Schema Algebra

Dr. Rong Qu

rxq@cs.nott.ac.uk

http://www.cs.nott.ac.uk/~rxq/#g53fsp
Schema Algebra

Operations which can be performed on whole schema

We look in this section at some of the operations on schema.
### Linear Notation

\[
\text{Schema} \doteq [\text{declaration}_1; \text{declaration}_2 \mid \text{predicate}_1 \land \text{predicate}_2]
\]

<table>
<thead>
<tr>
<th>Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration_1</td>
</tr>
<tr>
<td>declaration_2</td>
</tr>
<tr>
<td>predicate_1</td>
</tr>
<tr>
<td>predicate_2</td>
</tr>
</tbody>
</table>
Schema Extension

We can define a new schema consisting of an existing one but with additional declarations or predicates using the notation

\[
New \triangleq [Old\_schema; new\_declaration \mid new\_predicate]
\]
Basic Type Definition

At the outer level of a specification, we can have, as well as Z schema of various types, basic type definitions of our own

\[ \text{[NAMES, HEIGHT, WEIGHT]} \]
Free Type Definition

At the outer level of a specification, we can also have free type definitions

\[ MESSAGE ::= OK \mid NotKnown \mid AlreadyKnown \]

This creates three global objects \( OK, NotKnown \) and \( AlreadyKnown \) of type \( MESSAGE \).

The identifiers cannot be used elsewhere in the specification.
Schema Inclusion

We have already encountered the concept of schema inclusion in a schema diagram.

The effect is to add

the included schema’s declarations to the declaration part of the current schema

and its predicates to the predicate part.
Ornamentation of Schema Names

When a schema is referred to by name, the name may be ornamented at the beginning or end in certain ways.

If the name has ornamentation added at the end, it is assumed that the ornament has also been appended to all identifiers in the schema.
Ornamentation of Schema Names

The example already encountered is of the use of a prime.

The two accepted ornaments at the start of a schema name are \( \Delta \) and \( \Xi \), which have already been defined.
The tuple and pred Operators

The monadic operator \( \text{tuple} \) acts on a schema, and delivers a tuple of all its declarations.

The monadic operator \( \text{pred} \) acts on a schema, and delivers the conjunction of all its predicates (a logical).

It must not be confused with the precondition operator, \( \text{pre} \), which gives the pre-condition.
Logical Operations on Schema

All the standard logical operators can be used between schema

\[ \iff, \implies, \land, \lor, \neg \]

the declarations of the schema being combined must be compatible.
Logical Operations on Schema

the declaration part of the combined schema is always the union of the declaration parts of the operand schema, irrespective of the particular logical operator involved.

the new predicate part is the two predicate parts logically combined by the given logical operator.
Schema Quantification

Quantifiers can be applied to schema to produce new schema

\[ \forall \text{var} : S \bullet \text{SchemaName} \]

where \text{var} is an object appearing as a declaration inside the schema, and of the same type as the elements in the set.
Schema Quantification

The resulting schema has

a signature omitting the identifiers declared in the quantification, and

a predicate part in which each predicate involving $\textit{var}$ is individually prefaced by

$$\forall \textit{var} : S \bullet$$
Example

For instance, if we have

```
schema_1
  this, that : N
  this \geq that
```
Example

then the schema

\[ \text{schema}_2 \equiv \forall \text{this} : N \bullet \text{schema}_1 \]

has the value

\[
\begin{array}{c}
\text{schema}_2 \\
\hline
\text{that} : N \\
\hline
\forall \text{this} : N \bullet \text{this} \geq \text{that}
\end{array}
\]
Schema Composition

If one schema represents an operation, we may wish to combine it before or after another operation schema.

We wish to define the schema corresponding to the effect of one schema followed by the effect of the other.
Schema Composition

Firstly objects denoted by primed identifiers from the first schema are identified with corresponding objects denoted by unprimed identifiers in the second, where possible.

Thus if the second schema has an object \textit{queue}, and the first an object \textit{queue}', then the \textit{queue}' in the first (the value of \textit{queue} after the first operation) is matched with \textit{queue} in the second (the value before the second operation).

All identifiers in the second schema have an extra prime added.
Schema Composition

Then for any identifiers appearing with double primes, the single primed version is hidden, and the double primed version renamed with a single prime.

Thus primed identifiers in the first schema not matched in the second remain unmodified

and primed identifiers arising in the second which do not occur in the first are also preserved.
ChooseBlock

The schema *ChooseBlock* could be defined
to determine the particular block from the free list to be allocated to the next file extension,
and remove it from the free list,
using a state schema called *FileSystem* defined earlier.
ChooseBlock

We use the definition

\[ \text{ChooseBlock} \]
\[ \Delta \text{FileSystem} \]
\[ \text{next\_block} : \text{BLOCK\_NOS} \]

\[ \text{next\_block} \in \text{free\_blocks} \]
\[ \text{free\_blocks}' = \text{free\_blocks} - \{\text{next\_block}\} \]

The value of \text{next\_block} after this operation is the chosen block, and the block is no longer in the free list.
AddNamedBlock

We can now define $AddNamedBlock$ to add the block specified by $next\_block$ to a named file using

$$
\begin{array}{l}
\Delta FileSystem \\
file\? : FILE\_NAMES \\
next\_block : BLOCK\_NOS
\end{array}
$$

$$
\begin{array}{l}
file\? \in file\_store \\
next\_block \in free\_blocks \\
occupies\' \ file\? = occupies \ file\? \cup \{next\_block\}
\end{array}
$$
ChooseAndAdd

\[ \begin{align*}
\text{ChooseAndAdd} & \quad \Delta \text{FileSystem} \\
\text{file?} & : \text{FILE\_NAMES} \\
\text{next\_block} & : \text{BLOCK\_NOS}
\end{align*} \]

\[
\begin{align*}
\text{file?} & \in \text{file\_store} \\
\text{next\_block} & \in \text{free\_blocks} \\
\text{free\_blocks}' & = \text{free\_blocks} - \{\text{next\_block}\} \\
\text{occupies'} & \text{ file?} = \text{occupies file?} \cup \{\text{next\_block}\}
\end{align*}
\]

The composition is written with a bold semi-colon, as in

\[ \text{ChooseAndAdd} \doteq \text{ChooseBlock};\text{AddNamedBlock} \]
Schema Piping

Piping between two schema represents the composition of two schema as described earlier, with an additional feature that

the output objects of the first schema (those ornamented with an exclamation mark)

are identified

with the corresponding input objects of the second (those ornamented with a question mark)

wherever appropriate declaration exist.
Schema Piping

\[ u \cong s \gg t \]

The declaration part of the combined schema is

the merged declaration parts of the two schema

with those output object of \( s \) which have corresponding
input objects in \( t \) renamed as the input objects and hidden
Example

One operation chooses a number, and the second one uses the chosen value.

The result of the first operation is an output value, instead of a primed variable.

The second operation will pick this up as an input variable.
Example

ChooseValue

\[\text{set} : \mathbb{P} N\]
\[\text{value!} : N\]

\[\text{value!} \in \text{set}\]
\[\forall i : \text{set} \bullet \text{value!} \leq i\]
Example

\[ \text{RemoveIt} \]

\[ \text{set} : P \ N \]
\[ \text{value}?: N \]

\[ \text{value}?: \in \text{set} \]
\[ \text{set}' = \text{set} - \{\text{value}?\} \]

\[ \text{ChooseAndRemove} \triangleq \text{ChooseValue} \gg \text{RemoveIt} \]
Axiomatic Descriptions

This appear as

\[
\begin{align*}
\text{declarations} \\
\text{predicates}
\end{align*}
\]

Axiomatic description introduces one or more global objects and optionally specifies a constraint on their values.

An axiom schema has no name.
Axiomatic Descriptions

These objects must not have a previous global declaration, and cannot be redeclared elsewhere.

The scope of these objects extends from the declaration to the end of the specification.

Declerations and their associated predicates become part of global property.
Example

Another example might be

\[
N\text{files} : \text{NAME} \rightarrow N
\]

\[
\forall \text{user} : \text{system\_users} \bullet
\]

\[
N\text{files user} = \#\text{owns user}
\]

declaring the global function $Nfile$ to deliver the number of files owned by any given user.
Identifier Renaming

Individual identifiers can be renamed temporarily using the notion

\[
New \equiv Old[new\_identifier\_1/old\_identifier\_1, \\
new\_identifier\_2/old\_identifier\_2]
\]

This replaces all occurrences of \textit{old\_identifier\_1} and \textit{old\_identifier\_2} by \textit{new\_identifier\_1} and \textit{new\_identifier\_2} respectively in the new schema.
Identifier Renaming

Renaming is useful when composing schema from several smaller schema, to avoid clashing identifiers.

One schema may be included in another with objects renamed as in

\[
\begin{array}{l}
\underline{NewSchema} \\
ChooseNumber[one/out!] \\
ChooseNumber[two/out!] \\
\hline
one > two
\end{array}
\]
Identifier Hiding

These is a technique for hiding identifiers within a schema. It is written

$$NewSchema \equiv OldSchema \setminus identifier_1$$

or to hide more than one

$$NewSchema \equiv OldSchema \setminus (identifier_1, identifier_2)$$
Identifier Hiding

The effect of hiding is that

the named identifiers are removed from the signature part of the schema,
and existentially quantified wherever they occur in the predicate part.
Example

Thus if

\[
\begin{align*}
\text{\textit{schema\_1}} \quad \text{\textit{this, that}} : N \\
\text{\textit{this < that}}
\end{align*}
\]

then

\[
\text{\textit{schema\_2}} \equiv \text{\textit{schema\_1}} \setminus \text{\textit{this}}
\]
Example

can be defined either as

\[ \text{schema}_2 \equiv \exists \text{this} : N \cdot \text{schema}_1 \]

or can be written in full as

\[
\begin{array}{c}
\text{schema}_2 \\
\hline
\text{that} : N \\
\hline
\exists \text{this} : N \cdot \text{this} < \text{that}
\end{array}
\]

which is true if that \( \geq 1 \).
Schema Pre-condition

The pre-condition of a schema is the schema obtained by hiding (existentially quantifying) all the primed and output (ornamented with an exclamation mark) components.

It is written

\[ \text{pre schema} \]

The predicates then have input components as their only free (undound) components.
Summary

Linear notation, extension, inclusion, tuple, pred, pre, ornamentation, quantification

Logical operations (⇔, ⇒, ∧, ∨, ¬)

Composition (;), Piping

Identifier renaming/hiding

Axiomatic description