Another Z Specification Example

- Write a specification for an Assembler
- Describe: translating a single address assembly program syntactically to machine language
- Translation process:
  - symbolic operand is replaced by memory address
  - numeric operands are unchanged
  - symbolic ops are replaced by number opcodes
  - get rid of symbolic labels
Example of Translation

**Assembly program**

<table>
<thead>
<tr>
<th>label</th>
<th>op</th>
<th>operand (ref/num)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>v1:</td>
<td>const 100</td>
</tr>
<tr>
<td>2</td>
<td>v2:</td>
<td>const 250</td>
</tr>
<tr>
<td>3</td>
<td>loop:</td>
<td>load v2</td>
</tr>
<tr>
<td>4</td>
<td>subc</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>6</td>
<td>comp</td>
<td>v1</td>
</tr>
<tr>
<td>7</td>
<td>jumple</td>
<td>exit</td>
</tr>
<tr>
<td>8</td>
<td>jump</td>
<td>loop</td>
</tr>
<tr>
<td>9</td>
<td>exit:</td>
<td>return</td>
</tr>
</tbody>
</table>

**Machine program**

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>62</td>
</tr>
<tr>
<td>8</td>
<td>61</td>
</tr>
<tr>
<td>9</td>
<td>70</td>
</tr>
</tbody>
</table>
The “World” of Assembler Example

• Basic sets:
  A – set of assembly instructions
  M – set of machine instructions
  SYM – set of possible labels
  OPSYM – set of possible symbolic assembly ops

• A function given: (translating from assembly operation symbols to machine opcodes)
  cor: OPSYM $\rightarrow$ N
In the Example:

<table>
<thead>
<tr>
<th>Assembly program</th>
<th>Machine program</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>opcode</td>
</tr>
<tr>
<td>1 v1:</td>
<td>const</td>
</tr>
<tr>
<td>2 v2:</td>
<td>const</td>
</tr>
<tr>
<td>3 loop:</td>
<td>load</td>
</tr>
<tr>
<td>4</td>
<td>subc</td>
</tr>
<tr>
<td>5</td>
<td>store</td>
</tr>
<tr>
<td>6</td>
<td>comp</td>
</tr>
<tr>
<td>7</td>
<td>jumple</td>
</tr>
<tr>
<td>8</td>
<td>jump</td>
</tr>
<tr>
<td>9 exit:</td>
<td>return</td>
</tr>
</tbody>
</table>

= in A

= in M
Assembler Components

**ASSEMBLY**

- $label: A \mapsto SYM$
- $op: A \mapsto OPSYM$
- $ref: A \mapsto SYM$
- $num: A \mapsto N$

$\text{dom } ref \cap \text{dom } num = \{ \}$
$\text{dom } op \cup \text{dom } num \cup \text{dom } ref = A$

**MACHINE**

- $\text{opcode: } M \mapsto N$
- $\text{operand: } M \mapsto N$

$\text{dom } \text{opcode } \cup \text{dom } \text{operand } = M$
Putting Together an Assembler

\begin{itemize}
\item \textbf{ASSEMBLER} \hspace{1cm}
\item \textbf{ASSEMBLY}
\item \textbf{MACHINE}
\item in?: seq A
\item out!: seq M
\item \( symtab \in SYM \mapsto N \)
\item \( \text{ran}(\text{in?} ; \text{ref}) \subseteq \text{dom} symtab \)
\item \( \text{ran}(\text{in?} ; \text{op}) \subseteq \text{dom} \text{cor} \)
\item \( (\text{out!} ; \text{operand}) = (\text{in?} ; \text{ref} ; symtab) \cup (\text{in?} ; \text{num}) \)
\item \( (\text{out!} ; \text{opcode}) = (\text{in?} ; \text{op} ; \text{cor}) \)
\end{itemize}
Refinements in Z
Implementation/Refinement

• **Qn:** Given two system descriptions, what does it mean to say the first is implemented by the second?
• **Ans:** The second (concrete) system should do something appropriate whenever the first (abstract) system does something.
• **Note:** It can do more for situations irrelevant to the abstract system, and if the abstract system leaves some options open, then it can choose one of those options.
Methods of Refinement in Z

- Basic problem: How to get from a specification to a program?
- No single method
- Usually development is divided into two parts
  
  **Part 1:**
  
  repeat two step process of
  
  – data refinement
  
  – operation refinement
  
  until "sufficiently close" to a programming language

  **Part 2:**
  
  "associate" code with the specification
Relating Two Systems

In general, two system descriptions may have different state spaces:

Our first task is therefore to produce an abstraction relation from the "concrete" system to the "abstract" system (Abs). [Example: array implementing a set]
Then for each state transition, $T$, establish the correspondence.

$T\text{-}abs \ (\text{Abs}(\text{state}\text{-}con)) = \text{Abs}(T\text{-}con(\text{state}\text{-}con))$
Observations

- Abs usually associates many concrete states with each abstract state.
  (eg., a sequence representing a set)
- Abs is usually a function: each concrete state has at most one abstract state.
  (If not, what does this say about the abstract state model?)
- Must also account for initial states.
- Concrete operations
  - must be defined for all the corresponding states for which the abstract operation is defined (applicability)
  - must produce appropriate values (correctness)
· In Z we first describe the global domain of discourse as a set of states (defined by a schema) – the state space.
· Then we define the operations in terms of the transition relationships between subsets of these states.
· For each operation:
  – The pre-condition is a predicate that determines the legal starting states.
  – The post-condition is a predicate that determines the resulting states.
Data Refinement in Z

BirthdayBook

known: P NAME

birthday: NAME → DATE

known = dom birthday

BirthdayBook1

names: seq[ NAME ]
dates: seq[ DATE ]
size: N

" i,j: 1 .. size · i ≠ j ⇒ names(i) ≠ names(j)"
Abstraction Function

Abs
BirthdayBook
BirthdayBook1

known = {i: 1.. size ∙ names(i)}

" i: 1 .. size ∙ birthday(names(i)) = dates(i)"
The Initial States Theorem

· Must show that every initial concrete state corresponds to an initial abstract state.

· This is the theorem:

\[ \text{InitConcrete} \land \text{Abs} \implies \text{InitAbstract} \]

· Example:

InitBirthBook

BirthdayBook

known = \emptyset

birthday = \emptyset

InitBirthBook1

BirthdayBook1

size = 0

names = dates = <>
Alternative: init actions

• Instead of initial states, use initialize actions, and relate to the state after those actions.
• Then want:

  init-conc-act \ abs’ \init-abs-act

The init actions only have primed states (since there is no “before” state)
To prove

InitConcrete ∧ Abs |- InitAbstract

we would argue:

Known

\[ \text{Known} = \{ i: 1..\text{size} \cdot \text{names}(i) \} \quad \text{[Abs]} \]

\[ = \{ i: 1..0 \cdot \text{names}(i) \} \quad \text{[size = 0]} \]

\[ = \emptyset \]

and similarly we could show

birthday = \emptyset
Operation Refinement

· When an abstract operation $\text{Op1}$ is refined by concrete operation $\text{Op2}$, we write: $\text{Op1} \sqsubseteq \text{Op2}$

· To show $\text{Op1}$ is refined by $\text{Op2}$:

  1. **Applicability**
     
     Whenever $\text{Op1}$ is applicable so is $\text{Op2}$.

  2. **Correctness**
     
     When $\text{Op1}$ is applicable but $\text{Op2}$ is applied, then the result is consistent with $\text{Op1}$.
Relations

- For the case of relations $R$ and $S$, $R \sqsubseteq S$ means:
  1. $\text{dom } R \subseteq \text{dom } S$ [applicability]
  2. $((\text{dom } R) \smallsetminus S) \subseteq R$ [correctness]

- Example

  $R = \{(\text{Todd, blue}), (\text{Todd, red}), (\text{Jim, green}), (\text{Jim, red})\}$
  
  $S = \{(\text{Todd, blue}), (\text{Jim, green}), (\text{Hillary, red})\}$

  $R \sqsubseteq S$ because
  1. $\{\text{Todd, Jim}\} \subseteq \{\text{Todd, Jim, Hillary}\}$
  2. $\{(\text{Todd, blue}), (\text{Jim, green})\} \subseteq R$
Schemas

- For the case of schemas, $R$ and $S$, that represent operations on the same state, $R \sqsubseteq S$ (“$R$ is refined by $S$”) means:

1. $\text{pre } R \Rightarrow \text{pre } S$ [applicability]
2. $(\text{pre } R \land S) \Rightarrow R$ [correctness]
General Case

In the general case, the operations apply to different states (concrete and abstract).

- In this case \( \text{Op-abs} \subseteq \text{Op-conc} \) means:

1. \((\text{pre Op-abs} \land \text{Abs}) \Rightarrow \text{pre Op-conc} \) [applicability]
   
   *whenever Op-abs can be legally applied to an abstract state, Op-conc can be legally applied to any corresponding concrete state.*

2. \((\text{pre Op-abs} \land \text{Abs} \land \text{Op-conc} \land \text{Abs'}) \Rightarrow \text{Op-abs} \) [correctness]
   
   *Op-conc is consistent with the post-condition of Op-abs*
Example

AddBirthday

Δ BirthdayBook

name?: NAME

date?: DATE

name? \notin \text{known}

\text{birthday}' = \text{birthday} \cup \{(\text{name?}, \text{date?})\}
Combinations

AddBirthday1

Δ BirthdayBook1

name?: NAME

date?: DATE

name? ∉ ran names

size + 1 = size'

names' = names + <name?> (concatenation)

dates' = dates ∪ {(size’, date?)}
Applicability (informal)

· Must show that whenever AddBirthday can be legally applied to an abstract state, AddBirthday1 can be legally applied to any corresponding concrete state.

· The precondition of AddBirthday1 is

  " i: 1 .. size · name? ≠ names(i)

· Abs tells us that this precondition is equivalent to saying

  name? ∉ known

· But this is precisely the precondition of AddBirthday

Pre AddBirthday ∧ Abs => PreAddBirthday1
Correctness (informal)

- We must show that the concrete operation is consistent with the postcondition of AddBirthday:

  \[ \text{birthday}' = \text{birthday} \cup \{ (\text{name}?, \text{date}?) \} \]

- The proof:
  1. We show that the function represented by the two arrays after the operation has the same domain as \textit{birthday}' (as viewed through \textit{Abs}).
  2. Then we show that the values are the same.

The formal proof just substitutes in the definition of refinement, and simplifies to show the implication.
Schema refinement?

- **Lookup versus Rlookup: what refines what?**

  ─── Lookup ─────────────
  
  st, st ' : ST
  s? : STR ; v! : VAL

  ─────────────
  
  s? ∈ dom st
  v! = st s?
  st' = st

  ─────────────

  ─── BadLookup ──────────
  
  st : ST
  s? : STR ; rep! : STR × STR

  ─────────────
  
  s? ∉ dom st
  rep! = ("string not in table: ", s?)

  ─────────────

  Rlookup = Lookup ∨ BadLookup
Summary on Refinement

• Need to show abstraction schema Abs between states, and show initializations are OK

• For each concrete-abstract operation pair, need to show Abs lets us prove
  • Applicability: the concrete op can be used whenever the abstract op can and
  • Correctness: if we do the concrete when we could have done the abstract, we get a result we could have gotten from the abstract (after considering Abs)

• Interesting ideas, but full Z refinements are rarely done, except for safety critical systems with strict verification requirements