1 The Truth About Schema Inclusion

- We saw last week how, a schema could be included by just listing its name in the declarations part of a schema. We now look at what this actually means.

- Suppose we had the following definition:

\[
S_1 \\
\begin{array}{c}
v_1 : T_1 \\
v_2 : T_2 \\
P_1 \\
P_2 \\
\end{array}
\]

and later on

\[
S_2 \\
\begin{array}{c}
S_1 \\
v_3 : T_3 \\
P_3 \\
\end{array} (* schema inclusion *)
\]
• Then this would have been equivalent to:

\[ S_2 \begin{array}{c} v_1 : T_1 \\ v_2 : T_2 \\ v_3 : T_3 \\ \hline P_1 \\ P_2 \\ P_3 \end{array} \]
• We now need to introduce *schema decoration*.

• Suppose we had the following declaration:

\[
\begin{array}{c}
S_3 \\
S'_1 \\
S_1 \\
P_4
\end{array}
\]

then this declaration would have been equivalent to

\[
\begin{array}{c}
S_3 \\
v'_1 : T_1 \\
v'_2 : T_2 \\
P_1 \\
P_2 \\
P_4
\end{array}
\]

with all references to \(v_1, v_2\) changed to \(v'_1, v'_2\).

• Remember that the decorated form of a variable means “the variable after the operation has been performed”; the undecorated version means “the variable before the operation has been performed”.

Mike Wooldridge
• Let’s now consider the $\Delta$ notation.

• Suppose we had:

\[
\begin{array}{c}
S_4 \\
\Delta S_1 \\
P_5
\end{array}
\]

• This would have been equivalent to

\[
\begin{array}{c}
S_4 \\
S_1 & (* \text{ include } S_1 *) \\
S'_1 & (* \text{ include } S'_1 *) \\
P_5
\end{array}
\]
• The $\Xi$ notation means something similar. Suppose we had the schema:

\[
\begin{array}{c}
S_5 \\
\Xi S_1 \\
P_5
\end{array}
\]

then this would expand to

\[
\begin{array}{c}
S_5 \\
S_1 \\
S'_1 \\
P_5 \\
v'_1 = v_1 \\
v'_2 = v_2
\end{array}
\]

• So when we use the $\Xi$ notation before a schema, it means “include the decorated and undecorated version of this schema, with the postcondition that all the variables remain unchanged.”
2 The Schema Calculus

• One of the nice things about Z is that it allows us some sort of modular construction; we can build things in little pieces and put them together to make big pieces.

• The way we do this is by using the schema calculus.

• First we need to introduce horizontal form schemas (as opposed to the vertical form schemas we have been looking at so far).
• **Definition:** The following vertical-form schema

\[
S \overline{\begin{array}{c}
\text{Declarations} \\
P_1 \\
P_2 \\
\ldots \\
P_n
\end{array}}
\]

may be defined in the following horizontal form

\[
S \equiv [\text{Declarations} | P_1; P_2; \ldots P_n]
\]

• The symbol \( \equiv \) is for schema definition; it may be read ‘is defined to be’.

• Using \( \equiv \), we can make one schema an alias for another:

\[
\text{NewPhoneBook} \equiv \text{PhoneBooks}
\]

• On the RHS of the \( \equiv \) symbol can be any valid schema calculus expression.
• Such an expression may be a schema definition (as above); but we can also make new schemas using the propositional connectives $\land, \lor, \neg, \Rightarrow, \ldots$. Although these symbols are the same as in propositional logic, they have a different (but related) meaning.

• **Definition:** Two schemas are said to be type compatible if every variable common to both has the same type in both.

• We can use the connectives to make new schemas out of old ones only if they are type compatible. Let $\alpha$ be an arbitrary unary connective, $\beta$ be an arbitrary binary connective, and $S$ and $T$ be the two schemas

\[
S = [D_1; \cdots; D_m | P_1; \cdots; P_n]
\]

\[
T = [D_{m+1}; \cdots; D_{m+p} | P_{n+1}; \cdots; P_{n+q}]
\]

$\alpha S$ is the following schema

\[
[D_1; \cdots; D_m | \alpha(P_1 \land \cdots \land P_n)]
\]

If $S$ and $T$ are type compatible, then $S \beta T$ is the following schema

\[
[D_1; \cdots; D_{m+p} |
(P_1 \land \cdots \land P_n)\beta(P_{n+1} \land \cdots \land P_{n+q})]
\]
• EXAMPLE: Specification of a robust ‘Find’ operation (i.e. one whose behaviour is defined even when the input name is not known).

• First define a schema which assigns the string ‘okay’ to a variable. This schema will be used to signify that an operation has been successful.

\[
\begin{align*}
\text{Success} & \\
\Rightarrow & \quad \text{rep}! : \text{REPORT} \\
\Rightarrow & \quad \text{rep}! = \text{’okay’}
\end{align*}
\]
Then define a schema to capture the situation where a phone number is not in the database. Note that the schema causes an error message to be assigned to the report variable \textit{rep!}.

\[
\text{\textbf{NotKnown}} \\
\equiv \text{\textbf{PhoneBook}} \\
\text{\texttt{name? : NAME}} \\
\text{\texttt{rep! : REPORT}} \\
\text{\texttt{name? \notin \text{known}}} \\
\text{\texttt{rep! = 'name not known'}}
\]
• The robust ‘Find’ operation is

\[
\text{DoFindOp} \equiv (\text{Find} \land \text{Success}) \lor \text{NotKnown}
\]

the full expansion of which is:

\[
\begin{align*}
\text{DoFindOp} & \equiv (\text{dom tel} = \text{known} \land \text{dom tel'} = \text{known} \\
& \land \text{known'} = \text{known} \land \text{tel'} = \text{tel} \\
& \land \text{name?} \in \text{known} \\
& \land \text{phone!} = \text{tel(name?)} \\
& \land \text{rep!} = 'okay') \\
\lor \\
( \text{dom tel} = \text{known} \land \text{dom tel'} = \text{known} \\
& \land \text{known'} = \text{known} \land \text{tel'} = \text{tel} \\
& \land \text{name?} \notin \text{known} \\
& \land \text{rep!} = 'name not known')
\end{align*}
\]
• After logical simplification, the expanded schema becomes:

```plaintext
DoFindOp

\[
\begin{align*}
\text{dom } tel &= \text{known} \\
\land \text{known'} &= \text{known} \land \text{tel'} = \text{tel} \\
\land ( (\text{name}? \in \text{known} \\
\land \text{phone!} &= \text{tel(name?)} \\
\land \text{rep!} &= \text{‘okay’} ) \\
\lor \\
( \text{name}? \not\in \text{known} \\
\land \text{rep!} &= \text{‘name not known’} )
\end{align*}
\]
```
Things to Note

- The use of abstraction: The derived version of DoFindOp is easier to read and understand than the expanded version!

- The behaviour of the system is now rigorously specified. For instance, we could prove that, when the precondition of the find operation is satisfied, then a phone number is found.

- Notice that the value of the variable phone is undefined when the operation fails.