Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Author's address from September 1985:
Department of Computing Science
Queensland University
St. Lucia
Queensland 4067
Australia

SPECIFICATION CASE STUDIES

Ian Hayes

Technical Monograph PRG-46

July 1985

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

The first paper is an elementary introduction to the basic ideas of using mathematics to specify computing systems. It consists of three specifications: a simple symbol table, a file update, and a sort.

The second paper is a tutorial example which introduces the schema language for presenting specifications. The example is a symbol table for use in processing a block structured language such as Algol 60.

The third paper is a specification of a sequential file system. It does not contain any extra tutorial explanation of the system being specified; it is an example of the style of specification as would be written in practice.

The fourth paper comes from an industrial case study. It is interesting in that this quite small specification contains the essential parts of a real system. It illustrates some points about specifying real systems that are not covered in the more textbook examples above.

The final paper outlines the basics of a diary system. It provides a quite abstract basis of a diary system which could be developed into a more realistic system.

SPECIFICATION CASE STUDIES

Ian Hayes

Contents

A Symbol Table

The first example specifies a simple symbol table. It demonstrates using a mathematical function to specify an abstract data type. We will specify a symbol table with operations to update, lookup and delete entries in the symbol table. We will describe our table by a partial function from symbols (SYM) to values (VAL)

$$st : SYM \rightarrow\!\!\!\!\!+ \; VAL$$

The arrow $\rightarrow\!\!\!\!\!+$ indicates a function from SYM to VAL that is not necessarily defined for all elements of SYM (hence "partial"). The subset of SYM for which it is defined is its domain of definition

$$dom(st)$$

If a symbol s is in the domain of definition of st ($s \in dom(st)$) then st(s) is the unique value associated with s ($st(s) \in VAL$). The notation $\{ s \mapsto v \}$ describes a function which is only defined for that particular s

$$dom(\{ s \mapsto v \}) = \{ s \}$$

and maps that s onto v

$$\{ s \mapsto v \}(s) = v$$

More generally we can use the notation

$$\{ x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n \}$$

where all the x_k's are distinct to define a function whose domain is

$$\{ x_1, x_2, \ldots, x_n \}$$

and whose value for each x_k is the corresponding y_k. For example, if we let our symbols be names and values be ages we have the following mapping

$$st = \{ \text{“John”} \mapsto 23, \text{“Mary”} \mapsto 19 \}$$

which maps "John" onto 23 and "Mary" onto 19, then the domain of `st` is the set

```
    dom(st) = { "John", "Mary" }
```

and

```
    st("John") = 23
    st("Mary") = 19
```

The range of `st`, `rng(st)`, is the set of values that are associated with a symbol in the table. For the example above

```
    rng(st) = { 19, 23 }
```

The notation {} is used to denote the empty function whose domain of definition is the empty set.

Initially the symbol table will be empty

```
    st = {}
```

We are describing a symbol table by modelling it as a partial function. This use of a function is quite different to the normal use of functions in computing where an algorithm is given to compute the value of the function for a given argument. Here we use it to describe a data structure. There may be many possible models that we can use to describe the same object. Other models of a symbol table could be a list of pairs of symbol and value, or a binary tree containing a symbol and value in each node. These other models are not as abstract because many different lists (or trees) can represent the same function. We would like two symbol tables to be equal if they give the same values for the same symbols. However, it is possible to distinguish between two unordered list representations that as symbol tables are equal; on the other hand, for the function representation different functions represent different symbol tables. The list and tree models of a symbol table tend to bias an implementor working from the specification towards a particular implementation. In fact, both lists and trees could be used to implement such a symbol table. However, any reasoning we wish to perform involving symbol tables is far easier using the partial function model than either the list or tree model.

As some operations can change the symbol table we represent the effect of an operation by the relationship between the symbol table before the operation and the symbol table after the operation. We use

$$st,\ st' : SYM \rightarrow\!\!\!\!+ VAL$$

where by convention we use the undecorated symbol table (st) to represent the state before the operation and the dashed symbol table (st′) the state after. The operation to update an entry in the table is described by the following schema

```
Update___________________________________
| st, st' : SYM -+> VAL
| s? : SYM
| v? : VAL
|--------------------
| st' = st ⊕ { s? ↦ v? }
|________________________________________
```

A schema consists of two parts: the declarations (above the centre line) in which variables to be used in the schema are declared, and a predicate (below the centre line) containing predicates giving properties of and relating those variables. In the schema Update the second line declares a variable with name "s?" which is the symbol to be updated. The third line declares a variable with name "v?" to be the value to be associated with s? in the symbol table. By convention names in the declarations ending in "?" are inputs and names ending in "!" will be outputs; the "?" and "!" are otherwise just part of the name.

The predicate part of the schema states that it updates the symbol table (st) to give a new symbol table (st′) in which the symbol s? is associated with the value v?. Any previous value associated with s? (if there was one) is lost.

The operator ⊕ (function overriding) combines two functions of the same type to give a new function. The new function f ⊕ g is defined at x if either f or g are defined at x, and will have value g(x) if g is defined at x, otherwise it will have value f(x)

$$\mathrm{dom}(f \oplus g) = \mathrm{dom}(f) \cup \mathrm{dom}(g)$$

$$x \in \mathrm{dom}(g) \Rightarrow (f \oplus g)(x) = g(x)$$

$$x \notin \mathrm{dom}(g) \wedge x \in \mathrm{dom}(f) \Rightarrow (f \oplus g)(x) = f(x)$$

For example

$$\begin{aligned}&\{\ \text{“Mary”}\mapsto 19,\ \text{“John”}\mapsto 23\ \}\oplus\{\ \text{“John”}\mapsto 25,\ \text{“George”}\mapsto 62\ \}\\&\quad=\{\ \text{“Mary”}\mapsto 19,\ \text{“John”}\mapsto 25,\ \text{“George”}\mapsto 62\ \}\end{aligned}$$

For the operation Update above the value of $st'(x)$ is $v?$ if $x = s?$, otherwise it is $st(x)$ provided x is in the domain of st. In our example we are only using $\oplus$ to override one value in our symbol table function; the operator $\oplus$ is, however, more general: its arguments may both be any functions of the same type.

The following schema describes the operation to look up an identifier in the symbol table

$$\begin{array}{|l}
\text{LookUp}\\
\hline
st,\ st' : SYM \nrightarrow VAL\\
s? : SYM\\
v! : VAL\\
\hline
s? \in dom(st)\ \wedge\\
v! = st(s?)\ \wedge\\
st' = st\\
\hline
\end{array}$$

The second line of the signature declares a variable with name “s?” which is the symbol to be looked up. The third line of the signature declares a variable with name “v!” which is the value that is associated with $s?$ in the symbol table.

The first line of the predicate states that the identifier being looked up should be in the symbol table before the operation is performed; the above schema does not define the effect of looking up an identifier which is not in the table. The second line states that the output value is the value associated with $s?$ in the symbol table st. The final line states that the contents of the symbol table is not changed by a LookUp operation.

The operation to delete an entry in the symbol table is given by

```
Delete─────────────────────────────────
│  st, st′ : SYM ⇸ VAL
│  s? : SYM
├───────────────
│  s? ∈ dom(st) ∧
│  st′ =  { s? } ⩤ st
└──────────────────────────────────────
```

To delete the entry for $s?$ from the symbol table it must be in the table to start with ($s? \in \mathrm{dom}(st)$). The resultant symbol table st' is the symbol table st with $s?$ deleted from its domain. We use the domain subtraction operator $⩤$ where

$$\mathrm{dom}(s ⩤ f) = \mathrm{dom}(f) - s$$

$$x \in \mathrm{dom}(s ⩤ f) \quad \Rightarrow \quad (s ⩤ f)(x) = f(x)$$

where f is a function and s is a set of elements of the same type as the domain of f. For example

$$\{ \text{“Mary”}, \text{“John”} \} ⩤ \{ \text{“Mary”} \mapsto 19, \text{“John”} \mapsto 25, \text{“George”} \mapsto 62 \}$$
$$= \{ \text{“George”} \mapsto 62 \}$$

Exercise: In place of a single Update operation define two separate operations: Add, to add a symbol and value if the symbol is not already in the table, and Replace, to replace the value associated with a symbol already in the table. □

File Update

The second example is a specification of a simple file update. It uses sets and functions to model the file update operation.

Each record in the file is indexed by a particular key. We will model the file as a partial function from keys to records

$$f : Key \nrightarrow Record$$

A transaction may either delete an existing record or provide a new record which either replaces an existing record or is added to the file. The transactions for an update of a file will be specified as a set of keys d? which are to be deleted from the file, and a partial function u? giving the keys to be updated and their corresponding new records. We add the further restriction that we cannot both delete a record with a given key and provide a new record for that key. For example, if

$$f = \{ k_1 \mapsto r_1, k_2 \mapsto r_2, k_3 \mapsto r_3, k_4 \mapsto r_4 \}$$

$$d? = \{ k_2, k_4 \}$$

$$u? = \{ k_3 \mapsto r_5, k_5 \mapsto r_6 \}$$

then the resultant file f ′ will be

$$f' = \{ k_1 \mapsto r_1, k_3 \mapsto r_5, k_5 \mapsto r_6 \}$$

Our specification is

```
File Update
  f, f′ : Key ⇸ Record
  d? : ℙ Key
  u? : Key ⇸ Record
──────────────
  d? ⊆ dom(f) ∧
  d? ∩ dom(u?) = {} ∧
  f′ = (d? ⩤ f) ⊕ u?
```

The original file f and the updated file f' are modelled by partial functions from keys to records. The keys to be deleted ($d?$) are a subset of Key. Hence $d?$ is an element of the powerset of Key (the set of all subsets of Key); the notation $\mathbb{P}\ Key$ is used to denote the powerset of Key. The updates $u?$ are specified as a partial function from Key to $Record$.

We can only delete records already in the file f. Hence the set of keys to be deleted $d?$ must be a subset of the domain of the original file ($d? \subseteq dom(f)$). We are precluded from trying to both delete a key and add a new record for the same key as the intersection of the deletions with the domain of the updates must be empty ($d? \cap dom(u?) = \{\}$). The resultant file f' is the original file f with all records corresponding to keys in $d?$ deleted ($d? \ntriangleleft f$), overridden by the new records $u?$.

The last line of $File\ Update$ could have equivalently been written

$$f' = d? \ntriangleleft (f \oplus u?)$$

Although it is not always the case that these two lines are equivalent, the extra condition that the intersection of $d?$ and $dom(u?)$ is empty ensures their equivalence in this case.

Lemma: Given $d? \cap dom(u?) = \{\}$ the following identity holds

$$d? \ntriangleleft (f \oplus u?) = (d? \ntriangleleft f) \oplus u?$$

Proof: Firstly we show the domains of the two sides are equal

$$\begin{aligned}
dom(d? \ntriangleleft (f \oplus u?)) &= dom(f \oplus u?) - d? \\
&= (dom(f) \cup dom(u?)) - d? \\
&= (dom(f) - d?) \cup (dom(u?) - d?) \\
&= (dom(f) - d?) \cup dom(u?) \\
&\qquad \text{as } d? \cap dom(u?) = \{\} \\
&= dom(d? \ntriangleleft f) \cup dom(u?) \\
&= dom((d? \ntriangleleft f) \oplus u?)
\end{aligned}$$

Secondly, for any key k in the domain, the two sides are equal. We prove this for the two cases: $k \in dom(u?)$ and $k \notin dom(u?)$.

(a) If $k \in dom(u?)$ then

$$\begin{array}{llll}
 & k \notin d? & & \text{as } dom(u?) \cap d? = \{\} \\
 & (d? \ntriangleleft (f \oplus u?))(k) & = (f \oplus u?)(k) & \text{as } k \notin d? \\
 & & = u?(k) & \text{as } k \in dom(u?) \\
\text{and} & ((d? \ntriangleleft f) \oplus u?)(k) & = u?(k) & \text{as } k \in dom(u?)
\end{array}$$

(b) If $k \notin dom(u?)$ then

$$\begin{array}{llll}
 & (d? \ntriangleleft (f \oplus u?))(k) & = (f \oplus u?)(k) & \text{as } k \in dom(d? \ntriangleleft (f \oplus u?)) \\
 & & = f(k) & \text{as } k \notin dom(u?) \\
\text{and} & ((d? \ntriangleleft f) \oplus u?)(k) & = (d? \ntriangleleft f)(k) & \text{as } k \notin dom(u?) \\
 & & = f(k) & \text{as } k \in dom(d? \ntriangleleft (f \oplus u?)) \quad \square
\end{array}$$

In the specification of File Update if we were not given the extra restriction then, as specified in the last line, updated records would have precedence over deletions. If the alternative specification were used then deletions would have precedence over updates. It is sensible to include the extra restriction in the specification as it allows the most freedom in implementation without any real loss of generality.

Exercise: Define an operation (File Add) to add a number of keys with associated records to a file. The keys should not already be contained in the file. $\square$

Sorting

The third example specifies sorting a sequence into non-decreasing order; it uses bags (multi-sets) and sequences.

The input and the output to Sort are sequences of items of a given type X which has a total order "$<_X$" defined on it. We model a sequence as a partial function from the positive natural numbers ($\mathbb{N}^+$) to the base type X as follows

$$\mathrm{seq}\, X \mathrel{\widehat{=}} \{\ s : \mathbb{N}^+ \twoheadrightarrow X \mid \mathrm{dom}(s) = 1..|s|\ \}$$

where $|s|$ is the number of entries in the mapping s (which is also the length of the sequence s). The notation of enclosing a list of items in angle brackets can be used to construct a sequence consisting of the list of items. For example

$$t = [\ a,\ b,\ c\ ]$$

$$= \{\ 1 \mapsto a,\ 2 \mapsto b,\ 3 \mapsto c\ \}$$

We can select an item in a sequence by indexing the sequeuce with the position of the item

$$t(2) = b$$

$$s = [s(1),\ s(2),\ \dots,\ s(|s|)]$$

The empty sequence is denoted by [ ].

The output of Sort must be in non-decreasing order. We define

```
Non-Decreasing
  s : seq X
  ─────────────
  ∀ i, j : dom(s) •
          (i < j) ⇒ ¬(s(j) <_X s(i))
```

where "$<_X$" is the given total ordering on the base type X.

The output of Sort must contain the same values as the input, with the same frequency. We can state this property using bags. A bag is similar to a set except that multiple occurrences of an element in a bag are significant. We can model a bag as a partial function from the base type X of the bag to the positive integers ($\mathbb{N}^+$) where for each element in the bag the value of the function is the number of times that element occurs in the bag

$$\mathsf{bag}\ X \ \widehat{=}\ X \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}^+$$

We use the notation $[\![\ \ldots \ ]\!]$ to construct a bag. For example

$$[\![1,\ 2,\ 2,\ 2]\!] = \{ 1 \mapsto 1,\ 2 \mapsto 3 \}$$

The following gives some examples of how sets, bags, and sequences (in this case, of natural numbers) are related

$$\{1,2,2,2\} = \{1,2,2\} = \{2,1,2\} = \{1,2\} = \{2,1\}$$

$$[\![1,2,2,2]\!] \neq [\![1,2,2]\!] = [\![2,1,2]\!] \neq [\![1,2]\!] = [\![2,1]\!]$$

$$[1,2,2,2] \neq [1,2,2] \neq [2,1,2] \neq [1,2] \neq [2,1]$$

In specifying Sort we would like to say that the bag formed from all the items in the output sequence is the same as the bag of items in the input sequence. We introduce the function items which forms the bag of all the elements in a sequence. For example

$$\mathsf{items}([\,]) = [\![\,]\!]$$

$$\mathsf{items}([1]) = [\![1]\!]$$

$$\mathsf{items}([1,2,2]) = \mathsf{items}([2,1,2]) = [\![1,2,2]\!]$$

$$\mathsf{items}([1,2,3]) = \mathsf{items}([2,1,3]) = [\![1,2,3]\!]$$

More precisely

```
items : seq X ⇸ bag X

∀ s: seq X •
    items(s) = { x : rng(s) •
                   x ↦ |{ i : dom(s) | s(i) = x }|
               }
```

Each element x that occurs in the sequence is mapped onto its frequency of occurrence in the sequence (i.e. the size of the set of positions in the sequence that have value x).

The specification of sorting is given by

```
Sort──────────────────────────────
│ in?,
│ out! : seq X
├──────────
│ Non-Decreasing[out!/s] ∧
│ items(out!) = items(in?)
└─────────────────────────────────
```

The output of the sort is non-decreasing (in the use of Non-Decreasing above the variable s has been renamed to out! so that the predicate of Non-Decreasing applies to the output of the sort). The output sequence must contain the same items as the input, with the same frequency.

Sort is an example of a non-algorithmic specification. It specifies what Sort should achieve but not how to go about achieving it. The advantage of a non-algorithmic specification is that its meaning may be more obvious than one which contains the extra detail necessary to be algorithmic. The specification is given in terms of the (defining) properties of the problem without biasing the implementor towards a particular form of algorithm. There are many possible sorting algorithms. The implementor should be allowed the freedom to choose the most appropriate.

Exercise: Rewrite the sort specification for the case of sorting a sequence with no duplicates into strictly ascending order. □

References

1. J.-R. Abrial, Programming as a mathematical exercise. In *Mathematical Logic and Programming Languages (eds. C.A.R. Hoare and J.C. Shepherdson)*, Prentice-Hall, 1985.

2. C. C. Morgan and B. A. Snfrin, Specification of the UNIX file system. *IEEE Transactions on Software Engineering, Vol. 10, No. 2,* (March 1984), pp. 128-142.

3. B. A. Sufrin, Mathematics for system specification. *University of Oxford Programming Research Group lecture notes*, 1983-84.

4. P. Halmos, *Naive Set Theory*. Springer-Verlag, 1974.

Solutions to Exercises

Symbol table

```
Add_____________________________________
│ st,st′ : SYM ⇸ VAL
│ s? : SYM
│ v? : VAL
├──────────────────
│ s? ∉ dom(st) ∧
│ st′ = st ∪ { s? ↦ v? }
└───────────────────────────────────────
```

```
Replace_________________________________
│ st, st′ : SYM ⇸ VAL
│ s? : SYM
│ v? : VAL
├──────────────────
│ s? ∈ dom(st) ∧
│ st′ = st ⊕ { s? ↦ v? }
└───────────────────────────────────────
```

File add

```
File Add────────────────────────────────
│ f, f′ : Key ⇸ Record
│ a?    : Key ⇸ Record
├──────────────
│ dom(a?) ∩ dom(f) = {} ∧
│ f′ = f ∪ a?
└────────────────────────────────────────
```

Sorting

```
NoDuplicates────────────────────────────
│ s : seq X
├──────────────
│ ∀ i, j : dom(s) •
│           (i ≠ j) ⇒ (s(i) ≠ s(j))
└────────────────────────────────────────
```

```
Ascending───────────────────────────────
│ s : seq X
├──────────────
│ ∀ i, j : dom(s) •
│           (i < j) ⇒ (s(i) <_X s(j))
└────────────────────────────────────────
```

```
Sort────────────────────────────────────
│ in?, out! : seq X
├──────────────
│ NoDuplicates[in?/s] ∧
│ Ascending[out!/s] ∧
│ rng(in?) = rng(out!)
└────────────────────────────────────────
```

Block-Structured Symbol Table Specification

Abstract

A specification of a symbol table for a block structured language is given. This specification is intended to demonstrate how using the specification notation Z,[1,2,3] a specification can be built from components.

A simple symbol table suitable for a single block is described first; it has operations to look up, update and delete entries. This simple symbol table is the same as that given in the section entitled "Examples of Specification Using Mathematics"[4] preceding this section. The treatment given here differs from that in the earlier paper in that it emphasises how such a specification can be built using the schema notation of Z[5] and includes a treatment of error conditions not given in the earlier paper. Readers not familiar with the mathematics used in this specification should consult the earlier paper for a more detailed explanation.

The second part of this paper specifies a block structured symbol table in terms of a sequence of simple symbol tables; one for each nested block. Operations are given to search the *environment* for a symbol, and to start and finish nested blocks; the operations on a simple symbol table are upgraded to work on the symbol table corresponding to the smallest enclosing block.

Explanations of notations in the paper are given in *italics* within the paper and a summary of the notations used is given in an appendix.

Symbol Table

A symbol table associates a unique value (from the set VAL) with a symbol (from the set SYM). The operations allowed on a symbol table are to:

- update the value associated with a symbol in the table; if the symbol is not already in the table it will be added,
- look up the value associated with a symbol in the table, aud
- delete a symbol and its associated value from the table.

To specify an abstract data type for a symbol table we first give a model of the state of a symbol table and a description of the initial state, then we specify each of the operations on a symbol table in terms of the relationship between the state before an operation, the inputs to the operation, the outputs from the operation, and the state after the operation.

The State

The state of a symbol table can be modelled by a partial function from symbols to values

$$\mathsf{ST} \mathrel{\widehat=} \mathsf{SYM} \nrightarrow \mathsf{VAL}$$

Initially the symbol table is empty

$$\mathsf{st}_{\mathsf{INIT}} \mathrel{\widehat=} \{\}$$

Operations

Each operation on a symbol table transforms a symbol table before (st) into a symbol table after (st').

$$\Delta\mathsf{ST} \mathrel{\widehat=} [\ \mathsf{st},\ \mathsf{st}' : \mathsf{ST}\ ]$$

The definition of each operation must include declarations of the before and after states of the operation; rather than write out these declarations in full in each definition, we introduce a schema ΔST that contains just these declarations and include this schema in the definition of each operation as an *abbreviation for the declarations. The "Δ" (for "change") in "ΔST" is just part of the name of the schema; we allow Greek letters in names. By convention names beginning with "Δ" are used for schemas that contain before and after state components.*

Error handling and the operation to look up a symbol do not modify the symbol table.

```
    ≡ST  ≙   [ ΔST | st′ = st ]
```

The schema ≡ST declares the before and after states (in ΔST) and constrains them to be equal; this schema describes the effect on the state of inquiry-like operations (such as looking up a symbol in the symbol table) and error handling; both of these do not modify the state. The "≡" (for no change) in "≡ST" is again just part of the name. By convention names beginning with "≡" are used for schemas which are written to express that there is no change.

An extra constraint may be added to the predicate part of a schema by following the schema with a "|" followed by the predicate. The additional predicate is and'ed with the existing predicate of the schema to form the predicate of the resulting schema; in the case of ≡ST the existing predicate is true (the default when no predicate is given as in ΔST). Expanding the definition of ≡ST we get

```
≡ST  ≙
┌──────────────────────────────
│ st, st′ : ST
├──────────────
│ st′ = st
└──────────────────────────────
```

To look up the value v! associated with a symbol s? we use

```
┌─LookUp───────────────────────
│ ≡ST
│ s? : SYM
│ v! : VAL
├──────────────
│ s? ∈ dom(st) ∧
│ v! = st(s?)
└──────────────────────────────
```

The schema ΞST *is used in the definition of* LookUp *to declare the before and after states (*st *and* st*′) and to constrain them to be equal. The convention of using the* ΞST *schema saves writing out all the state components and the equality constraint explicitly.*

A schema may be included in the declaration part of a schema; the declarations of the included schema are merged with the other declarations and its predicates are and'ed with the predicates of the schema.

```
LookUp
≙
┌──────────────────────────────────
│ st, st′ : ST
│ s?  : SYM
│ v!  : VAL
├──────────────
│ st′ = st  ∧
│ s?  ∈ dom(st) ∧
│ v!  = st(s?)
└──────────────────────────────────
```

To update the value associated with a symbol we use

```
Update ───────────────────────────
│ ΔST
│ s? : SYM
│ v? : VAL
├──────────────
│ st′ = st ⊕ { s? ↦ v? }
└──────────────────────────────────
```

This schema uses ΔST *to include the declarations of the before and after states.*

If the symbol was already in the table its old value is replaced by v?; if it was not in the table it is added.

To delete an entry in the symbol table we use

```
Delete______________________________
│ ΔST
│ s? : SYM
├──────────────
│ s?  ∈ dom(st) ∧
│ st' = {s?} ⩤ st
└───────────────────────────────────
```

Errors

LookUp and Delete are only defined if the symbol is present in the table. If the symbol is not present an error is reported and the symbol table is not modified.

```
NotPresent!_________________________
│ ΞST
│ s?   : SYM
│ rep! : Report
├──────────────
│ s? ∉ dom(st) ∧
│ rep! = "Symbol not present"
└───────────────────────────────────
```

The schema ΞST is included in the above schema to introduce the declarations of the before and after states and constrain them to be equal.

A convention used within this specification is that schemas denoting errors have names ending in "!"; the "!" is just part of the name.

Successful operations return a report of "OK".

```
Success  ≙  [ rep! : Report | rep! = "OK" ]
```

The operations with error handling are

```
STLookUp   ≙  (LookUp  ∧ Success) ∨ NotPresent!

STUpdate   ≙   Update  ∧ Success

STDelete   ≙  (Delete  ∧ Success) ∨ NotPresent!
```

Either a LookUp *operation can be successfully performed (if* s? ∈ dom(st)*), in which case a report of "*OK*" is given, or the* LookUp *cannot be performed (if* s? ∉ dom(st)*), in which case an error report of "*Symbol not present*" is given.*

The conjunction (∧) of two schemas is formed by merging their declarations (variables common to both declarations must have the same type) and and'ing their predicates. Below we give expanded versions of the look up operation. We do not normally find it necessary to expand such definitions to understand the specification but the expansions are intended to help people who are not familiar with the notation.

```
LookUp ∧ Success
≙
┌──────────────────────────────
│ ΞST
│ s?   : SYM
│ v!   : VAL
│ rep! : Report
├──────────────────────────────
│ s?   ∈ dom(st) ∧
│ v!   = st(s?) ∧
│ rep! = "OK"
└──────────────────────────────
```

In this example there are no common variables.

The disjuction (∨) of two schemas is formed by merging their declarations (variables common to both must have the same type) and or'ing their predicate parts.

```
STLookUp
≙
┌──────────────────────────────────────────
│ ΞST
│ s?     : SYM
│ v!     : VAL
│ rep!   : Report
├───────────────────────
│ (s? ∈ dom(st) ∧
│  v! = st(s?) ∧
│  rep! = "OK")
│ ∨
│ (s? ∉ dom(st) ∧
│  rep! = "Symbol not present")
└──────────────────────────────────────────
```

In this example the declarations in ΞST *and the declarations of* s? *and* rep! *are common and have the same types, and hence can be merged. Note that no constraint is placed on the value of* v! *returned in the error case.*

<u>Exercise 1:</u> Give expanded forms of the schemas STUpdate and STDelete. □

Block-Structured Symbol Table

We will now describe a symbol table suitable for use in processing (e.g., compiling) a block-structured language such as Algol 60. In such languages each variable declaration is associated with a block and a variable may be referenced only from within the block with which it is associated. Blocks may be nested within other blocks to an arbitrary level; each nested block must be completely enclosed by the block in which it is included. For example, consider the following fragment of Algol 60

```
begin A
      integer x, y;
      . . .
      x := 2; y := 3;                          (1)
      . . .
      begin B
            real y; integer z;
            . . .
            y := 0.5;  x := z;                 (2)
            . . .
      end B;
      . . .
      y := x;                                  (3)
      . . .
end A
```

The outer block A declares variables x and y of type integer. These variables may be referenced anywhere within block A, except that the variable y of block A may not be referenced within block B because there is a variable with the same name declared in block B; within block B the outer (block A) declaration of y is "hidden" by the declaration of y in block B. We refer to those parts of the program in which a variable may be referenced as being within the "scope" of that variable.

A symbol table suitable for sequential processing of a block-structured language should support the scoping rules of block-structured languages; it should have operations for starting and finishing blocks as well as operations to access, update and delete entries in the table.

The State

The simple symbol table described in the earlier part of this paper is suitable only for keeping track of the variables of a single block. At a given point in a program we need to keep track of all the variables declared in all the blocks enclosing that point; this can be done by associating a simple symbol table with each block enclosing the point. To keep track of the order in which the blocks are nested we will arrange the symbol tables into a sequence so that, if a block A encloses another block B, the symbol table for A will precede the symbol table for B in the sequence. We can model a block-structured symbol table by

$$BST \mathrel{\widehat{=}} \text{seq } ST$$

The first symbol table in the sequence is for the outermost block enclosing a poiut.

In the example given above, the block-structnred symbol table within block A but excluding block B (e.g., at the positions marked (1) and (3)) will be a sequence contaiuing a single symbol table

$$[\ \{\ x \mapsto \text{integer},\ \ y \mapsto \text{integer}\ \}\ ]$$

Within block B (e.g., at the position marked (2)) the sequeuce contains two symbol tables

$$[\ \{\ x \mapsto \text{integer},\ y \mapsto \text{integer}\ \},\ \{\ y \mapsto \text{real},\ z \mapsto \text{integer}\ \}\ ]$$

At auy point within a program at most one variable of a given name may be referenced. We will refer to the variables that may be referenced at a given point, aloug with their associated information, as the "environment" of that point. An environment may be represented as a simple symbol table. In the example above, the environment within block A but excluding block B (e.g., (1) and (2)) is

$$\{\ x \mapsto \text{integer},\ y \mapsto \text{integer}\ \}$$

and within block B it is equal to the symbol table for block A overridden by the symbol table for block B

$$\{\ x \mapsto \text{integer},\ y \mapsto \text{integer}\ \} \oplus \{\ y \mapsto \text{real},\ z \mapsto \text{integer}\ \}$$

$$= \{\ x \mapsto \text{integer},\ y \mapsto \text{real},\ z \mapsto \text{integer}\ \}$$

In general, if we have a block-structured symbol table consisting of a sequence of symbol tables the environment is given by overriding the symbol tables in sequence. For example, for the sequence

$$[\ st_1,\ st_2,\ \ldots,\ st_n\ ]$$

the environment is

$$st_1 \oplus st_2 \oplus \ldots \oplus st_n.$$

We can define the distributed override operator $\oplus/$ which extracts the environment from a sequence of symbol tables by

$$\oplus/ : \text{seq } ST \longrightarrow ST$$

$$\oplus/[\,] = \{\}$$

$$\oplus/(s \frown [t]) = (\oplus/s) \oplus t$$

Initially no blocks have been entered; hence the block structured symbol table is the empty sequence

$$bst_{INIT} \mathrel{\widehat{=}} [\,]$$

Operations

The operations on a block-structured symbol table transform a state before (bst) to a state after (bst′).

```
ΔBST  ≙  [ bst, bst′ : BST ]
```

Some operations leave the state unchanged.

```
ΞBST  ≙  [ ΔBST | bst′ = bst ]
```

There are two operations which retrieve information about a symbol from a block-structured symbol table: BLookUp and BSearch. BLookUp looks in the mosted nested symbol table only; it will be defined in terms of STLookUp. BSearch searches for a symbol in the environment (i.e., the most nested occurrence of the symbol in the block structured symbol table).

```
BSearch₀ ────────────────────────────────
│ ΞBST
│ s? : SYM
│ v! : VAL
├──────────────
│ s? ∈ dom(⊕/bst) ∧
│ v! = (⊕/bst)(s?)
└────────────────────────────────────────
```

When the start of a block is encountered a new (empty) symbol table is appended to the sequence

```
BStart₀ ─────────────────────────
│ ΔBST
├──────────────
│ bst′ = bst ⁀ [st_INIT]
└─────────────────────────────────
```

When the end of a block is encountered the last symbol table in the sequence is deleted

```
BEnd0 ──────────────────────────────────
│ ΔBST
├──────────────────
│ bst  ≠ []     ∧
│ bst' = front(bst)
└──────────────────────────────────────
```

We want to be able to perform the simple symbol table operations (STLookUp, STUpdate and STDelete) on the most nested (last) symbol table in the sequence. These operations can only be performed provided the sequence is non-empty, and they only change the last symbol table in the sequence. The relationship between the before and after values of the last symbol table in the sequence is determined by the simple symbol table operations.

The common part of the three upgraded operations is given by

```
Upgrade ────────────────────────────────
│ ΔBST
│ ΔST
├──────────────────
│ bst ≠ []  ∧
│ front(bst') = front(bst) ∧
│ st  = last(bst) ∧
│ st' = last(bst')
└──────────────────────────────────────
```

The above description does not specify the relationship between the last symbol table in the sequence before (st) and after (st') an operation; we have already described these relationships in our definitions of the simple symbol table operations. We can now define the upgraded symbol table operations in terms of the definitions of the simple symbol table operations given earlier.

The upgraded operations are given by

$$\mathsf{BLookUp}_0 \ \widehat{=}\ (\mathsf{STLookUp} \wedge \mathsf{Upgrade}) \setminus \Delta\mathsf{ST}$$

$$\mathsf{BUpdate}_0 \ \widehat{=}\ (\mathsf{STUpdate} \wedge \mathsf{Upgrade}) \setminus \Delta\mathsf{ST}$$

$$\mathsf{BDelete}_0 \ \widehat{=}\ (\mathsf{STDelete} \wedge \mathsf{Upgrade}) \setminus \Delta\mathsf{ST}$$

A schema may have a list of its components hidden by use of schema hiding ("\"). The declarations of the hidden variables are removed from the declaration part of the schema and are existentially quantified in the predicate part. If the second operand to "\" is a schema then all the variables in the declaration part of the second schema are hidden in the first schema.

The components of ΔST *(*st *and* st′*) are hidden in the above definitions because we wish to define the operations as working on before and after states which are of type* BST*; the* ΔST *components are only used to make the link between the specifications of the operations on the simple symbol table and the part of the* BST *state that the simple operations are to be performed on. The reason for introducing* Upgrade *is to allow the definitions of the operations on simple symbol tables to be used directly in the definitions of the operations on block structured symbol tables.*

$\mathsf{BUpdate}_0$
$\widehat{=}$

```
ΔBST
s?    : SYM
v?    : VAL
rep!  : Report
─────────────────────────
(∃ st, st′ : ST •
  bst ≠ [] ∧
  front(bst′) = front(bst) ∧
  st  = last(bst) ∧
  st′ = st ⊕ { s? ↦ v? } ∧
  rep! = "OK" ∧
  st′ = last(bst′)
)
```

BUpdate$_0$ *may be simplified to*

```
ΔBST
s?   : SYM
v?   : VAL
rep! : Report
─────────────
bst ≠ [] ∧
front(bst') = front(bst) ∧
last(bst') = last(bst) ⊕ { s? ↦ v? } ∧
rep! = "OK"
```

Errors

The upgraded operations and BEnd will fail if the sequence is empty

```
Empty! ________________________________________
| ≡BST
| rep!  : Report
|__________________
| bst   = []  ∧
| rep!  = "Not within any block"
|________________________________________
```

The BSearch operation will fail if the symbol is not in the environment. If the sequence is empty we give preference to the Empty! error, hence for this error we require that the sequence is non-empty.

```
NotFound! _____________________________________
| ≡BST
| s?    : SYM
| rep!  : Report
|__________________
| bst ≠ [] ∧
| s? ∉ dom(⊕/bst) ∧
| rep!  = "Symbol not found"
|________________________________________
```

The final definitions of the operations are

$$\mathrm{BSearch} \mathrel{\widehat{=}} (\mathrm{BSearch}_0 \wedge \mathrm{Success}) \vee \mathrm{NotFound!} \vee \mathrm{Empty!}$$

$$\mathrm{BStart} \mathrel{\widehat{=}} \mathrm{BStart}_0 \wedge \mathrm{Success}$$

$$\mathrm{BEnd} \mathrel{\widehat{=}} (\mathrm{BEnd}_0 \wedge \mathrm{Success}) \vee \mathrm{Empty!}$$

$$\mathrm{BLookUp} \mathrel{\widehat{=}} \mathrm{BLookUp}_0 \vee \mathrm{Empty!}$$

$$\mathrm{BUpdate} \mathrel{\widehat{=}} \mathrm{BUpdate}_0 \vee \mathrm{Empty!}$$

$$\mathrm{BDelete} \mathrel{\widehat{=}} \mathrm{BDelete}_0 \vee \mathrm{Empty!}$$

The expanded and simplified definition of BSearch *is*

```
≡BST
s? : SYM
v! : VAL
rep! : Report
─────────────────
 (s? ∈ dom(⊕/bst) ∧
  v! = (⊕/bst)(s?) ∧
  rep! = "OK")
∨
 (bst ≠ [] ∧
  s? ∉ dom(⊕/bst) ∧
  rep! = "Symbol not found")
∨
 (bst = [] ∧
  rep! = "Not within any block")
```

<u>Exercise 2:</u> Give an expansion of BDelete. □

<u>Exercise 3:</u> Define a search operation BLocate that returns not only the value associated with a symbol but also the level of the innermost block in which it is declared. □

References

1. Abrial, J.-R. The specification language Z: Basic library. *Oxford University Programming Research Group internal report*, (April 1980).

2. Morgan, C. C., and Sufrin, B. A. Specification of the UNIX file system. *IEEE Transactions on Software Engineering, Vol. 10, No. 2*, (March 1984), pp. 128-142.

3. Sufrin, B. A. Mathematics for system specification. *University of Oxford Programming Research Group lecture notes*, 1983-84.

4. Hayes, I. J. Examples of specification using mathematics. *Alvey Software Engineering Newsletter*, (January 1985).

5. Morgan, C. C. Schemas in Z: A preliminary reference manual. *Oxford University Programming Research Group Distributed Computing Project report*, (March 1984).

Solutions to Exercises

1.

```
STUpdate
  ΔST
  s? : SYM
  v? : VAL
  rep! : Report
  ─────────────
  st' = st ⊕ { s? ↦ v? } ∧
  rep! = "OK"
```

```
STDelete
  ΔST
  s? : SYM
  rep! : Report
  ─────────────
  (s? ∈ dom(st) ∧
   st' = { s? } ⩤ st ∧
   rep! = "OK")
 ∨
  (s? ∉ dom(st) ∧
   st' = st ∧
   rep! = "Symbol not found")
```

2.
```
BDelete________________________________________________
  ΔBST
  s? : SYM
  rep! : Report
________________
  (∃ st, st' : ST •
    bst ≠ [] ∧
    front(bst') = front(bst) ∧
    st = last(bst) ∧
    ((s? ∈ dom(st) ∧ st' = { s? } ⩤ st ∧ rep! = "OK")
    ∨(s? ∉ dom(st) ∧ st' = st ∧ rep! = "Symbol not found")
    ) ∧
    st' = last(bst'))
 ∨
  (bst = [] ∧
   bst' = bst ∧
   rep! = "Not within any block")
_______________________________________________________
```

This is equivalent to

```
BDelete________________________________
  ΔBST
  s? : SYM
  rep! : Report
________________
  (bst ≠ [] ∧ s? ∈ dom(last(bst)) ∧
   front(bst') = front(bst) ∧
   last(bst') = { s? } ⩤ last(bst) ∧
   rep! = "OK")
 ∨
  (bst ≠ [] ∧ s? ∉ dom(lest(bst)) ∧
   bst' = bst ∧
   rep! = "Symbol not found")
 ∨
  (bst = [] ∧
   bst' = bst ∧
   rep! = "Not within any block")
_______________________________________
```

3.

$BLocate_0$

$$\begin{array}{|l}
\Xi BST \\
s? : SYM \\
v! : VAL \\
level! : \mathbb{N} \\
\hline
bst \neq [\,] \land \\
s? \in \mathrm{dom}(\oplus/bst) \land \\
level! = \max \{ \imath : \mathrm{dom}(bst) \mid s? \in \mathrm{dom}(bst(\imath)) \} \land \\
v! = bst(level!)(s?)
\end{array}$$

$$BLocate \mathrel{\widehat{=}} (BLocate_0 \land Success) \lor NotFound! \lor Empty!$$

Sequential File Specification

Ian Hayes and Ib Holm Sørensen

Abstract

This specification describes a file system with operations to

- open, close and abort access to a file,
- sequentially read and write an open file,
- reposition, find out the current position, and find out the length of an open file, and
- delete an existing closed file.

The specification is organised as follows: the action of operations on an individual file (Read, Write, Reposition, Position, and Length) are described followed by the related error conditions; these operations are then upgraded to specify their action on the state of the whole file system (i.e. as operating on an individual file in the larger state); operations to open, close and abort access to a file and to delete a file are defined; error conditions for files being nonexistent, not open, or already open are defined and the final definitions of all the operations, complete with error handling, are given.

Individual Files

We can represent the contents of a file by a sequence of basic units

$$\mathsf{File} \ \widehat{=}\ \mathsf{seq\ Unit}$$

where Unit is the smallest addressable unit of information in a file (e.g., a byte).

Initially a file is empty

$$\mathsf{File}_{\mathsf{INIT}} \ \widehat{=}\ [\,]$$

When a file is in use (open) for sequential reading or writing we need to keep track of how much of the file has been processed. We can model this by introducing two sequences: the part already $\mathsf{processed}$, and the $\mathsf{remainder}$ of the file.

```
OpenFile_______________________________
| contents,
| processed,
| remainder : File
|_______________
| contents = processed ⁀ remainder
|______________________________________
```

The contents of the file is the concatenation of the processed part and the remainder. The current position in a file is the point between its two parts.

Operations

The operations transform a state before ($\mathsf{OpenFile}$) into a state after ($\mathsf{OpenFile}'$).

$$\Delta\mathsf{OpenFile} \ \widehat{=}\ \mathsf{OpenFile} \wedge \mathsf{OpenFile}'$$

If the state of a file is unchanged by an operation we use

$$\Xi\mathsf{OpenFile} \ \widehat{=}\ [\ \Delta\mathsf{OpenFile} \mid \mathsf{OpenFile}' = \mathsf{OpenFile}\ ]$$

A Read operation returns a sequence $v!$ of up to $n?$ units in length starting at the current position in the file

$$\begin{array}{|l}
\textsf{Read}_0 \\
\hline
\Delta \textsf{OpenFile} \\
n? : \mathbb{N} \\
v! : \textsf{File} \\
\hline
v! = (1..n?) \lhd \textsf{remainder} \land \\
\textsf{processed}' = \textsf{processed} \frown v! \land \\
\textsf{contents}' = \textsf{contents} \\
\hline
\end{array}$$

The value returned is the initial part of the remainder of the file; if $|\textsf{remainder}| \geqslant n?$ then the length of $v!$ will be $n?$; if $|\textsf{remainder}| \leqslant n?$ then $v!$ will be the whole of $\textsf{remainder}$. The current position moves to the end of the portion read. The contents of the file is unchanged.

The sequence read, concatenated with the final remainder, is equal to the initial remainder

$$\textsf{Read}_0 \quad \Rightarrow \quad v! \frown \textsf{remainder}' = \textsf{remainder}$$

A $\textsf{Write}$ operation changes a file by overwriting the initial part of the remainder of the file with an input sequence $v?$. If necessary the file will be extended to accomodate the extra information.

$$\begin{array}{|l}
\textsf{Write}_0 \\
\hline
\Delta \textsf{OpenFile} \\
v? : \textsf{File} \\
\hline
\textsf{contents}' = \textsf{processed} \frown (\textsf{remainder} \oplus v?) \land \\
\textsf{processed}' = \textsf{processed} \frown v? \\
\hline
\end{array}$$

The current position moves to the end of the sequence written. If the remainder is empty before a $\textsf{Write}$ then the input sequence is appended to the end of the file.

$$\begin{array}{l}
\textsf{Write}_0 \mid \textsf{remainder} = [\,] \\
\Rightarrow \quad \textsf{contents}' = \textsf{contents} \frown v?
\end{array}$$

To move to a particular position in the file we can use the Reposition operation

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathsf{Reposition}_0} \\ \hline
\Delta \mathsf{OpenFile} \\
p? : \mathbb{N} \\ \hline
p? \in 0 \,..\, |contents| \land \\
|processed'| = p? \land \\
contents' = contents \\ \hline
\end{array}$$

Provided p? refers to a valid position in the file, a Reposition will position the file so that the processed part of the file is of length p?. The contents of the file is unchanged.

To find out the current position in the file and the length of the file we use, respectively

$$\mathsf{Position}_0 \ \widehat{=}\ [\ \Xi \mathsf{OpenFile};\ p! : \mathbb{N} \mid p! = |processed|\ ]$$

$$\mathsf{Length}_0 \ \widehat{=}\ [\ \Xi \mathsf{OpenFile};\ n! : \mathbb{N} \mid n! = |contents|\ ]$$

The contents and current position of the file are unchanged by both Position and Length.

Exercise 1: Define an operation to truncate a file to the current processed part leaving the file positioned at its end. □

Errors

A Reposition can fail if the requested new position is outside the bounds of the file. All other operations are total.

```
OutofBounds! ______________________________
| ΞOpenFile
| p? : ℕ
| rep! : Report
|___________________
| p? ∉ 0..|contents| ∧
| rep! = "Position out of bounds"
|__________________________________________
```

where Report ≙ seq Char.

Successful operations return the report "OK"

```
Success  ≙  [ rep! : Report | rep! = "OK" ]
```

The operations with error handling are

```
Read        ≙  Read₀         ∧ Success

Write       ≙  Write₀        ∧ Success

Reposition  ≙ (Reposition₀   ∧ Success) ∨ OutofBounds!

Position    ≙  Position₀     ∧ Success

Length      ≙  Length₀       ∧ Success
```

Named Files

File names are sequences of characters.

$$Name \mathrel{\widehat{=}} \mathrm{seq}\ Char.$$

We will represent a file system by the currently open files (open) and a file store (fs) which contains the contents of the closed files plus the contents of open files at the time they were opened if they existed in the file store at that time; the latter are kept so that a sequence of operations on a file may be aborted and the file reverts to its state prior to being opened. As we need to keep track of the current position of open files each is represented by an OpenFile; the files in the file store are just represented by their contents.

$$\begin{array}{|l}
\multicolumn{1}{l}{FSys} \\ \hline
open : Name ⇻ OpenFile \\
fs \quad : Name ⇻ File \\ \hline
\end{array}$$

Initially a file system is empty

$$FSys_{INIT} \mathrel{\widehat{=}} [\ FSys \mid fs = \{\} \wedge open = \{\}\ ]$$

File system operations transform a state before (FSys) into a state after (FSys′).

$$\Delta FSys \mathrel{\widehat{=}} FSys \wedge FSys'$$

The operations defined previously on individual files may be used to operate on an open file.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{FSUpgrade}} \\
\hline
\Delta \mathit{FSys} \\
\mathit{fn?} : \mathit{Name} \\
\Delta \mathit{OpenFile} \\
\hline
\mathit{fn?} \in \mathrm{dom}(\mathit{open}) \land \\
\mathit{OpenFile} = \mathit{open}(\mathit{fn?}) \land \\
\mathit{open}' = \mathit{open} \oplus \{\mathit{fn?} \mapsto \mathit{OpenFile}'\} \land \\
\mathit{fs}' = \mathit{fs} \\
\hline
\end{array}$$

The appropriate file, which must be open, is selected and updated by one of the operations on a file. The file store does not change.

The upgraded operations follow

$$\begin{array}{lcll}
\mathit{FRead}_0 & \widehat{=} & (\mathit{Read} & \land\ \mathit{FSUpgrade}) \setminus \Delta \mathit{OpenFile} \\
\mathit{FWrite}_0 & \widehat{=} & (\mathit{Write} & \land\ \mathit{FSUpgrade}) \setminus \Delta \mathit{OpenFile} \\
\mathit{FReposition}_0 & \widehat{=} & (\mathit{Reposition} & \land\ \mathit{FSUpgrade}) \setminus \Delta \mathit{OpenFile} \\
\mathit{FPosition}_0 & \widehat{=} & (\mathit{Position} & \land\ \mathit{FSUpgrade}) \setminus \Delta \mathit{OpenFile} \\
\mathit{FLength}_0 & \widehat{=} & (\mathit{Length} & \land\ \mathit{FSUpgrade}) \setminus \Delta \mathit{OpenFile}
\end{array}$$

The file operations defined so far are only permitted on open files. We need operations to open a file, close a file (saving its contents in the file store), and abort use of an open file (uot saving its contents).

```
Open____________________________________________
  ΔFSys
  fn? : Name
 ______________
  ∃ OpenFile •
    fn? ∉ dom(open) ∧
    fn? ∈ dom(fs)  ⇒  contents = fs(fn?) ∧
    fn? ∉ dom(fs)  ⇒  contents = [] ∧
    processed = [] ∧
    open′ = open ∪ { fn? ↦ OpenFile } ∧
    fs′ = fs
________________________________________________
```

The file must not already be open. If the file exists the contents of the open file become the contents of the file in the file store otherwise the open file is initially empty. The opened file is positioned at its beginning. The file store remains unchanged.

```
Close___________________________________________
  ΔFSys
  fn? : Name
 ______________
  fn? ∈ dom(open) ∧
  fs′ = fs ⊕ { fn? ↦ open(fn?).contents } ∧
  open′ = { fn? } ⩤ open
________________________________________________
```

The file must be open. The contents of the open file replaces any previous value (i.e. the contents of the file at the time of the last open) in the file store. The file is deleted from the open files.

```
Abort_________________________________
|  ΔFSys
|  fn? :  Name
|_____________________
|
|  fn? ∈  dom(open)   ∧
|  open′  = { fn? } ⩤ open ∧
|  fs′ =  fs
|______________________________________
```

The file must be open. It is deleted from the open files. The contents of the file store remains unchanged. If the file existed in the file store before being opened its previous value remains in the file store; if it did not exist it still does not.

If there is a system crash while the file system is in use it is intended that the effect of the crash should be the same as if an Abort operation was performed for each open file.

```
Crash_________________________________
|  ΔFS
|_____________________
|
|  open  =  {} ∧
|  fs′  =  fs
|______________________________________
```

The final operation needed is that to delete an existing file.

```
Delete _______________________________
|  ΔFSys
|  fn? :  Name
|_____________________
|
|  fn? ∈  (dom(fs) - dom(open)) ∧
|  fs′  =  {fn?} ⩤ fs ∧
|  open′  = open
|______________________________________
```

The file to be deleted must exist in the file store and not currently be open. The file is deleted from the file store. The open files do not change.

Exercise 2: Define operations to rename a file and to copy a file. Does the following hold for your definitions? Rename ≡ Copy ⨾ Delete[oldfn?/fn?] □

Errors

All errors leave the file system unchanged

$$\Xi FSysError \mathrel{\widehat=} [\ \Delta FSys;\ fn?{:}Name;\ rep!{:}Report \mid FSys' = FSys\ ]$$

If the file does not exist then `Delete` will fail.

```
FileNonExistent!
  ΞFSysError
  ─────────────────
  fn? ∉ dom(fs)  ∧
  rep! = fn? ⁀ "does not exist."
```

If the file is open `Delete` and `Open` will fail.

```
FileOpen!
  ΞFSysError
  ─────────────────
  fn? ∈ dom(open)  ∧
  rep! = fn? ⁀ " is open."
```

If the file is not open the operatious on individual files and `Close` and `Abort` will fail.

```
FileNotOpen!
  ΞFSysError
  ─────────────────
  fn? ∉ dom(open)  ∧
  rep! = fn? ⁀ " not open."
```

The operations with error handling are

```
FRead        ≙    FRead₀
                ∨ FileNotOpen!

FWrite       ≙    FWrite₀
                ∨ FileNotOpen!

FReposition  ≙    FReposition₀
                ∨ FileNotOpen!

FPosition    ≙    FPosition₀
                ∨ FileNotOpen!

FLength      ≙    FLength₀
                ∨ FileNotOpen!

FOpen        ≙   (Open  ∧ Success)
                ∨ FileOpen!

FClose       ≙   (Close ∧ Success)
                ∨ FileNotOpen!

FAbort       ≙   (Abort ∧ Success)
                ∨ FileNotOpen!

FDelete      ≙   (Delete ∧ Success)
                ∨ FileOpen!
                ∨ FileNonExistent!
```

Note that for FDelete, if the file did not exist and had been opened, the specification allows either error message to be reported; this is an example of a non-deterministic specification.

<u>Exercise 3:</u> Define the additional errors for Rename and Copy and give final definitions of these two operations. □

Acknowledgements

The above specification is based on a specification of an indexed file system written by Ib Holm Sørensen[3]. That specification was itself derived from an earlier specification of a reliable file store written by Jean-Raymonde Abrial[1]. The above specification has also been heavily influenced by the specification of the Unix file system written by Carroll Morgan and Bernard Sufrin[2].

References

1. Abrial, J.-R. A Low Level File Handler Design. *Oxford University Programming Research Group working paper*, (1980).

2. Morgan, C. C., and Sufrin, B. A. Specification of the UNIX file system. *IEEE Transactions on Software Engineering, Vol. 10, No. 2*, (March 1984), pp. 128-142.

3. Sørensen, I. H. Specification and Design of a Filing System. *Oxford University Programming Research Group lecture notes*, (1984).

Solutions to Exercises

1.
```
Truncate_________________________
| ΔOpenFile
|______________
|
| processed' = processed ∧
| remainder' = []
|________________________________
```

2.
```
Rename_____________________________________________
| ΔFSys
| oldfn?, newfn? : Name
|______________
|
| oldfn? ∈ dom(fs) − dom(open) ∧
| newfn? ∉ dom(fs) ∪ dom(open) ∧
| fs' = {oldfn?} ⩤ fs ∪ { newfn?↦fs(oldfn?) } ∧
| open' = open
|__________________________________________________
```

```
Copy_____________________________
| ΔFSys
| oldfn?, newfn? : Name
|______________
|
| oldfn? ∈ dom(fs) − dom(open) ∧
| newfn? ∉ dom(fs) ∪ dom(open) ∧
| fs' = fs ∪ { newfn?↦fs(oldfn?) } ∧
| open' = open
|________________________________
```

```
3.   FileExists!________________________________
     |  ΞFSysError
     |__________________
     |
     |  fn? ∈ dom(fs) ∧
     |  rep! = fn? ⁀ " already exists."
     |__________________________________________


     FRename  ≙   (Rename   ∧  Success)
                ∨  FileNonExistent![oldfn?/fn?]
                ∨  FileOpen![oldfn?/fn?]
                ∨  FileExists![newfn?/fn?]
                ∨  FileOpen![newfn?/fn?]

     FCopy    ≙   (Copy     ∧  Success)
                ∨  FileNonExistent![oldfn?/fn?]
                ∨  FileOpen![oldfn?/fn?]
                ∨  FileExists![newfn?/fn?]
                ∨  FileOpen![newfn?/fn?]
```

Flexitime Specification

Abstract

This paper gives a simplified specification of an actual flexitime system. It is interesting for a number of reasons. It is quite brief and not all that complicated, and gives some good examples of the power of using set theory for specification. The specification makes use of a state which is far richer than that necessary for an implementation; this approach has its rewards in an overall simplification of the specification. The specification is also simplified by using an absolute time frame rather than one using times only within the current pay period.

Flexitime allows people to vary the hours they attend work (within certain bounds) provided they work the required total number of hours within each pay period. Keeping track of the time worked for each employee can be computerised by having employees clock in when they begin work for the day and clock out when they leave.

State

We will only record workiug time to the nearest minute.

$\mathsf{Time} \mathrel{\widehat{=}} \mathbb{N}$ -- iu minutes

A period of time can be represented by a set of (not necessarily contiguous) minutes.

$\mathsf{Period} \mathrel{\widehat{=}} \mathbb{P}\ \mathsf{Time}$

We can represent the standard working times for a pay period by a set containing all the minutes between 9am and 5pm, excluding the lunch break from 12noon to 1pm, for all the days in the pay period. In a similar way we can represent the range of permissible flexitime working hours by a set of times. The function Standard_Hours takes a time as argument and gives the set of standard working times for the pay period emcompassing the time given as its argument. Similarly, the function Flexitime_Hours gives the set of flexitimes in the period encompassing the time given as an argument.

Our model of the system will record the times worked for all the employees, plus the time at which people currently working clocked in. Each employee is assigned a unique identifier from the set Id.

```
┌─Flexi──────────────────────────────────
│ Standard_Hours,
│ Flexitime_Hours : Time → Period
│ worked : Id ⇸ Period
│ in     : Id ⇸ Time
├──────────────────
│ dom(in) ⊆ dom(worked)
└────────────────────────────────────────
```

Operations

Each operation transforms a state before (Flexi) to a state after (Flexi′).

$$\Delta Flexi \ \widehat{=} \ Flexi \wedge Flexi'$$

Some operations do not change the state.

$$\Xi Flexi \ \widehat{=} \ [\ \Delta Flexi \mid Flexi' = Flexi\ ]$$

Clocking in and out operations performed by employees involve them inserting their unique (card) key into a special terminal which transmits the employees identifier and the current time to the system. The system responds with an indicator of the operation performed. The common part of the clocking operations is given by

$$\begin{array}{|l}
\multicolumn{1}{l}{\Delta Clocking} \\
\hline
\Delta Flexi \\
id? : Id \\
t? : Time \\
ind! : Response \\
\hline
id? \in dom(worked) \wedge \\
Standard\_Hours' = Standard\_Hours \wedge \\
Flexitime\_Hours' = Flexitime\_Hours \\
\hline
\end{array}$$

where $Response \ \widehat{=} \ \{$ "ClockIn", "ClockOut", "ReadOut", "Unknown" $\}$. The identity of the employee must be known. Clocking operations do not effect Standard_Hours or Flexitime_Hours.

An employee clocking in is given by

```
ClockIn₀
  ΔClocking
  ─────────────
  id? ∉ dom(in) ∧
  t? ∈ Flexitime_Hours(t?) ∧
  in′ = in ∪ { id? ↦ t? } ∧
  worked′ = worked ∧
  ind! = "ClockIn"
```

The employee must not have clocked in already and the current time must be in the bonnds of the flexitime working hours for the current pay period. The employee is clocked in at the given time.

An employee clocking out is given by

```
ClockOut₀
  ΔClocking
  ─────────────
  id? ∈ dom(in) ∧
  worked′ = worked ⊕
            { id? ↦ (worked(id?) ∪ in(id?)..(t?-1)) } ∧
  in′ = { id? } ⩤ in ∧
  ind! = "ClockedOut"
```

The employee must have clocked in. The minutes worked since clocking in are credited to the employees time worked. Only the period that lies within flexitime hours really counts towards flexitime but we have chosen to record the total working time in this specification in order to simplify it and allow extensions to keep track of overtime worked etc. The minutes worked are all those minutes from the time the employee clocked in (although he may not have worked the whole of that minute) upto but not including the minute in which he clocks out (even though he has worked part of that minnte). On average partial minutes not worked at clock in should cancel out partial minutes worked at clock out.

On each transaction the system responds with the current credit or debit of time worked by the employee within the current pay period, relative to the standard times.

```
┌─Worked────────────────────────────────────────────
│ ΔClocking
│ cr!   : RelMinutes
├──────────────────────
│ cr!  = |worked′(id?) ∩ Flexitime_Hours(t?)|
│      − |{ t : Standard_Hours(t?) | t < t? }|
└───────────────────────────────────────────────────
```

where Relminutes $\widehat{=}$ $\mathbb{Z}$. The credit (cr!) is of type RelMinutes (relative minutes) which is positive to indicate a credit and negative to indicate a debit. Only the period of time worked that is within the flexitime hours for the current pay period counts.

The clocking operations in full are

$$\mathsf{ClockIn} \;\widehat{=}\; \mathsf{ClockIn_0} \wedge \mathsf{Worked}$$

$$\mathsf{ClockOut} \;\widehat{=}\; \mathsf{ClockOut_0} \wedge \mathsf{Worked}$$

If an employee not currently working inserts a card outside flexitime hours they will not be clocked in. However, they will receive an indication of the current time credit.

```
┌─ReadOut──────────────────────────────────
│ Worked
├──────────────────────
│ id? ∉ dom(in) ∧
│ t? ∉ Flexitime_Hours(t?) ∧
│ ind! = “ReadOut” ∧
│ Flexi′ = Flexi
└──────────────────────────────────────────
```

If an unknown key is inserted an error respouse is given

```
┌─Unknown!──────────────────────────────
│ ΞFlexi
│ id?  : Id
│ ind! : Response
├──────────────────
│ id ∉ dom(worked) ∧
│ ind! = "Unknown"
└───────────────────────────────────────
```

An administrative operation is required to add a new employee. The identity of new employee is chosen from those not already in use.

```
┌─Add_Employee──────────────────────────
│ ΔFlexi
│ id! : Id
├──────────────────
│ id! ∉ dom(worked) ∧
│ worked' = worked ∪ { id! ↦ {} } ∧
│ in' = in ∧
│ Standard_Hours' = Standard_Hours ∧
│ Flexitime_Hours' = Flexitime_Hours
└───────────────────────────────────────
```

Acknowledgement

This paper specifies a simplified version of a problem specified by Jolanta Imbert of the GEC Research Laboratories, Marconi Research Centre.

Diary Specification

A diary records details of events. It records when an event will take place and what the event is. We will model an event by

```
┌── Event ─────────────────────────────┐
│ when : Period
│ what : Information
└──────────────────────────────────────┘
```

where Period $\mathrel{\widehat=}$ $\mathbb{P}$ Time. A period is a, not necessarily contiguous, set of times. We will not specify the structure of Information here; it can be thought of as just text.

A diary consists of a set of events.

$$\text{Diary} \mathrel{\widehat=} \mathbb{P}\ \text{Event}$$

We will not put any constraints on the entries in a diary as we view a diary purely as a mechanism for recording information about events.

Initially a diary is empty.

$$\text{Diary}_{\text{INIT}} \mathrel{\widehat=} \{\}$$

Operations

Each operation on a diary transforms a state before (Diary) into a state after (Diary′).

$$\Delta\text{Diary} \mathrel{\widehat=} [\ \text{entries}, \text{entries}' : \text{Diary}\ ]$$

Some operations leave the diary unchanged.

$$\Xi\text{Diary} \mathrel{\widehat=} [\ \Delta\text{Diary} \mid \text{entries}' = \text{entries}\ ]$$

Two primitive operations on diaries are adding and deleting entries.

Add_{EV}

ΔDiary
Event?

entries′ = entries ∪ { Event? }

Delete_{EV}

ΔDiary
Event?

entries′ = entries − { Event? }

When querying a diary it is useful to extract all the events that overlap a given time period.

```
Query_Period
ΞDiary
at? : Period
entries! : Diary
------------------------------
entries! = { ev : entries | at? ∩ ev.when ≠ {} }
```

The above operation allows the input of any set of times as the period at?. In a realistic diary system there would be a small language that is used for specifying time periods; such a language can be specified independently of the above and the time period associated with an input string can be input to the Query_Period operation.

Another useful querying operation is to select all the events which match some criterion.

```
┌─ Query_Match ──────────────────────────────────
│ ≡Diary
│ match? : Event → Boolean
│ entries! : Diary
├──────────────────────
│ entries! = { ev : entries | match(ev) }
└────────────────────────────────────────────────
```

In a realistic diary system the function match?, which is an input to the above operation, would be specified according to a small input language which allowed such operations as pattern matching against the information field of an event and tests on the time period of an event.

Multiple Diaries

So far we have only a single diary. The system should be able to maintain many diaries: one for each person plus others for such entities as groups of people, rooms, etc.

```
Diaries  ≙  Entity ⇸ Diary
```

where Entity is the set of all possible entity names.

Initially there are no diaries in the system.

$\text{Diaries}_{INIT} \mathrel{\widehat{=}} \{\}$

The operations on individual diaries may be promoted to the multiple diary state by

```
┌─ Multiple ──────────────────────────────
│ ΔDiaries
│ e? : Entity
│ ΔDiary
├──────────────
│ e? ∈ dom(diaries) ∧
│ entries = diaries(e?) ∧
│ diaries′ = diaries ⊕ { e?↦entries′ }
└─────────────────────────────────────────
```

where $\Delta Diaries \mathrel{\widehat{=}} [\ diaries,\ diaries' : Diaries\ ]$. The promoted operationsare

$$Add_M \mathrel{\widehat{=}} (Multiple \land Add_{EV}) \setminus \Delta Diary$$

$$Delete_M \mathrel{\widehat{=}} (Multiple \land Delete_{EV}) \setminus \Delta Diary$$

$$Query\_Period_M \mathrel{\widehat{=}} (Multiple \land Query\_Period) \setminus \Delta Diary$$

$$Query\_Match_M \mathrel{\widehat{=}} (Multiple \land Query\_Match) \setminus \Delta Diary$$

The diaries of a set of entities may be combined into a single diary.

```
┌─ Combine ───────────────────────────────────────────────
│ ΞDiaries
│ se? : ℙ Entity
│ entries! : Diary
│ missing! : ℙ Entity
├──────────────
│ missing! = se? − dom(diaries) ∧
│ entries! = ⋃ { e: (se? ∩ dom(diaries)) • diaries(e) }
└─────────────────────────────────────────────────────────
```

where $\Xi Diaries \mathrel{\widehat{=}} [\ \Delta Diaries \mid diaries' = diaries\ ]$. The diaries of all those entities in the set se? that exist are combined into a single diary. The entities in se? that do not have diaries are reported in the set missing!.

Groups of Entities

So far we have a single diary associated with an entity. An entity would have its own individual diary but would also be interested in the diaries of all the groups it is a member of. We can introduce a relation between entities representing group membership.

$$\mathsf{Groups} \mathrel{\widehat{=}} \mathsf{Entity} \leftrightarrow \mathsf{Entity}$$

Given a relation $\mathsf{isingrp} : \mathsf{Groups}$ and an entity p, the groups of which p is a member are those contained in the set

$$\mathsf{isingrp}(\!|\ \{\ p\ \}\ |\!).$$

Because the groups are themselves entities we can form supergroups from a number of groups, etc. For example, the group of people employed in a division consist of the groups of people in the departments of the division.

An entity p will be interested in its own diary, the diaries of the groups it is a member of, the diaries of the supergroups those groups are members of, and so on. In mathematics we can use the (reflexive) transitive closure of the relation $\mathsf{isingrp}$, which we denote by $\mathsf{isingrp}^*$, to provide a new relation: an entity is related to all the entities whose diaries it is interested in.

```
┌── Select_Entities ──────────────────
│ ΞGroups
│ e? : Entity
│ se! : ℙ Entity
├───────────────
│ se! isingrp*⦇{p}⦈
└─────────────────────────────────────
```

where $\Xi\mathsf{Groups} \mathrel{\widehat{=}} [\ \mathsf{isingrp},\ \mathsf{isingrp}' : \mathsf{Groups} \mid \mathsf{isingrp}' = \mathsf{isingrp}\ ].$

Note: if R is a relation of the form

$$R : X \leftrightarrow X$$

then its reflexive transitive closure R^* is the smallest relation with the followiug properties for all $x, y, z : X$

$$x\ R^*\ x$$
$$x\ R\ y \Rightarrow x\ R^*\ y$$
$$x\ R^*\ y \wedge y\ R^*\ z \Rightarrow x\ R^*\ z$$

To extract the combined diary that an entity is interested in we use

$$\mathrm{Combined\_Diary} \mathrel{\widehat{=}} \mathrm{Select\_Entities} \gg \mathrm{Combine}$$

The set of entities selected by Select_Entities are input to the Combine operation which combines the diaries of the entities that exist into a single diary; any of the entities that do not have diaries are reported in the set missing!.

End note: The above specification only provides a basis for a diary system. Such administrative operations as setting up new entities, removing entities, and setting up and modifying the group membership relation are not covered.

Z Reference Card
Mathematical Notation
Version 2.1

Programming Research Group
Oxford Uuiversity

1. Definitions and declarations.

Let x, x_k be identifiers and T, T_k sets.

$LHS \mathrel{\widehat=} RHS$	Definition of LHS as syntactically equivalent to RHS.
$x : T$	Declaration of x as type T.
$x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n$	List of declarations.
$x_1,\ x_2,\ \ldots\ ,\ x_n\ :\ T$	$\mathrel{\widehat=} x_1 : T;\ x_2 : T;\ \ldots\ ;\ x_n : T$.

2. Logic.

Let P, Q be predicates and D declarations.

$\neg\ P$	Negation: "not P".
$P \land Q$	Conjunction: "P and Q".
$P \lor Q$	Disjunction: "P or Q".
$P \Rightarrow Q$	Implication: "P implies Q" or "if P then Q".
$P \Leftrightarrow Q$	Equivalence: "P is logically equivalent to Q".
$\forall\ x : T \bullet P$	Uuiversal quantification: "for all x of type T, P holds".
$\exists\ x : T \bullet P$	Existential quantification: "there exists an x of type T such that P".
$\exists_1\ x : T \bullet P_x$	Unique existence: "there exists a unique x of type T such that P". $\mathrel{\widehat=} (\exists\ x : T \bullet P_x \land \neg(\exists y : T \mid y \neq x \bullet P_y))$
$\forall\ x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n \bullet P$	"For all x_1 of type T_1, x_2 of type T_2, . . . , and x_n of type T_n, P holds.
$\exists\ x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n \bullet P$	Similar to $\forall$.
$\exists_1\ x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n \bullet P$	Similar to $\forall$.
$\forall\ D \mid P \bullet Q$	$\mathrel{\widehat=} (\forall\ D \bullet P \Rightarrow Q)$.
$\exists\ D \mid P \bullet Q$	$\mathrel{\widehat=} (\exists\ D \bullet P \land Q)$.
$t_1 = t_2$	Equality between terms.
$t_1 \neq t_2$	$\mathrel{\widehat=} \neg(t_1 = t_2)$.

3. Sets.

Let S, T and X be sets; t, t_k terms; P a predicate and D declarations.

$t \in S$	Set membership: "t is an element of S".
$t \notin S$	$\mathrel{\widehat=} \neg(t \in S)$.
$S \subseteq T$	Set inclusion: $\mathrel{\widehat=} (\forall\ x : S \bullet x \in T)$.
$S \subset T$	Strict set inclusion: $\mathrel{\widehat=} S \subseteq T \land S \neq T$.
$\{\}$	The empty set.
$\{\ t_1,\ t_2,\ \ldots\ ,\ t_n\ \}$	The set containing $t_1, t_2, \ldots$ and t_n.
$\{\ x : T \mid P\ \}$	The set containing exactly those x of type T for which P holds.
$(t_1,\ t_2,\ \ldots\ ,\ t_n)$	Ordered n-tuple of $t_1, t_2, \ldots$ and t_n.
$T_1 \times T_2 \times \ldots \times T_n$	Cartesian product: the set of all n-tuples such that the kth component is of type T_k.
$\{\ x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n \mid P\ \}$	The set of n-tuples $(x_1,\ x_2,\ \ldots\ ,\ x_n)$ with each x_k of type T_k such that P holds.

$\{ D \mid P \bullet t \}$	The set of t's such that given the declarations D, P holds.
$\{ D \bullet t \}$	$\widehat{=}$ $\{ D \mid true \bullet t \}$.
$\mathbb{P}\ S$	Powerset: the set of all subsets of S.
$\mathbb{F}\ S$	Set of finite subsets of S: $\widehat{=}$ $\{ T : \mathbb{P}\ S \mid T$ is finite $\}$.
$S \cap T$	Set intersection: given $S, T : \mathbb{P}\ X$, $\widehat{=}$ $\{ x : X \mid x \in S \wedge x \in T \}$..
$S \cup T$	Set union: given $S, T : \mathbb{P}\ X$, $\widehat{=}$ $\{ x : X \mid x \in S \vee x \in T \}$.
$S - T$	Set difference: given $S, T : \mathbb{P}\ X$, $\widehat{=}$ $\{ x : X \mid x \in S \wedge x \notin T \}$.
$\bigcap SS$	Distributed set intersection: given $SS : \mathbb{P}\ (\mathbb{P}\ X)$, $\widehat{=}$ $\{x : X \mid (\forall S : SS \bullet x \in S)\}$.
$\bigcup SS$	Distributed set union: given $SS : \mathbb{P}\ (\mathbb{P}\ X)$, $\widehat{=}$ $\{x : X \mid (\exists S : SS \bullet x \in S)\}$.
$\lvert S \rvert$	Size (number of distinct elements) of a finite set.

4. Numbers.

$\mathbb{N}$	The set of natural numbers (non-negative integers).
$\mathbb{N}^*$	The set of strictly positive natural numbers: $\widehat{=}$ $\mathbb{N} - \{ 0 \}$.
$\mathbb{Z}$	The set of integers (positive, zero and negative).
$m\,..\,n$	The set of integers between m and n inclusive: $\widehat{=}$ $\{ k : \mathbb{Z} \mid m \leq k \wedge k \leq n \}$.
$min\ S$	Minimum of a set, $S : \mathbb{F}\ \mathbb{N}$. $min\ S \in S \wedge (\forall x : S \bullet x \geq min\ S)$.
$max\ S$	Maximum of a set, $S : \mathbb{F}\ \mathbb{N}$. $max\ S \in S \wedge (\forall x : S \bullet x \leq max\ S)$.

5. Relations.

A relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations.

Let X, Y, and Z be sets; $x : X$; $y : Y$; and $R : X \leftrightarrow Y$.

$X \leftrightarrow Y$	The set of relations from X to Y: $\widehat{=}$ $\mathbb{P}\ (X \times Y)$.
$x\ R\ y$	x is related by R to y: $\widehat{=}$ $(x, y) \in R$.
$x \mapsto y$	$\widehat{=}$ (x, y)
$\{ x_1 \mapsto y_1,\ x_2 \mapsto y_2,\ \ldots\ ,\ x_n \mapsto y_n \}$	The relation $\{ (x_1, y_1), \ \ldots\ , (x_n, y_n) \}$ relating x_1 to y_1, ..., and x_n to y_n.
$dom\ R$	The domain of a relation: $\widehat{=}$ $\{x : X \mid (\exists y : Y \bullet x\ R\ y)\}$.
$rng\ R$	The range of a relation: $\widehat{=}$ $\{y : Y \mid (\exists x : X \bullet x\ R\ y)\}$.
$R_1 \mathbin{;} R_2$	Forward relational composition: given $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$, $\widehat{=}$ $\{ x : X;\ z : Z \mid (\exists y : Y \bullet x\ R_1\ y \wedge y\ R_2\ z\ )\}$.
$R_1 \circ R_2$	Relational composition: $\widehat{=}$ $R_2 \mathbin{;} R_1$.
R^{-1}	Inverse of relation R: $\widehat{=}$ $\{ y : Y;\ x : X \mid x\ R\ y \}$.
$id\ X$	Identity function on the set X: $\widehat{=}$ $\{ x : X \bullet x \mapsto x \}$.
R^k	The relation R composed with itself k times: given $R : X \leftrightarrow X$, $R^0 \mathrel{\widehat{=}} id\ X$, $R^{k+1} \mathrel{\widehat{=}} R^k \circ R$.
R^*	Reflexive transitive closure: $\widehat{=}$ $\bigcup \{ n : \mathbb{N} \bullet R^n \}$.
R^+	Non-reflexive transitive closure: $\widehat{=}$ $\bigcup \{ n : \mathbb{N}^+ \bullet R^n \}$.
$R(\!\lvert S \rvert\!)$	Image: given $S : \mathbb{P}\ X$, $\widehat{=}$ $\{y : Y \mid (\exists x : S \bullet x\ R\ y)\}$.

$S \lhd R$	Domain restriction to S: given $S: \mathbb{P}\ X$, $\widehat{=} \{x:X;y:Y \mid x \in S \wedge x\ R\ y\}$.
$S ⩤ R$	Domain subtraction: given $S: \mathbb{P}\ X$, $\widehat{=} (X - S) \lhd R$.
$R \rhd T$	Range restriction to T: given $T: \mathbb{P}\ Y$, $\widehat{=} \{x:X;y:Y \mid x\ R\ y \wedge y \in T\}$.
$R ⩥ T$	Range subtraction of T: given $T: \mathbb{P}\ Y$, $\widehat{=} R \rhd (Y - T)$.
$R_1 \oplus R_2$	Overriding: given $R_1, R_2 : X \leftrightarrow Y$, $\widehat{=} (\mathrm{dom}\ R_2 ⩤ R_1) \cup R_2$.

6. Functions.

A function is a relation with the property that for each element in its domain there is a uniqne element in its range related to it. As functions are relations all the operators defined above for relations also apply to functions.

$X \nrightarrow Y$	The set of partial functions from X to Y: $\widehat{=} \{ f: X \leftrightarrow Y \mid$ $(\forall x: \mathrm{dom}\ f \bullet$ $(\exists ! y: Y \bullet x\ f\ y)) \}$.
$X \rightarrow Y$	The set of total functions from X to Y: $\widehat{=} \{ f: X \nrightarrow Y \mid \mathrm{dom}\ f = X \}$.
$X ⤔ Y$	The set of one-to-one partial functions from X to Y: $\widehat{=} \{ f: X \nrightarrow Y \mid$ $(\forall y: \mathrm{rng}\ f \bullet$ $(\exists ! x: X \bullet x\ f\ y)) \}$.
$X \rightarrowtail Y$	The set of one-to-one total functions from X to Y: $\widehat{=} \{ f: X ⤔ Y \mid \mathrm{dom}\ f = X \}$.
$f\ t$	The function f applied to t.
$(\lambda\ x : X \mid P \bullet t)$	Lambda-abstraction: the function that given an argument x of type X such that P holds the result is t. $\widehat{=} \{ x: X \mid P \bullet x \mapsto t \}$.
$(\lambda\ x_1: T_1; \ldots ; x_n: T_n \mid P \bullet t)$	$\widehat{=} \{x_1:T_1; \ldots ; x_n:T_n \mid P \bullet$ $(x_1, \ldots , x_n) \mapsto t \}$.

7. Orders.

partial_order X	The set of partial orders on X. $\widehat{=} \{ R: X \leftrightarrow X \mid \forall x, y, z: X \bullet$ $x\ R\ x \wedge$ $x\ R\ y \wedge y\ R\ x \Rightarrow x=y \wedge$ $x\ R\ y \wedge y\ R\ z \Rightarrow x\ R\ z$ $\}$.
total_order X	The set of total orders on X. $\widehat{=} \{ R: \mathrm{partial\_order}\ X \mid$ $\forall x, y: X \bullet$ $x\ R\ y \vee y\ R\ x$ $\}$.
monotonic X $<_X$	The set of functions from X to X that are monotonic with respect to the order $<_X$ on X. $\widehat{=} \{ f : X \nrightarrow X \mid$ $x <_X y \Rightarrow f(x) <_X f(y)$ $\}$.

8. Sequences.

$\mathrm{seq}\ X$	The set of sequences whose elements are drawn from X: $\widehat{=}\ \{A : \mathbb{N}^{+} \nrightarrow X \mid \mathrm{dom}\ A = 1..\lvert A\rvert\ \}$.
$\lvert A\rvert$	The length of sequence A.
$[\,]$	The empty sequence $\{\}$.
$[a_1, \ldots, a_n]$	$\widehat{=}\ \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$.
$[a_1, \ldots, a_n] ^\frown [b_1, \ldots, b_m]$	Concatenation: $\widehat{=}\ [a_1, \ldots, a_n, b_1, \ldots, b_m]$, $[\,] ^\frown A = A ^\frown [\,] = A$.
$\mathrm{head}\ A$	$\widehat{=}\ A(1)$.
$\mathrm{last}\ A$	$\widehat{=}\ A(\lvert A\rvert)$.
$\mathrm{tail}\ [x] ^\frown A$	$\widehat{=}\ A$.
$\mathrm{front}\ A ^\frown [x]$	$\widehat{=}\ A$.
$\mathrm{rev}\ [a_1, a_2, \ldots, a_n]$	Reverse: $\widehat{=}\ [a_n, \ldots, a_2, a_1]$, $\mathrm{rev}\ [\,] = [\,]$.
$^\frown/AA$	Distributed concatenation: given $AA : \mathrm{seq}(\mathrm{seq}(X))$, $\widehat{=}\ AA(1) ^\frown \ldots ^\frown AA(\lvert AA\rvert)$, $^\frown/[\,] = [\,]$.
$⨾/AR$	Distributed relational composition: given $AR : \mathrm{seq}\ (X \leftrightarrow X)$, $\widehat{=}\ AR(1) ⨾ \ldots ⨾ AR(\lvert AR\rvert)$, $⨾/[\,] = \mathrm{id}\ X$.
<u>disjoint</u> AS	Pairwise disjoint: given $AS : \mathrm{seq}\ (\mathbb{P}\ X)$, $\widehat{=}\ (\forall\ i, j : \mathrm{dom}\ AS \bullet i \neq j$ $\Rightarrow AS(i) \cap AS(j) = \{\})$.
AS <u>partitions</u> S	$\widehat{=}\ \mathrm{disjoint}\ AS$ $\wedge \bigcup \mathrm{ran}\ AS = S$.
A <u>in</u> B	Contiguous subsequence: $\widehat{=}\ (\exists C, D : \mathrm{seq}\ X \bullet$ $C ^\frown A ^\frown D = B)$.
$\mathrm{squash}\ f$	Convert a function, $f : \mathbb{N} \nrightarrow X$, into a sequence by squashing its domain. $\mathrm{squash}\ \{\} = [\,]$, and if $f \neq \{\}$ then $\mathrm{squash}\ f =$ $[f(i)] ^\frown \mathrm{squash}(\{i\} ⩤ f)$ where $i = \min(\mathrm{dom}\ f)$ e.g. $\mathrm{squash}\ \{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\}$ $= [A, B, C]$
$S \upharpoonleft A$	Restrict the sequeuce A to those items whose index is in the set S: $\widehat{=}\ \mathrm{squash}(S \lhd A)$
$A \upharpoonright T$	Restrict the range of the sequence A to the set T: $\widehat{=}\ \mathrm{squash}(A \rhd T)$.

9. Bags.

$\mathrm{bag}\ X$	The set of bags whose elements are drawn from X: $\widehat{=}\ X \nrightarrow \mathbb{N}^{+}$ A bag is represented by a fuuction that maps each element in the bag onto its frequency of occurrence in the bag.
$[\![\,]\!]$	The empty bag $\{\}$.
$[\![x_1, x_2, \ldots, x_n]\!]$	The bag containing $x_1, x_2, \ldots$ and x_n with the frequency they occur in the list.
$\mathrm{items}\ s$	The bag of items contained in the sequence s: $\widehat{=}\ \{ x : \mathrm{rng}\ s \bullet$ $x \mapsto \lvert\{i : \mathrm{dom}\ s \mid s(i) = x\}\rvert$ $\}$

Z Reference Card
Schema Notation

[For details see "Schemas in Z"]

Programming Research Group
Oxford University

Schema definition: a schema groups together some declarations of variables and a predicate relating these variables. There are two ways of writing schemas: vertically, for example

```
S_____________________
| x : ℕ
| y : seq ℕ
|__________________
| x ≤ |y|
|_____________________
```

or horizontally, for the same example

S ≙ [x: ℕ; y: seq ℕ | x≤|y|].

Use in signatures after ∀, λ, {...}, etc.:

(∀S • y ≠ [ ]) ≙ (∀x:ℕ; y: seq ℕ | x≤|y| • y≠[ ]).

tuple S — The tuple formed of a schema's variables.

pred S — The predicate part of a schema: e.g. pred S is x ≤ |y|.

Inclusion — A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must be the same type) and the predicates of S and T are conjoined. e.g.

```
T_____________________
| S
| z : ℕ
|__________________
| z < x
|_____________________
```

is

```
______________________
| x, z : ℕ
| y : seq ℕ
|__________________
| x ≤ |y| ∧ z < x
|_____________________
```

S | P — The schema S with P conjoined to its predicate part. e.g. (S | x>0) is [x:ℕ;y:seq ℕ | x≤|y|∧x>0].

S ; D — The schema S with the declarations D merged with the declarations of S. e.g. (S ; z : ℕ) is [x, z:ℕ; y:seq ℕ | x≤|y|]

S[new/old] — Renaming of components: the schema S with the component old renamed to new in its declaration and every free use of that old within the predicate. e.g. S[z/x] is [z:ℕ; y:seq ℕ | z ≤ |y|] and S[y/x, x/y] is [y:ℕ; x:seq ℕ | y ≤ |x|]

Decoration — Decoration with subscript, superscript, prime, etc.; systematic renaming of the variables declared in the schema. e.g. S′ is [x′:ℕ; y′:seq ℕ | x′≤|y′|]

¬S — The schema S with its predicate part negated. e.g. ¬S is [x:ℕ; y:seq ℕ | ¬(x≤|y|)]

S ∧ T — The schema formed from schemas S and T by merging their declarations (see inclusion above) and and'ing their predicates. Given T ≙ [x: ℕ; z: ℙ ℕ | x∈z], S ∧ T is

```
x : ℕ
y : seq ℕ
z : ℙ ℕ
──────────────
x ≤ |y| ∧ x ∈ z
```

$S \lor T$ The schema formed from schemas S and T by merging their declarations and or'ing their predicates. e.g. $S \lor T$ is

```
x : ℕ
y : seq ℕ
z : ℙ ℕ
──────────────
x ≤ |y| ∨ x ∈ z
```

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking pred $S \Rightarrow$ pred T as the predicate. e.g. $S \Rightarrow T$ is similar to $S \land T$ and $S \lor T$ except the predicate contains an "$\Rightarrow$" rather than an "$\land$" or an "$\lor$".

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking pred $S \Leftrightarrow$ pred T as the predicate. e.g. $S \Leftrightarrow T$ the same as $S \land T$ with "$\Leftrightarrow$" in place of the "$\land$".

$S \setminus (v_1, v_2, \ldots, v_n)$
Hiding: the schema S with the variables $v_1, v_2, \ldots,$ and v_n hidden: the variables listed are removed from the declarations and are existentially quantified in the predicate. e.g. $S \setminus x$ is $[y{:}\,\mathrm{seq}\ \mathbb{N} \mid (\exists x{:}\mathbb{N} \cdot x \le |y|)]$

A schema may be specified instead of a list of variables; in this case the variables declared in that schema are hidden.
e.g. $(S \land T) \setminus S$ is

```
z : ℙ ℕ
──────────────
(∃ x: ℕ; y: seq ℕ •
   x ≤ |y| ∧ x ∈ z)
```

$S \upharpoonright (v_1, v_2, \ldots, v_n)$.
Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden: the variables removed from the declarations are existentially quantified in the predicate.
e.g. $(S \land T) \upharpoonright (x, y)$ is

```
x : ℕ
y : seq ℕ
──────────────
(∃ z : ℙ ℕ •
   x ≤ |y| ∧ x ∈ z)
```

The list of variables may be replaced by a schema as for hiding; the variables declared in the schema are used for the projection.

The following conventions are used for variable names in those schemas which represent operations:

undashed	state before the operation,
dashed	state after the operation,
ending in "?"	inputs to the operation, and
ending in "!"	outputs from the operation.

The following schema operations only apply to schemas following the above conventions.

pre S — Precondition: all the state after components (dashed) and the outputs (ending in "!") are hidden. e.g. given

```
S________________________
| x?, s, s′, y! : ℕ
|_______________
| s′ = s − x? ∧ y! = s
|________________________
```

pre S is

```
_________________________
| x?, s : ℕ
|_______________
| (∃ s′, y! : ℕ •
|    s′ = s−x? ∧ y! = s)
|________________________
```

post S — Postcondition: this is similar to precondition except all the state before components (undashed) and inputs (ending in "?") are hidden.

S ⊕ T — Overriding:
≙ (S ∧ ¬pre T) ∨ T.
e.g. given S above and

```
T________________________
| x?, s, s′ : ℕ
|_______________
| s < x? ∧ s′ = s
|________________________
```

S ⊕ T is

```
_________________________
| x?, s, s′, y! : ℕ
|_______________
| (s′ = s−x? ∧ y! = s ∧
|   ¬(∃ s′ : ℕ •
|       s < x? ∧ s′ = s))
| ∨ (s < x? ∧ s′ = s)
|________________________
```

The predicate can be simplified:

```
_________________________
| x?, s, s′, y! : ℕ
|_______________
| (s′ = s−x? ∧ y! = s
|    ∧ s ≥ x?)
| ∨
|  (s < x? ∧ s′ = s)
|________________________
```

S ⨾ T — Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the state after components of S and the state before components of T that have a basename* in common, rename both to new variables, take the schema "and" (∧) of the resulting schemas, and hide the new variables.
e.g. S ⨾ T is

```
_________________________
| x?, s, s′, y! : ℕ
|_______________
| (∃ s₀ : ℕ •
|    s₀ = s−x? ∧ y! = s ∧
|    s₀ < x? ∧ s′ = s₀)
|________________________
```

* basename is the name with any decoration ("′", "!", "?",etc.) removed.

S >> T	Piping this schema operation is similar to schema composition; the difference is that rather than identifying the state after components of S with the state before components of T, the output components of S (ending in "!") are identified with the input components of T (ending in "?") that have the same basename.