Specification of state-based Systems

Prof. Martin Wirsing

25.11.2002

Specification Development with Z

Refinement of Operations

For the refinement of operations we consider the following schema:

$$
\begin{array}{|l}
\hline
\textit{OP} \\
\Delta \textit{State} \\
x? : \textit{Input} \\
y! : \textit{Output} \\
\hline
P(s, x?, s', y!) \\
\hline
\end{array}
$$

where s, s' are variables of the sort state. In sequential notation the schema OP has the form:

$$OP \quad \widehat{=} \quad [\Delta State;\ x? : Input;\ y! : Output \mid P(s, x?, s', y!)]$$

Note:

In general s and s' may have different sorts $State_1$ and $State_2$, where $State_1$ and $State_2$ are records.

If there are several input- or output variables, we write $x?$ and $y!$ for vectors.

The semantics of OP can be characterised by relations.

$$[\![OP]\!] \quad = \quad \{\, \langle\langle s, x?\rangle, \langle s', y!\rangle\rangle \in (State \times Input) \times (State \times Output) \;\mid P(s, x?, s', y!)\,\}$$

In general OP describes a relation, which accepts for every input several possible post states and output values.

Example:

Let S be the following specification

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\
\hline
x?, y! : \mathbb{N} \\
\hline
(0 < x? < 3) \wedge (y! < x? + 1) \\
\hline
\end{array}$$

In sequential notation

$$S \mathrel{\widehat{=}} [x?, y! : \mathbb{N} \mid (0 < x? < 3) \wedge (y! < x? + 1)]$$

Then

$$[\![S]\!] \quad = \quad \{\, (x?, y!) : \mathbb{N} \times \mathbb{N} \;\mid\; 0 < x? < 3 \wedge y! < x? + 1 \,\}$$

For the input $x? = 2$, the values 0, 1, 2 are possible as output $y!$. The input $x? = 7$ is not correct.

Every implementation R of S in a programming language is deterministic, i.e. every input has exactly one result (a value or undefined, if R does not terminate), and R should accept all input values of S.

R is an operational (or relational) refinement of a schema S, if

- R accepts all inputs of S (i.e. $\operatorname{dom} S \subseteq \operatorname{dom} R$) and
- if R is more determinate than S for the inputs of S (i.e. $((\operatorname{dom} S) \lhd R) \subseteq S$).

Definition:

Let $Input$, $Output$, $State$ be sets and R, $S \subseteq (State \times Input) \times (State \times Output)$.

R refines S operationally, if

1. $\operatorname{dom} S \subseteq \operatorname{dom} R$ and
2. $((\operatorname{dom} S) \lhd R) \subseteq S)$ holds.

This can also be expressed as follows:

$$T \subseteq M_1 \times M_2$$

The characteristic predicate pre T of the domain is defined:

$$\begin{aligned} (\text{pre } T)(a) \quad &=_{def} \quad a \in M_1 \land \exists\, b \in M_2 : a \; \underline{T} \; b \\ (\text{pre } T)(a) \quad &\Leftrightarrow \quad a \in \mathsf{dom}\, T \end{aligned}$$

Then R is an operational refinement of S, if:

1. pre$S \Rightarrow$ pre$R$, i.e.

$$\forall\, a, x? : (a, x?) \in \mathsf{dom} S \Rightarrow (a, x?) \in \mathsf{dom} R$$

2. pre$S \land R \Rightarrow S$, i.e.

$$\forall\, a, x?, b, y! : (a, x?) \in \mathsf{dom} S \land (a, x?) \; \underline{R} \; (b, y!) \Rightarrow (a, x?) \; \underline{S} \; (b, y!)$$

holds.

Example (Refinement of $BirthdayBook$.):

$BirthdayBook$ will be combined with two new schemata.

The enumeration type:

$$REPORT ::= ok \mid already\_known \mid not\_known$$

and the two schemata:

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{--- } Success \text{ ---}} \\
result! : REPORT \\
\hline
result! = ok \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{--- } AlreadyKnown \text{ ---}} \\
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \in known \\
result! = already_known \\
\hline
\end{array}$$

The new schema

$$RAddBirthday \mathrel{\widehat=} (AddBirthday \land Success) \lor AlreadyKnown$$

terminates for every input.

$$
\begin{array}{|l}
\hline
\textbf{RAddBirthday} \\
\Delta Birthday \\
name? : NAME \\
date? : DATE \\
result! : REPORT \\
\hline
(name? \notin known \land birthday' = birthday \cup \{name? \mapsto date?\} \land result! = ok) \quad \lor \\
(name? \in known \land birthday' = birthday \land result! = already\_known) \\
\hline
\end{array}
$$

A robust version of $FindBirthday$ and $Remind$ can be achieved by using the auxiliary schema:

$$
\begin{array}{|l}
\hline
\textit{NotKnown} \\
\hline
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \notin known \\
result! = not\_known \\
\hline
\end{array}
$$

$$
\begin{array}{rcl}
RFindBirthday & \widehat{=} & (FindBirthday \land Success) \lor NotKnown \\
RRemind & \widehat{=} & Remind \land Success
\end{array}
$$

Change of the Data Structure

- The abstract data types are replaced by concrete data types.
- After a refinement a relation $R \subseteq AS \times KS$ must exist between the abstract specification AS and the concrete specification KS.
- Compatibility conditions must exist between the initial states of AS and KS, and between the operations AOP and KOP of AS and KS.

Example:

1. Representation of 2-dimensional matrices by 1-dimensional vectors:

$$\begin{array}{|l}\hline \textit{MATRIX} \\ a : (1 \mathinner{..} r) \times (1 \mathinner{..} s) \rightarrow \mathbb{N} \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{VEKTOR} \\ c : (1 \mathinner{..} r * s) \rightarrow \mathbb{N} \\ \hline \end{array}$$

The representation relation can be defined as follows:

$$R \mathrel{\widehat{=}} [\, \textit{MATRIX};\ \textit{VEKTOR} \mid \forall\, i : 1 \mathinner{..} r;\ j : 1 \mathinner{..} s \bullet a(i,j) = c((i-1) * s + j)\,]$$

This representation is bijective.

2. Representation of bags by sequences.
Consider the bags:

$$b_1 = [\![0,1,1]\!], \quad b_2 = [\![0,1]\!], \quad b_3 = [\![1,0,1]\!]$$

Then $b_1 = b_3$, but $b_1 \neq b_2$.

In Z every bag b with elements of type X is represented by a partial function $b : X \nrightarrow \mathbb{N}$. e.g.

$$b_1 : \mathbb{N} \nrightarrow \mathbb{N} \text{ with } b_1(0) = 1, b_1(1) = 2, b_1(x) \text{ undefined for } x > 1$$

Constructors are the empty bag $[\![]\!]$, the one-element bag $[\![x]\!]$ and the union $\uplus$ of bags; $count(b, x)$ returns the number of x in b.

Bags are often represented by unordered sequences of natural numbers. Formally we define the representation relation $R \subseteq \text{bag}\,\mathbb{N} \times \text{seq}\,\mathbb{N}$ with

$$[\![b_1, \ldots, b_n]\!] \ \underline{R}\ \langle c_1, \ldots, c_k\rangle \text{ iff. } k = n \text{ and } c_1, \ldots, c_k \text{ is a permutation of } b_1, \ldots, b_n$$

Every bag $[\![b_1, \ldots, b_n]\!]$ with $n \geq 2$ has several representatives, e.g.

$[\![0, 1]\!]$	representatives	$\langle 0, 1\rangle$ and $\langle 1, 0\rangle$
$[\![0, 1, 1]\!]$	representatives	$\langle 0, 1, 1\rangle$, $\langle 1, 0, 1\rangle$ and $\langle 1, 1, 0\rangle$

Vice versa, every representative has exactly one bag.

3. Representation of bags of bit sequences with paritycheck by sequences of paritychecks

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{AS} \\
a : \mathrm{bag}\, \textsf{seq}\{0,1\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{InitAS} \\
AS \\
\hline
a = [\![\,]\!] \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{AOP} \\
\Delta AS \\
x? : \textsf{seq}\{0,1\} \\
y! : \{0,1\} \\
\hline
a' = a \uplus [\![x?]\!] \\
y! = (x?(1) + \ldots + x?(\#x?)) \ \textsf{mod}\ 2 \\
\hline
\end{array}
$$

CS

$c : \mathsf{seq}\{0, 1\}$

$InitCS$

CS

$c = \langle\rangle$

COP

ΔCS

$x? : \mathrm{seq}\{0, 1\}$

$y! : \{0, 1\}$

$c' = c \frown \langle \Sigma_2(x?) \rangle$

$y! = \Sigma_2(x?)$

The representation relation R relates every element of a bag with its paritycheck.
A sequence c with elements from $\{0,1\}$ represents a bag a, if

- the number of 1's in c equals the number of elements of a, which have a paritycheck 1 and
- the number of 0's in c equals the number of elements of a, which have a paritycheck 0.

$$
\begin{array}{|l}
\multicolumn{1}{l}{R} \\
\hline
AS \\
CS \\
\hline
\forall j \in \{0,1\} \bullet \#(c \upharpoonright \{j\}) = sum\,\{\, x : \mathsf{dom}\, a \mid \Sigma_2(x) = j \bullet x \mapsto a(x) \,\} \\
\hline
\end{array}
$$

where $sum\ b$ is the number of elements of a bag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{[X]} \\
\hline
sum : \mathrm{bag}\, X \rightarrow \mathbb{N} \\
\hline
sum\, [\![ ]\!] = 0 \\
sum(b \uplus [\![x]\!]) = sum(b) + 1 \\
\hline
\end{array}
$$

The relation R permits, that abstract elements have several concrete representations and vice versa.

$$a_1 = [\![\langle 0\rangle, \langle 0, 1\rangle]\!] \quad a_2 = [\![\langle 0, 0\rangle, \langle 0, 1\rangle]\!]$$

$$c_1 = \langle 0, 1\rangle \quad c_1 = \langle 1, 0\rangle$$

Verification Conditions for Refinements

Reminiscence

C is an operational refinement of A.

1.	C applicable, if A applicable	$\mathsf{dom}\,A \subseteq \mathsf{dom}\,C$	$\mathsf{pre}\,A \Rightarrow \mathsf{pre}\,C$
2.	C is more determinate than A	$(\mathsf{dom}\,A \lhd C) \subseteq A$	$(\mathsf{pre}\,A) \wedge C \Rightarrow A$

Now

Extension of this condition for the change of the data structure.

The relation between state descriptions in abstract specifications and concrete implementations is given by a representation relation.

Idea

The implementation of an operation . . .

1. is applicable in every representative of a state, where the abstract operation is applicable, and
2. leads to a representative of a possible result state of the abstract operation.

Specifications often describe components of large systems. Component specifications serve as

- starting point for the development of the component and
- description of the component interfaces.

During the component development . . .

- it is allowed to reduce the Non-determinism of operations:
  the other components have to accept every result, permitted by the original specification.
- it is not allowed to restrict the pre-domain of operations:
  for every permitted input, it is expected to get an output in return, which fulfils the specification.

It is permitted to extend the pre-domain of operations during the refinement, but it can not be used by other components.

Example (Mean Value):

$$\begin{array}{|l}\hline \textit{AbsCalculator} \\ \textit{store} : \mathrm{bag}\,\mathbb{Z} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Calculator} \\ \textit{sum} : \mathbb{Z} \\ \textit{cnt} : \mathbb{N} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{AbsInitCalculator} \\ \textit{AbsCalculator} \\ \hline \textit{store} = \emptyset \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{InitCalculator} \\ \textit{Calculator} \\ \hline \textit{sum} = \textit{cnt} = 0 \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{AbsEnter} \\ \Delta \textit{AbsCalculator} \\ n? : \mathbb{Z} \\ \hline \textit{store}' = \textit{store} \uplus [\![n?]\!] \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Enter} \\ \Delta \textit{Calculator} \\ n? : \mathbb{Z} \\ \hline \textit{sum}' = \textit{sum} + n? \\ \textit{cnt}' = \textit{cnt} + 1 \\ \hline\end{array}$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{AbsMean} \\
\Delta AbsCalculator \\
mean! : \mathbb{Z} \\
\hline
store \neq \emptyset \\
store' = store \\
mean! = (\sum store) \ \mathsf{div}\ (card\ store) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Mean} \\
\Delta Calculator \\
mean! : \mathbb{Z} \\
\hline
cnt \neq 0 \\
sum' = sum \\
cnt' = cnt \\
mean! = sum \ \mathsf{div}\ cnt \\
\hline
\end{array}
$$

The relation between $Calculator$ and $AbsCalculator$ is described by the representation relation.

$$
\begin{array}{|l}
\hline
\textit{RepCalculator} \\
AbsCalculator \\
Calculator \\
\hline
sum = \sum store \\
cnt = card\ store \\
\hline
\end{array}
$$

For example the abstract states:

$\langle\!| store = [\![2, 2, 4, 6]\!] |\!\rangle$ and $\langle\!| store = [\![1, 3, 5, 5]\!] |\!\rangle$

are described by the implementation state

$\langle\!| sum = 14, cnt = 4 |\!\rangle$

Example (ID Assignment):

$$\begin{array}{l} \textbf{AbsID} \\ \hline used : \mathbb{P}\mathbb{N} \end{array}$$

$$\begin{array}{l} \textbf{ID} \\ \hline nextID : \mathbb{N} \end{array}$$

$$\begin{array}{l} \textbf{AbsInitID} \\ \hline AbsID \\ \hline used = \emptyset \end{array}$$

$$\begin{array}{l} \textbf{InitID} \\ \hline ID \\ \hline nextID = 0 \end{array}$$

$$\begin{array}{l} \textbf{AbsAssignID} \\ \hline \Delta AbsID \\ newID! : \mathbb{N} \\ \hline newID! \notin used \\ used' = used \cup \{newID!\} \end{array}$$

$$\begin{array}{l} \textbf{AssignID} \\ \hline \Delta ID \\ newID! : \mathbb{N} \\ \hline newID! = nextID \\ nextID' = nextID + 1 \end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{AbsReclaimID} \\ \hline
\Delta AbsID \\
freeID? : \mathbb{N} \\ \hline
used' = used \setminus \{freeID?\} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{ReclaimID} \\ \hline
\Xi ID \\
freeID? : \mathbb{N} \\ \hline
\end{array}$$

- Determinate implementations: counter instead of sets
- The specification does not exclude the behavior of the implementation.

General case: Given two schema tuples

$$Abs = \langle AbsState, AbsInit, AbsOps \rangle \quad \text{and} \quad Conc = \langle ConcState, ConcInit, ConcOps \rangle$$

and a representation relation

$$Rep \mathrel{\widehat{=}} [\, AbsState;\; ConcState \mid RepInv \,]$$

$Conc$ refines Abs w.r.t. Rep, if holds:

1. Every possible initial state of $Conc$ represents a possible initial state of Abs.
2. For every operation COp of $Conc$, there is one operation AOp of Abs, so that
   (a) If AOp is applicable in state a and if c is a possible representative of a, so COp is applicable in c.
   (b) If c' is a possible result state of COp and if c is representative of a state a, so c' is a representative of a possible result state a' of AOp.

   holds.

Definition:

A schema tuple $Conc = \langle CState, CInit, COps \rangle$ refines a schema tuple $Abs = \langle AState, AInit, AOps \rangle$ w.r.t. a representation relation Rep, if:

[Initialising] $CInit \Rightarrow \exists\, AState \bullet AInit \wedge Rep$

[Operations] For every operation $COp \in COps$ exists an operation $AOp \in AOps$ with:

[Applicability] $Rep \wedge (\mathrm{pre}\ AOp) \Rightarrow \mathrm{pre}\ COp$
[Correctness] $Rep \wedge (\mathrm{pre}\ AOp) \wedge COp \Rightarrow \exists\, AState' \bullet AOp \wedge Rep'$

holds.

Note:
Operation refinement is a special case of this definition with $CState = AState$, $CInit = AInit$ and the identity as representation relation.

Refinement condition for ID Assignment

Representation relation

$$\begin{array}{|l}
\multicolumn{1}{l}{IDRep} \\
\hline
AbsID \\
ID \\
\hline
used \subseteq \mathsf{ran}(0 \mathinner{\ldotp\ldotp} nextID - 1) \\
\hline
\end{array}$$

Initialising $\quad ID \land nextID = 0 \quad \Rightarrow \quad \exists\, used \bullet used \subseteq \mathsf{ran}(0 \mathinner{\ldotp\ldotp} -1)$

Assign

(a) $IDRep \land (\mathsf{pre}\, AbsAssignID) \quad \Rightarrow \quad \mathsf{pre}\, AssignID$

(b) $IDRep \land (\mathsf{pre}\, AbsAssignID) \land AssignID$

$$\Rightarrow \exists\, AbsID' \bullet AbsAssignID \land IDRep'$$

$$\Leftrightarrow$$

$$used \subseteq \mathsf{ran}(0 \mathinner{\ldotp\ldotp} nextID - 1) \land newID! = nextID \land nextID' = nextID + 1$$

$$\Rightarrow \exists\, used' \bullet used' = used \cup \{newID!\} \land used' \subseteq \mathsf{ran}(0 \mathinner{\ldotp\ldotp} nextID' - 1)$$

Reclaim

(a) $IDRep \land (\mathsf{pre}\, AbsReclaimID) \quad \Rightarrow \quad \mathsf{pre}\, ReclaimID$

(b) $IDRep \land (\mathsf{pre}\, AbsReclaimID) \land ReclaimID$

$$\Rightarrow \exists\, AbsID' \bullet AbsReclaimID \land IDRep'$$

$$\Leftrightarrow$$

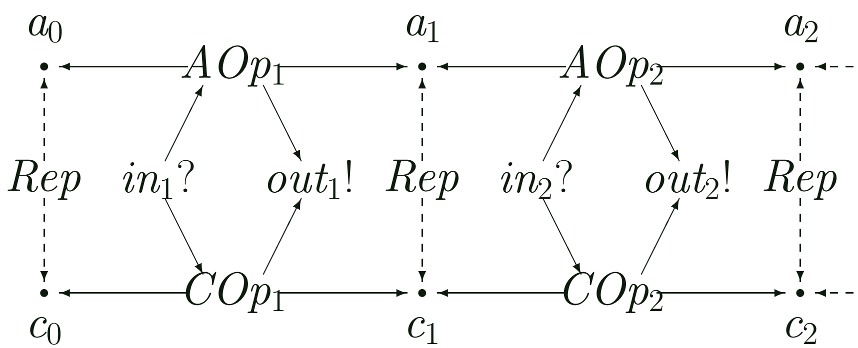
$$used \subseteq \mathsf{ran}(0 \mathinner{\ldotp\ldotp} nextID - 1) \land nextID' = nextID$$

$$\Rightarrow \exists\, used' \bullet used' = used \setminus \{freeID?\} \land used' \subseteq \mathsf{ran}(0 \mathinner{\ldotp\ldotp} nextID' - 1)$$

A state sequence $s_0, s_1, s_2, \ldots$ is called process of a Z-specification $\langle State, Init, Ops \rangle$, if s_0 satisfies the initialising condition $Init$ and if for all $i \geq 0$ there is an operation $Op \in Ops$, so that the state pair $\langle s_i, s_{i+1} \rangle$ is a model of Op.

Let $Conc$ be a refinement of Abs and let $c_0, c_1, \ldots$ be a process of $Conc$, where only concrete operations COp have been applied.

Then a process $a_0, a_1, \ldots$ of Abs exists, so that a_i is a representative of c_i (for all i). If the operation AOp is abblicable in a_k and if COp is a refinement of AOp, so COp is applicable in c_k.



Transfer to Imperative Programming Languages

Goals

- Understand the relation between Z-schemata and imperative programs
- Formal development of programs by description of states and operations
- Basics of the refinement calculus

Literature

R. Back, J. von Wright: Refinement calculus—A Systematic Introduction. Springer-Verlag, 1998.

C. Morgan: Programming from Specifications. Prentice-Hall, 3. Auflage 1998.

H. Partsch: Specification and Transformation of Programs. Springer-Verlag, 1990.

Programs as relations over states.

A Z-schema of the form

$$\begin{array}{|l}\hline Op \\ \Delta State \\ \hline P \\ \hline\end{array}$$

or $Op \mathrel{\widehat{=}} [\,\Delta State \mid P\,]$

describes a relation between states. A program can also be interpreted as a relation between two states.

Example:

The following programs transform a state with $x = 0 \wedge y = 1$ into a state with $x = 1 \wedge y = 1$.

- `x := 1`
- `x := y`
- `x := x+1`
- `while x<1 do x := x+1 end`

Example-programming language

according to guarded commands

(Multi-)Assignment	`x,y := x+y, y-x`
Sequential execution	P ; Q

Conditional statement (non-determinate)	`if`
	`[]` b_1 `->` P_1
	$\vdots$
	`[]` b_n `->` P_n
	`else` P_{n+1} (optional)
	`fi`

Loop	`do` b `->` P `od`
Local block	`|[ x:T ;` Π `]|`

During the development program skeletons are formed, which include Z-schemata for program fragments, which have to be developed.

General framework

A schema tuple $\langle State, Init, Ops \rangle$ is transformed into a program of the form

```
|[ StateDecls ;
   Init ;
   type Choice = Quit | Op1 | ...  | OpN ;
   |[ choice :  Choice ; MakeChoice ;
     do  choice ≠ Quit  ->
         if  choice = Op1  ->
             |[ InOutDecls1 ; GetInputs1 ; Op1 ; SendOutputs1  ]|
         ...
         []  choice = OpN  ->
             |[ InOutDeclsN ; GetInputsN ; Opn ; SendOutputsN  ]|
         fi ;
         MakeChoice
     od
   ]|
 ]|
```

Simplifying assumptions:

- All operations of the starting specification are total.
- Predicates of all schemata include only variables, which occur in the declaration part (no global variables!).
- The types of the declared variables are available in the programming language.

In the following we write

$$\begin{array}{|l}
\multicolumn{1}{l}{Op} \\ \hline
\Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U}] \\ \hline
P(\vec{x}, \vec{y}) \wedge Q(\vec{y}) \\ \hline
\end{array}$$

Implementations of Op are only allowed to include assignments for the variables $\vec{x}$.

Basic principles of the refinement rules:

- Z-schemata are only included at those parts of the program, where their pre-condition is satisfied.
- All variables listed in the declaration part of a schema are declared at this part of the program.
- Only explicitly listed variables in $\Delta[\vec{x} : \vec{T}]$ are allowed to be changed (frame rule).

The applicability principle holds in the starting program, because all operations are expected to be total.

The frame rule refers to the actually declared variables. Local variables may be changed, because they are not visible outside the block.

Implementation

Let Φ, Ψ be (mixed) programs.

$$\Phi \sqsubseteq \Psi \qquad (\Psi \text{ implements } \Phi)$$

holds, iff

1. Ψ is applicable, always when Φ is applicable and
2. every possible state transition according to Ψ is also permitted by Φ, provided that Φ is applicable in the start state.

If we see Φ and Ψ as relations over states,

$$\Phi \sqsubseteq \Psi \qquad \Leftrightarrow \qquad \mathrm{dom}\,\Phi \subseteq \mathrm{dom}\,\Psi \quad \wedge \quad (\mathrm{dom}\,\Phi \lhd \Psi) \subseteq \Phi$$

holds again.

If Φ and Ψ are schemata and if Ψ is an operational refinement of Φ, $\Phi \sqsubseteq \Psi$ holds. This permits predicate logic tranformations during application development. The following rules serve for stepwise transformation of operation schemata into programs.

Introduction of local variables

$$
\begin{array}{|l}
\hline
\quad Op \\
\Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U}] \\
\hline
P \\
\hline
\end{array}
\quad \sqsubseteq \quad |[\ \vec{z} : \vec{V}\ ;\
\begin{array}{|l}
\hline
\quad Op1 \\
\Delta[\vec{x} : \vec{T};\ \vec{z} : \vec{V}];\ \Xi[\vec{y} : \vec{U}] \\
\hline
P \\
\hline
\end{array}
\ ]|
$$

for the new variables $\vec{z}$

The new variables can be changed in later refinement steps.

However it is not allowed to make assumptions concerning the starting value, because that would restrict the applicability of $Op1$

Omitting unused variables

$$\begin{array}{|l}\hline \multicolumn{1}{l}{Op} \\ \Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U};\ \vec{z} : \vec{V}] \\ \hline P(\vec{x}, \vec{y}) \land Q(\vec{y}, \vec{z}) \\ \hline \end{array} \quad \sqsubseteq \quad \begin{array}{|l}\hline \multicolumn{1}{l}{Op1} \\ \Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U}] \\ \hline P(\vec{x}, \vec{y}) \\ \hline \end{array}$$

Reason

- The frame rule guarantees, that the variables $\vec{z}$ will not be changed.
- The applicability condition guarantees, that $Q(\vec{y}, \vec{z})$ holds before and after the execution of Op and $Op1$.

Example:

$$\begin{array}{|l}\hline \multicolumn{1}{l}{SwapIfGreater} \\ \Delta[x, y : \mathbb{Z}];\ \Xi[z : \mathbb{N}] \\ \hline z > 5 \land x' = y \land y' = x \\ \hline \end{array} \quad \sqsubseteq \quad \begin{array}{|l}\hline \multicolumn{1}{l}{Swap} \\ \Delta[x, y : \mathbb{Z}] \\ \hline x' = y \land y' = x \\ \hline \end{array}$$

Assignment rule for simple variables

$$\begin{array}{|l} \hline Op \\ \Delta[x_1 : T_1;\ \ldots;\ x_n : T_n];\ \Xi[\vec{y}] \\ \hline x_1' = e_1 \wedge \ldots \wedge x_n' = e_n \\ P \\ \hline \end{array} \quad \sqsubseteq \quad \texttt{x1,...,xn := e1,...,en}$$

Assumption:

- The variables $x_1, \ldots, x_n$ are pairwise different.
- The expressions $e_1, \ldots, e_n$ include no slashed variables and comply directly with the expressions `e1,...,en` of the programming language.

Correctness of the assignment rule

For the pre-condition of Op holds

$$\text{pre } Op \equiv P[e_1/x_1', \ldots, e_n/x_n', \vec{y}/\vec{y}']$$

and the applicability condition guarantees, that this predicate is satisfied before execution of the assigment.

Example:

$$\begin{array}{|l} \hline \multicolumn{1}{l}{SwapIfGreater} \\ \Delta[x, y : \mathbb{Z}];\ \Xi[z : \mathbb{N}] \\ \hline z > 5 \wedge x' = y \wedge y' = x \\ \hline \end{array} \quad \sqsubseteq \quad \texttt{x,y := y,x}$$

Generalisation: Array components

Arrays can be modeled in Z by sequences, assigments are equal to the overwriting of an array at a certain index position.

$$\begin{array}{|l} \textit{Op} \\ \hline \Delta[x_1 : T_1; \ \ldots; \ x_n : T_n]; \ \Xi[\vec{y}] \\ \hline \ldots \wedge x_i' = x_i \oplus \{j \mapsto e\} \wedge \ldots \\ P \\ \hline \end{array} \qquad \sqsubseteq \quad \texttt{...,xi[j],...:= ...,e,...}$$

if holds: $P \Rightarrow 1 \leq j \leq \# x_i$

Example:

$$\begin{array}{|l} \textit{SwapIJ} \\ \hline \Delta[x : \operatorname{seq} \mathbb{Z}]; \ \Xi[i, j : \mathbb{N}] \\ \hline x' = x \oplus \{i \mapsto x(j), j \mapsto x(i)\} \\ 1 \leq i < j \leq \# x \\ \hline \end{array} \qquad \sqsubseteq \quad \texttt{x[i],x[j] := x[j],x[i]}$$

Similarly: Assignment to record components

Sequentialization

$$\frac{\mathsf{pre}\,Op \Rightarrow \mathsf{pre}\,Op_1 \qquad \mathsf{pre}\,Op \wedge Op_1 \Rightarrow (\mathsf{pre}\,Op_2)' \qquad \mathsf{pre}\,Op \wedge (Op_1 \fatsemi Op_2) \Rightarrow Op}{Op \quad \sqsubseteq \quad Op_1 \;\; ; \;\; Op_2}$$

Reason

- Is Op applicable in a state s, so Op_1 is also applicable.
- Every execution of Op_1 starting from a state s, ends in a state, where Op_2 is applicable.
- If Op_1 and Op_2 are executed sequentially in a state s , so the result satisfies the specification Op.

Example (Search for the index position of the maximum in an array):

$$\begin{array}{|l}
\textit{FindMaxPos} \\
\hline
\Delta[\mathit{maxel} : \mathbb{N}];\ \Xi[a : \mathsf{seq}\,\mathbb{Z}] \\
\hline
\# a > 0 \\
1 \leq \mathit{maxel}' \leq \# a \\
\forall\, n : 1 \mathinner{\ldotp\ldotp} \# a \bullet a(\mathit{maxel}') \geq a(n) \\
\hline
\end{array}$$

Step 1: Introduction of a local variable for the iteration

$$\textit{FindMaxPos} \quad \sqsubseteq \quad \texttt{|[seen:int;} \begin{array}{|l}
\textit{FindMaxSeen} \\
\hline
\Delta[\mathit{maxel} : \mathbb{N};\ \mathit{seen} : \mathbb{Z}];\ \Xi[a : \mathsf{seq}\,\mathbb{Z}] \\
\hline
\# a > 0 \\
1 \leq \mathit{maxel}' \leq \# a \\
\forall\, n : 1 \mathinner{\ldotp\ldotp} \# a \bullet a(\mathit{maxel}') \geq a(n) \\
\hline
\end{array} \texttt{]|}$$

Step 2: Initialising and further computation of $FindMaxSeen$

$$FindMaxSeen \sqsubseteq \begin{array}{|l} \multicolumn{1}{l}{\textit{Initialise}} \\ \hline \Delta[maxel : \mathbb{N};\ seen : \mathbb{Z}] \\ \Xi[a : \mathsf{seq}\mathbb{Z}] \\ \hline MaxelInv' \\ \hline \end{array} \ ; \ \begin{array}{|l} \multicolumn{1}{l}{\textit{Complete}} \\ \hline \Delta[maxel : \mathbb{N};\ seen : \mathbb{Z}] \\ \Xi[a : \mathsf{seq}\mathbb{Z}] \\ \hline seen' = \#a \\ \Delta MaxelInv \\ \hline \end{array}$$

with the auxiliary schema

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{MaxelInv}} \\ \hline maxel : \mathbb{N};\ seen : \mathbb{Z};\ a : \mathrm{seq}\,\mathbb{Z} \\ \hline \#a > 0 \\ 1 \leq maxel \leq seen \leq \#a \\ \forall\, n : 1 \mathinner{\ldotp\ldotp} seen \bullet a(maxel) \geq a(n) \\ \hline \end{array}$$

Proof 1 (Search for the index position of the maximum in an array):

1. pre$FindMaxSeen \Rightarrow$ pre$Initialise$

   reduces to $\quad \#a > 0 \Rightarrow \#a > 0$

2. pre$FindMaxSeen \wedge Initialise \Rightarrow (\text{pre}\,Complete)'$

   holds, because $Initialise$ implies the condition $MaxelInv'$

3. pre$FindMaxSeen \wedge (Initialise \,;\, Complete) \Rightarrow FindMaxSeen$

   $Complete$ guarantees $\quad seen' = \#a \wedge MaxelInv'$

Step 3: Implementation of $Initialise$

$$Initialise \quad \sqsubseteq \quad \begin{array}{|l} \text{\underline{InitRefined}} \\ \Delta[maxel : \mathbb{N};\ seen : \mathbb{Z}];\ \Xi[a : \text{seq}\,\mathbb{Z}] \\ \hline MaxelInf' \\ seen' = maxel' = 1 \\ \hline \end{array} \qquad \text{(operational refinement)}$$

$\sqsubseteq$ `seen,maxel := 1,1` (Assignment rule)

Case analysis

Let Op be a schema, $b_1, \ldots, b_n$ conditions, where only unslashed variables from Op occur and which comply directly with the conditions `b1, . . . , bn` in the programming language.

If $\text{pre } Op \Rightarrow b_1 \vee \ldots \vee b_n,$ holds, so

$Op \quad \sqsubseteq$
```
if b1 -> b₁ ∧ Op
[] b2 -> b₂ ∧ Op
...
[] bn -> bₙ ∧ Op
fi
```

Idea: Choose b_i so that the single alternatives can be simplified in the following steps.

Correctness

- The applicability condition for the schemata $b_i \wedge Op$ hold because of the assumption

$$\text{pre}\, Op \Rightarrow b_1 \vee \ldots \vee b_n$$

- Correctness, because of propositional logical simplifications.

Example (Iteration step for the maximum search):

$$
\begin{array}{|l}
\hline
\textit{StepSeen} \\
\Delta[maxel : \mathbb{N};\; seen : \mathbb{Z}];\; \Xi[a : \mathsf{seq}\mathbb{Z}] \\
\hline
\Delta \textit{MaxelInv} \\
seen' = seen + 1 \\
\hline
\end{array}
$$

$$
\sqsubseteq \quad
\begin{array}{|l}
\hline
\textit{MoveSeen} \\
\Delta[seen : \mathbb{Z}];\; \Xi[a : \mathsf{seq}\mathbb{Z};\; maxel : \mathbb{N}] \\
\hline
\textit{MaxelInv} \\
1 \leq seen < \#a \\
seen' = seen + 1 \\
\hline
\end{array}
\quad ; \quad
\begin{array}{|l}
\hline
\textit{AdjustMaxel} \\
\Delta[maxel : \mathbb{N}];\; \Xi[a : \mathsf{seq}\mathbb{Z};\; seen : \mathbb{Z}] \\
\hline
1 \leq maxel < seen \leq \#a \\
\forall\, n : 1 \,..\, (seen - 1) \bullet a(maxel) \geq a(n) \\
\textit{MaxelInv}' \\
\hline
\end{array}
$$

$\sqsubseteq$
```
seen := seen+1 ;
if a[seen] > a[maxel]    ->
[] a[seen] <= a[maxel]   ->
fi
```
$a(seen) > a(maxel) \wedge AdjustMaxel$

$a(seen) \leq a(maxel) \wedge AdjustMaxel$

$\sqsubseteq \ldots \sqsubseteq$
```
seen := seen+1 ;
if a[seen] > a[maxel]    ->  maxel := seen
[] a[seen] <= a[maxel]   ->  maxel := maxel
fi
```

Iteration rule

Implementations by a loop $Op \sqsubseteq$ `do b ->` $Body$ `od`

$Idea$ Invariant Inv and variant v

- The invariant holds at the beginning and at the end of every execution of $Body$
- The invariant and the condition guarantee applicability in $Body$
- The invariant and negation of the condition guarantee correctness
- The variant decreases at every execution of $Body$ and ensures termination.

Formally: Let be

Inv, b, $Goal$	Predicates without slashed variables
`b`	Direct translation of b into the programming language
v	Numerical arithmetical expression without slashed variables
$Iterate$	Operation schema for the iteration step
$\vec{x}$, $\vec{y}$	Tuples of all occurring variables

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Op} \\
\Delta[\vec{x}];\ \Xi[\vec{y}] \\
\hline
\Delta Inv \\
Goal' \\
\hline
\end{array}
\quad \sqsubseteq \quad \texttt{do b ->} \quad
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Body} \\
\Delta[\vec{x}];\ \Xi[\vec{y}] \\
\hline
\Delta Inv \\
Iterate \\
\hline
\end{array}
\quad \texttt{od}
$$

holds, if all following conditions are satisfied:

$$Inv \wedge b \Rightarrow \text{pre } Body$$
$$Inv \wedge \neg\, b \Rightarrow Goal$$
$$b \wedge Body \Rightarrow 0 \leq v' < v$$

Example (Implementation of $Complete$ by a loop):

$Complete \sqsubseteq$ `do seen < #a -> ` $StepSeen$ ` od`

Instantiation of the iteration rule

Inv $Goal$ v	$MaxelInv$ $seen = \#a$ $\#a - seen$	b $Iterate$	$seen < \#a$ $seen' = seen + 1$

Generated code

```
|[ seen :  int ;
   seen, maxel := 1,1 ;
   do  seen < #a  ->
            seen := seen+1 ;
            if a(seen) > a(maxel)  ->  maxel := seen
            []  a(seen) <= a(maxel)  ->  maxel := maxel
            fi
   od  ]|
```

Summary

- The effect of operations of state-based systems is not only determined by the input variables, but also depend on the current system state.
- Z-specifications make model-oriented descriptions of state-based and interactive systems possible.
- A typical specification has the form $\langle State, Init, Ops \rangle$.

  Where the schema $State$ defines the components for the description of a state and defines the relation between state components by schemata invariants.

  The schema $Init$ describes the subset of possible initial states.

  Every operation is described by a schema $Op \in Ops$ with the help of a predicate over pre- and post states and input/output values.
- The basic data structures are described by predefined basic structures. Z has no recursive function definitions.
- Z-schemata can be renamed by decorations and can be combined by logical operators with quantifiers.
- The specification development in Z is based on the refinement concept.

- For an operation refinement AOp by COp must hold:
  Applicability condition pre$AOp \Rightarrow$ preCOp and
  Correctness condition (pre$AOp) \wedge COp \Rightarrow AOp$
- The concept of data refinement generalizes this concept for algebraic specifications, by expressing the relation between an abstract specification and the state of an implementation by a relation.
- The development of imperative programs from Z-specifications of operations is formally provided by the refinement calculus.