U O W

Software Requirements, Specifications and Formal Methods

A/Prof. Lei Niu

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Schemas and schema calculus

Steps for Modelling with Z Specification

1. Define the data type
    1. free type
    2. basic type
2. Define the axiomatic descriptions
    1. global variables
    2. global constrains
3. Define the system state schema
    1. local variables
    2. system state invariants
4. Initialise the system state schema
    1. Initialise all local variables and global variables
5. Create full operation schema
    1. successful scenarios (when all pre-conditions are true)
    2. non-successful scenarios (when each pre-condition is false)
    3. combine the successful and non-successful scenarios

Inside the schema boxes

We will use our simple editor example to explain the notation to write a single schema box

- *Define type*: $[CHAR]$

  $$TEXT == \mathrm{seq}\, CHAR$$

- *Define state schema*: the simple editor's state:

$$
\begin{array}{|l}
\textit{Editor} \\
\hline
left, right : TEXT \\
\hline
\#(left \frown right) \leq maxsize \\
\hline
\end{array}
$$

- This schema says that the editor consists of two texts named left and right
- The size of the document never exceeds *maxsize*
- Schema names usually begin with an initial capital letter followed by lowercases letters

Schema inclusion

Then we *define the editor's initial state*

$$
\begin{array}{|l}
\hline
\textit{Init} \\
\textit{Editor} \\
\hline
\textit{left} = \textit{right} = \langle\rangle \\
\hline
\end{array}
$$

- The editor starts up with an empty document
- The *Init* schema includes all the declarations and predicates in the state schema *Editor*.
- A longer version can be

$$
\begin{array}{|l}
\hline
\textit{Init} \\
\textit{left}, \textit{right} : \textit{TEXT} \\
\hline
\#(\textit{left} \frown \textit{right}) \leq \textit{maxsize} \\
\textit{left} = \textit{right} = \langle\rangle \\
\hline
\end{array}
$$

Operation schema

<u>*Define the full operation schema*</u>: Z uses operation schemas to model changes of state

- Axiomatic definition can be used anywhere

$$\mid \; printing : \mathbb{P}\, CHAR$$

$$
\begin{array}{|l}
\hline
\textit{Insert} \\
\Delta Editor \\
ch? : CHAR \\
\hline
ch? \in printing \\
left' = left \frown \langle ch? \rangle \\
right' = right \\
\hline
\end{array}
$$

- Δ tells us that Insert is an operation that changes the state of Editor
- ? Tells use that ch? is the input variable
- The prime ’ tells use the state after the operation

Operation schema

We can decorate the whole schema with the prime.

- Defines a *new schema with the prime*
- All variable names in the *new schema* are *marked with the prime*

$$\begin{array}{|l} \multicolumn{1}{l}{\underline{\;Editor'\;}} \\ left', right' : TEXT \\ \hline \#(left' \frown right') \leq maxsize \\ \hline \end{array}$$

- No prime mark on a global variable *maxsize*, because it is *not a state variable in schema Editor*

Operation schema

- The Δ naming convention is just an abbreviation for the schema that include both the *unprimed "before" state* and the *primed "after" state*.
- *Δ is not an operator*

$\Delta Editor$

$Editor$

$Editor'$

- When we expand it, we obtain

$\Delta Editor$

$left, left', right, right' : TEXT$

$\#(left \frown right) \leq maxsize$

$\#(left' \frown right') \leq maxsize$

UNIVERSITY OF WOLLONGONG AUSTRALIA

Operation schema

- In schemas, the repeated predicate is always true: it holds before and after any operation
- The predicate is an invariant
- So we can expand the insert operation by *replacing the $\Delta Editor$ with its full text*

Insert

$\Delta Editor$
$ch? : CHAR$

$ch? \in printing$
$left' = left \frown \langle ch? \rangle$
$right' = right$

$\Delta Editor$

$left, left', right, right' : TEXT$

$\#(left \frown right) \leq maxsize$
$\#(left' \frown right') \leq maxsize$

Insert

$left, left', right, right' : TEXT$
$ch? : CHAR$

$ch? \in printing$
$\#(left \frown right) \leq maxsize$
$\#(left' \frown right') \leq maxsize$
$left' = left \frown \langle ch? \rangle$
$right' = right$

Operation schema

- The Ξ symbol indicates an operation *where the state does not change*
- The operation only *consumes inputs* or *produce outputs*
- Ξ operation is another name convention.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Xi Editor} \\
\Delta Editor \\
\hline
left' = left \\
right' = right \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Xi Editor} \\
left, left', right, right' : TEXT \\
\hline
\#(left \frown right) \leq maxsize \\
\#(left' \frown right') \leq maxsize \\
left' = left \\
right' = right \\
\hline
\end{array}
$$

Vertical and horizontal schema format

- Z provides different ways to notate schemas
  - Vertical format

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Init}} \\
\textit{Editor} \\
\hline
\textit{left} = \textit{right} = \langle\rangle \\
\hline
\end{array}$$

  - Horizontal format

$$\textit{Init} \boxed{\mathrel{\widehat{=}}} [\, \textit{Editor} \mid \textit{left} = \textit{right} = \langle\rangle \,]$$

In the horizontal format, the schema name appears to the left of the definition system, and the schema body is enclosed within square brackets, with a vertical bar separating the declaration and predicates.

Vertical and horizontal schema format

- Vertical format

$$
\begin{array}{|l}
\hline
\textit{Insert} \\
\Delta \textit{Editor} \\
ch? : \textit{CHAR} \\
\hline
ch? \in \textit{printing} \\
\textit{left}' = \textit{left} \frown \langle ch? \rangle \\
\textit{right}' = \textit{right} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Insert} \\
\Delta \textit{Editor};\ ch? : \textit{CHAR} \\
\hline
ch? \in \textit{printing} \land \textit{left}' = \textit{left} \frown \langle ch? \rangle \land \textit{right}' = \textit{right} \\
\hline
\end{array}
$$

- Horizontal format

$$\textit{Insert} \mathrel{\widehat{=}} [\ \Delta \textit{Editor};\ ch? : \textit{CHAR} \mid ch? \in \textit{printing} \land \textit{left}' = \textit{left} \frown \langle ch? \rangle \land \ldots\ ]$$

Schema conjunction and disjunction

Schema conjunction combines the predicates using the logical connective and (∧)

For example, 12 divided by 5 yields quotient 2 and remainder 2.

Quotient

$$n, d, q, r : \mathbb{N}$$

$$d \neq 0$$
$$n = q * d + r$$

Any problem?

Schema conjunction and disjunction

Schema conjunction combines the predicates using the logical connective and (∧)

For example, 12 divided by 5 yields quotient 2 and remainder 2.

Quotient

$$n, d, q, r : \mathbb{N}$$

$$d \neq 0$$
$$n = q * d + r$$

- 12 = 2 * 5 + 2 (seems ok, but not good enough) because
- 12 = 0 * 5 + 12 (i.e., 12 divided by 5 gives quotient 0 and remainder 12) ✗

Schema conjunction and disjunction

So we shall also say that the remainder is less than the divisor

Remainder

$r, d : \mathbb{N}$

$r < d$

Finally, we can form the complete specification using the schema conjunction operator

$$Division \mathrel{\widehat{=}} Quotient \wedge Remainder$$

Schema conjunction and disjunction

A typical Z style:

- Define requirements separately
- Use the *schema conjunction operator* to *combine the requirements*

Schema conjunction and disjunction

- Schema disjunction combines the predicates using the logical connective or.

- We use *disjunction* to handle *separate cases, especially errors and other exceptional conditions*

- For example, our Division schema is partial: It doesn't say what happens when the divisor d is zero.

- Let's define a DivideByZero schema first.

Schema conjunction and disjunction

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{DivideByZero}}} \\
d, q, r : \mathbb{N} \\
\hline
d = 0 \land q = 0 \land r = 0 \\
\hline
\end{array}$$

- Then we join the normal and exceptional cases to describe the total operation

$$T\_Division \mathrel{\widehat{=}} Division \lor DivideByZero$$

- The specification can be expressed as a single schema box

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{T_Division}} \\
n, d, q, r : \mathbb{N} \\
\hline
(d \neq 0 \land r < d \land n = q * d + r) \lor \\
(d = 0 \land r = 0 \land q = 0) \\
\hline
\end{array}$$

Schema conjunction and disjunction

- Of course, we can combine conjunction and disjunction operators

$$T\_Forward \mathrel{\widehat=} Forward \lor (EOF \land RightArrow \land \Xi Editor)$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Forward}} \\ \hline
left, right, left', right' : TEXT \\
ch? : CHAR \\ \hline
ch? = right_arrow \\
right \neq \langle\rangle \\
\#(left \frown right) \leq maxsize \\
\#(left' \frown right') \leq maxsize \\
left' = left \frown \langle head\ right \rangle \\
right' = tail\ right \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{EOF}} \\ \hline
left, right : TEXT \\ \hline
\#(left \frown right) \leq maxsize \land right = \langle\rangle \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RightArrow}} \\ \hline
ch? : CHAR \\ \hline
ch? = right_arrow \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\Xi \textit{Editor}} \\ \hline
left, right, left', right' : TEXT \\ \hline
\#(left \frown right) \leq maxsize \\
\#(left' \frown right') \leq maxsize \\
left' = left \land right' = right \\ \hline
\end{array}$$

Schema conjunction and disjunction

Merging the declarations, combining the predicates, and simplifying, we obtain a full specification of T_Forward.

$T\_Forward$

$left, right, left', right' : TEXT$
$ch? : CHAR$

$\#(left \frown right) \leq maxsize$
$\#(left' \frown right') \leq maxsize$
$ch? = right\_arrow$
$((right \neq \langle\rangle \wedge left' = left \frown \langle head\ right \rangle \wedge right' = tail\ right) \vee$
$(right = \langle\rangle \wedge left' = left \wedge right' = right))$

Or

$T\_Forward$

$\Delta Editor$
$ch? : CHAR$

$ch? = right\_arrow$
$((right \neq \langle\rangle \wedge left' = left \frown \langle head\ right \rangle \wedge right' = tail\ right) \vee$
$(right = \langle\rangle \wedge left' = left \wedge right' = right))$

Other schema calculus operators

- We can define the ForwardTwo operation via using the *schema composition operator*

$$ForwardTwo \mathrel{\widehat{=}} Forward \mathbin{;} Forward$$

- We can use the *schema piping* for combining operations that communicate through input and output variables.

- For example, we wish to model the low-level interface to console keyboard from which our editor receives its input. The *Get* operation indicates the most recently pressed keyboard

```
┌─ Get ─────────────────
│ ΔConsole
│ ch! : CHAR
├──────────
│ ...
└───────────────────────
```

Other schema calculus operators

- Then we could form a new operation G_Insert by piping the output of *Get* to the input of *Insert*:

$$G\_Insert \mathrel{\widehat{=}} Get \gg Insert$$

- The pipe operator >> ensures that the output and input variables ch! And ch? are merged

Schema types

We have defined four kinds of fundamental data types

- *Free types*, i.e., listing all members

$$OP \ ::= plus \mid minus \mid times \mid divide$$

- *Basic types*, declared as in [X], instances are individuals

- *Set types*, declared as in P X, instances are sets

- *Cartesian product types*, declared as in X * Y, whose instances are tuples

Schema binding

- A binding is the formal realization of what we have been calling *a situation or a state*
- A binding is *an assignment of particular values* to *a collection of named variables*
- A binding resembles a tuple in that it is a composite object whose components can have different types.
- Components of a binding are distinguished by name.

An calendar example

- We try to define *an appointment calendar*
- The calendar includes *the current day, month and year*
- We need to define *an operation* the *advances the date by one day*
- First, we define data types

$$DAY == 1 \,..\, 31$$

$$MONTH == 1 \,..\, 12$$

$$YEAR == \mathbb{Z}$$

$$DATE == DAY \times MONTH \times YEAR$$

An calendar example

- We also need to know how many days in a particular month
  - Some months have 30 or 31 days
  - Feb has 28 days and 29 days in a leap year
  - We define the function *days*

$$days == \{1 \mapsto 31, 2 \mapsto 28, \ldots, 12 \mapsto 31\}$$

- Then we can define the function *next* that takes a DATE and returns the next DATE

$$\begin{array}{|l}
next : DATE \rightarrowtail\!\!\!\!\!\!+\ DATE \\
\hline
\forall d : DAY;\ m : MONTH;\ y : YEAR \bullet \\
\quad (d < days\ m \land next(d, m, y) = (d + 1, m, y)) \lor \\
\quad (d = days\ m \land m < 12 \land next(d, m, y) = (1, m + 1, y)) \lor \\
\quad (d = days\ m \land m = 12 \land next(d, m, y) = (1, 1, y + 1))
\end{array}$$

- Problems with this definition ?

An calendar example

- We also need to know how many days in a particular month
  - Some months have 30 or 31 days
  - Feb has 28 days and 29 days in a leap year
  - We define the function *days*

$$days == \{1 \mapsto 31, 2 \mapsto 28, \ldots, 12 \mapsto 31\}$$

- Then we can define the function *next* that takes a DATE and returns the next DATE

$$
\begin{array}{|l}
next : DATE \twoheadrightarrow DATE \\
\hline
\forall d : DAY;\ m : MONTH;\ y : YEAR \bullet \\
\qquad (d < days\ m \land next(d, m, y) = (d + 1, m, y)) \lor \\
\qquad (d = days\ m \land m < 12 \land next(d, m, y) = (1, m + 1, y)) \lor \\
\qquad (d = days\ m \land m = 12 \land next(d, m, y) = (1, 1, y + 1))
\end{array}
$$

- Problems with this definition ?   (the leap year)

Calendar example

- Then we can define a TypicalDate schema

TypicalDate

$day : 1 \,..\, 31$
$month : 1 \,..\, 12$
$year : \mathbb{Z}$

$day \leq days\ month$

- However, it is not a normalized schema because we just specify the range but not the type of variables

Calendar example

- Then we can define a normalized TypicalDate schema

$$
\begin{array}{|l}
\textit{TypicalDate} \\
\hline
day, month, year : \mathbb{Z} \\
\hline
day \in 1 \,..\, 31 \\
month \in 1 \,..\, 12 \\
day \leq days\ month \\
\hline
\end{array}
$$

- However, the TypicalDate schema does not cover the leap year, i.e., there is an extra day, February 29th.

- Define another schema to cover the exceptional case, then use schema disjunction to combine the cases

Calendar example

- We define a prefix unary relation leap_, so that “leap year” is true when year is a leap year
- Our “*leap_*” is just a set of years: leap years occur every four years, excluding only centuries not divisible by 400 (2000 is a leap year, but not 1900)

$$(leap\_) == \{ y : \mathbb{Z} \bullet 4 * y \} \setminus (\{ y : \mathbb{Z} \bullet 100 * y \} \setminus \{ y : \mathbb{Z} \bullet 400 * y \})$$

$$\begin{array}{|l}
\textit{Feb29} \\
\hline
month, day, year : \mathbb{Z} \\
\hline
month = 2 \\
day = 29 \\
leap\ year \\
\hline
\end{array}$$

$$Date \mathrel{\widehat{=}} TypicalDate \lor Feb29$$

Calendar example

- So the normalized Date Schema is:

$$
\begin{array}{|l}
\hline
\textbf{Date} \\
day, month, year : \mathbb{Z} \\
\hline
(day \in 1 \mathinner{..} 31 \land day \leq days\ month \land month \in 1 \mathinner{..} 12) \lor \\
(day = 29 \land month = 2 \land leap\ year) \\
\hline
\end{array}
$$

Calendar example

- Let's define the *<u>weekday function</u>* which tells use the day of the week for any date
- For example, July 20, 1967 was a Sunday
- We represent the days with numbers:
  - Sunday is 0, Monday is 1, and so on

$$
\begin{array}{|l}
weekday : Date \rightarrow 0 \mathbin{..} 6 \\
\hline
\forall\, d : Date \bullet weekday(d) = (d.day + d.year + d.year \operatorname{div} 4 + \ldots) \bmod 7
\end{array}
$$

- Problem !
- Date is a whole set, but the argument of weekday is a single binding

Calendar example

- To solve this problem, Z provides the binding formation operator θ (the Greek letter theta).
- θ Date refers to the member of Date that is currently in scope

$$
\begin{array}{|l}
weekday : Date \rightarrow 0 \mathrel{..} 6 \\
\hline
\forall Date \bullet weekday(\theta Date) = (day + year + year \operatorname{div} 4 + \ldots) \bmod 7
\end{array}
$$

- θ operator can also be used in predicates to avoid some details

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Xi Editor} \\
\hline
\Delta Editor \\
\hline
\theta Editor' = \theta Editor \\
\hline
\end{array}
$$

Generic definitions

- Z also provides generic constructs that enable you to write definitions that apply to any type

- For example, we used the concatenation operator $\frown$ to join texts together: left’ = left $\frown$ <head right>

- We can define a generic concatenation

$$
\begin{array}{|l}
[X] \\
\hline
\_ \frown \_ : \operatorname{seq} X \times \operatorname{seq} X \to \operatorname{seq} X \\
\hline
\ldots
\end{array}
$$

- X is a formal generic parameter that stands for any type.

Generic schema

- Z also provides generic schemas
- We define a generic schema Pool which contains the generic parameter RESOURCE
- RESOURCE can be used to represent any resource, such as memory pages, disk blocks, or processors

$$
\begin{array}{|l}
\textit{Pool}\ [\textit{RESOURCE}] \\
\hline
\textit{owner} : \textit{RESOURCE} \pinj \textit{USER} \\
\textit{free} : \mathbb{P}\,\textit{RESOURCE} \\
\hline
(\mathrm{dom}\ \textit{owner}) \cup \textit{free} = \textit{RESOURCE} \\
(\mathrm{dom}\ \textit{owner}) \cap \textit{free} = \emptyset \\
\hline
\end{array}
$$

Formal reasoning

So far, we introduced how to use Z to define types and schemas. However, how do we know the schemas and operations are correct and can represent the user's requirements without a missing.

We can use formal reasoning to validate a mathematical model against requirements

A model is called *valid* if *its properties satisfy the intent of the requirements from users*.

Formal reasoning can also show how one model is related to another, and can verify that code implements its specification.

In Z, an exercise in formal reasoning is also called a proof.

Calculation and proof

- Formal reasoning means reasoning with formulas.
- The arithmetic calculations are examples of formal reasoning.
- For example, "A train moves at a constant velocity of sixty km per hour. How far does the train travel in four hours?"

- To solve this problem we can express the problem in Z

$$
\begin{array}{|l}
\mathit{distance}, \mathit{velocity}, \mathit{time} : \mathbb{N} \\
\hline
\mathit{distance} = \mathit{velocity} * \mathit{time} \\
\mathit{velocity} = 60 \\
\mathit{time} = 4
\end{array}
$$

Calculation and proof

- A formal proof of the distance = 240 km can be:

$$
\begin{aligned}
distance &= velocity * time && [Definition] \\
&= 60 * time && [velocity = 60] \\
&= 60 * 4 && [time = 4] \\
&= 240 && [\text{Arithmetic}]
\end{aligned}
$$

- If we change the problem to say velocity < 60 km/hour, then the proof becomes:

$$
\begin{aligned}
distance &= velocity * time && [\text{Definition}] \\
&< 60 * time && [velocity < 60] \\
&= 60 * 4 && [time = 4] \\
&= 240 && [\text{Arithmetic}]
\end{aligned}
$$

Calculation and proof

- Calculations need not be arithmetic
- Set operators can also be used in the proof

$$
\begin{array}{|l}
philip : PERSON \\
adhesives, materials, research, manufacturing : \mathbb{P}\, PERSON \\
\hline
adhesives \subseteq materials \\
materials \subseteq research \\
philip \in adhesives
\end{array}
$$

- Can we proof Philip works in the research division?

$$
\begin{aligned}
philip &\in adhesives && \text{[Definition]} \\
&\subseteq materials && \text{[Definition]} \\
&\subseteq research && \text{[Definition]}
\end{aligned}
$$

UNIVERSITY OF WOLLONGONG AUSTRALIA

Calculation and proof

- Another example to find the value of x, given 2x + 7 = 13

$$\begin{array}{|l} x : \mathbb{Z} \\ \hline 2 * x + 7 = 13 \end{array}$$

We simply solve for x

$$
\begin{aligned}
& 2 * x + 7 = 13 && \text{[Definition.]} \\
& \quad \Leftrightarrow 2 * x = 13 - 7 && \text{[Subtract 7 from both sides.]} \\
& \quad \Leftrightarrow 2 * x = 6 && \text{[Arithmetic.]} \\
& \quad \Rightarrow (2 * x) \operatorname{div} 2 = 6 \operatorname{div} 2 && \text{[Divide both sides by 2.]} \\
& \quad \Leftrightarrow x = 6 \operatorname{div} 2 && \text{[Division on left side, algebra]} \\
& \quad \Leftrightarrow x = 3 && \text{[Division on right side, arithmetic]}
\end{aligned}
$$

This completes our proof of the predicate $2 * x + 7 = 13 \Rightarrow x = 3$.

Laws

- A proof is valid only when every step is justified, but we need an authority to confirm that every step is justified.

- In Z, *the authority, i.e., a collection of formulas, is called laws*. Laws are NOT variables, but are place-holders.

- *A variable always denotes a particular value, but a place-holder represents any expression of the appropriate type.*

- Also, a predicate may be true or false *depending on the values of its variables, but a law is always true.*

- *Predicates express facts that are particular to some specific situation, but laws express rules that are universally applicable.*

Laws

- For example, 2 * n = 6 is a predicate but not a law
  - It is only true when n = 3

- However, 0 * n = 0 is a law because it always true for all n.

- Another example, n = d * q + r is a predicate
  - It is true when n = 7, q = 2, d = 3 and r = 1, and some other cases. However it is false when n = 7, q = 2, d = 3 and r = 0

- n = d * (n div d) + (n mod d) for all d not equals 0
  - It is a law and always true

Checking specifications

- We can use formal reasoning to check our work for certain kinds of errors and oversights

- This is one of the most important qualities that distinguishes a formal method from informal ones

- For example, for the simple editor system, we can say the editor must have an initial state, i.e., $\exists State \bullet Init$ must be true.

$$\exists left, right : TEXT \mid \#(left \frown right) \leq maxsize \bullet left = right = \langle\rangle$$

Precondition calculation

- Many actual program fails because programmers did not account for all the preconditions

- For example, in the simple editor, our first attempt to define the operation that moves the cursor forward was inadequate because *it failed to account for the situation where the cursor is already at the end of the file*.

- If we had implemented that first version, we might have produced a faulty program that could crash and losing all the user's work

- We can calculate the precondition of any operation defined by a Z schema

Precondition calculation

- The precondition indicates *unprimed "before" variables and input variables only*

$$\begin{array}{|l}
\textit{Forward} \\
\hline
\Delta \textit{Editor} \\
ch? : \textit{CHAR} \\
\hline
ch? = \textit{right_arrow} \\
\textit{right} \neq \langle\rangle \\
\textit{left}' = \textit{left} \frown \langle \textit{head}(\textit{right}) \rangle \\
\textit{right}' = \textit{tail}(\textit{right}) \\
\hline
\end{array}$$

- The precondition is $ch? = \textit{right\_arrow} \land \textit{right} \neq \langle\rangle$

- *An operation is called total when the precondition covers all possibilities*. Cleary the above precondition is not total because it doesn't cover the case when right = < >

Precondition calculation

- So we must define T_Forward operation to account for that case

$$
\begin{array}{|l}
\hline
\textit{T\_Forward} \\
\Delta \textit{Editor} \\
ch? : \textit{CHAR} \\
\hline
ch? = \textit{right\_arrow} \\
((\textit{right} \neq \langle\rangle \land \textit{left}' = \textit{left} \frown \langle \textit{head right} \rangle \land \textit{right}' = \textit{tail right}) \lor \\
(\textit{right} = \langle\rangle \land \textit{left}' = \textit{left} \land \textit{right}' = \textit{right})) \\
\hline
\end{array}
$$

- The precondition is $(\textit{right} \neq \langle\rangle) \lor (\textit{right} = \langle\rangle)$

- This precondition is total

Precondition calculation

- Sometime the precondition of an operation is implicit, but we can calculate the precondition with Z schema

$$
\begin{array}{|l}
\textit{Insert} \\
\hline
\Delta \textit{Editor} \\
ch? : \textit{CHAR} \\
\hline
ch? \in \textit{printing} \\
\textit{left}' = \textit{left} \frown \langle ch? \rangle \\
\textit{right}' = \textit{right} \\
\hline
\end{array}
$$

- Does this operation always work?
- Does it have some preconditions that aren't obvious?
- Let's calculate the precondition

Precondition calculation

- We know that there must exist a state of Editor that satisfies the predicate of the insert operation schema, i.e.,

$$\exists \, Editor' \bullet Insert$$

- Because

Editor

$left, right : TEXT$

$\#(left \frown right) \leq maxsize$

- So

$$\exists \, left', right' : TEXT \mid \#(left' \frown right') \leq maxsize \bullet$$
$$ch? \in printing \land left' = left \frown \langle ch? \rangle \land right' = right$$

Precondition calculation

- It equals

$$\exists\, left', right' : TEXT \bullet$$
$$ch? \in printing \land \#(left' \frown right') \leq maxsize \land$$
$$left' = left \frown \langle ch? \rangle \land right' = right$$

- We can combine the left’ and right’ variables, then we obtain

$$\#ch? \in printing \land ((left \frown \langle ch? \rangle) \frown right) \leq maxsize$$

- The standard proof for this precondition can be as follows.

Precondition calculation

$$
\begin{array}{ll}
\exists\, Editor' \bullet Insert & [\text{Definition of precondition}] \\
\quad \Leftrightarrow \exists\, left', right' : TEXT \mid \ldots \bullet \ldots & [\text{Expand schemas}] \\
\quad \Leftrightarrow \exists\, left', right' : TEXT \bullet \ldots \land \ldots & [\text{Restricted } \exists\text{-quantifier}] \\
\quad \Leftrightarrow pr \land \#((left \frown \langle ch? \rangle) \frown right) \leq maxsize & [\text{One-point rule}] \\
\quad \Leftrightarrow pr \land \#left + \#\langle ch? \rangle + \#right \leq maxsize & [\#(s \frown t) = \#s + \#t] \\
\quad \Leftrightarrow pr \land \#left + 1 + \#right \leq maxsize & [\#\langle x \rangle = 1] \\
\quad \Leftrightarrow pr \land \#left + \#right < maxsize & [\text{Arithmetic}]
\end{array}
$$

- This example is obvious, but many software failures results from errors that seems obvious in retrospect

- In more complicated applications, the calculation sometimes reveals unexpected preconditions