays.