

# CSE4213 Lecture Notes

Revision of Logic, Substitutions

A Simple File System development

Ken Robinson

20050414 / Lecture 12

## 1 Objectives

### Objectives of this lecture

To explore the specification of a simple file system.

In order to present that specification we will have to visit the set model of sequences.

This development is based on a similar Z case-study by Ian Hayes and Ib Holme Sørensen.

## 2 A Simple File Model

### 2.1 Some Basics

#### The Single Anonymous file

We wish to model a single anonymous file.

We will model the type *FILE* as a sequence of some basic unit that we will call *BYTE*.

*BYTE* is specified as a deferred set in a type machine, *ByteTYPE*.<sup>1</sup>

*FILE* is specified as an abbreviation in another type machine, *FileTYPE*.

j++i

```
MACHINE    ByteTYPE
SETS      BYTE
END
```

The *FileTYPE* machine *SEES* *ByteTYPE*.

```
MACHINE    FileTYPE
SEES      ByteTYPE
DEFINITIONS  FILE == seq(BYTE)
END
```

j++i

---

<sup>1</sup>In earlier case studies we have usually used machine parameters to identify external sets. This is not an appropriate method for identifying global sets (or constants), that is sets that many developments may wish to reference. A common name for a machine parameter of two machines does not guarantee that the machines will be referencing the same set; that depends on how each machine is instantiated. Referencing the same machine does guarantee common sets. For other examples, see *Bool TYPE* and *Scalar TYPE* that are part of the *BToolkit* library.



### A Read operation

We wish to model an operation that attempts to read *length* bytes –a sequence of *BYTE* of size *length*– from the *remainder* of the file.

If the size of *remainder* is less than *length* then we should simply return all of *remainder*.

```
bytes <-- Read(length) =
  PRE
    length : NAT
  THEN
    bytes := remainder /\ length ||
    processed := processed ^
      (remainder /\ length) ||
    remainder := remainder \\/ length
  END
```

*i++i*

### A Write operation

We want to model an operation that will write a file, *bytes*, over the prefix of the remainder of the file, and move the position to follow the newly written data.

Unlike the read operation, this operation potentially changes all three of *contents*, *processed* and *remainder*.

```
Write(bytes) =
  PRE
    bytes : FILE
  THEN
    contents := processed ^ (remainder <+ bytes) ||
    processed := processed ^ bytes ||
    remainder := remainder \\/ (size(bytes))
  END
```

## 2.3 Some Theory

*i++i*

### Relational override

*ASCII*  $r_1 <+ r_2$

*Publication*  $r_1 \triangleleft r_2$

$r_1 \triangleleft r_2 = r_2 \cup (\text{dom}(r_2) \triangleleft r_1)$

$r_1 \triangleleft r_2$  behaves like  $r_2$  everywhere in the domain of  $r_2$ , and like  $r_1$  everywhere else.

Notice that although override is defined on relations, the operator maintains functions and sequences.

That is, if  $f_1$  and  $f_2$  are functions, then  $f_1 \triangleleft f_2$  is a function,

if  $s_1$  and  $s_2$  are sequences, then  $s_1 \triangleleft s_2$  is a sequence.

## 2.4 More Operations

*i++i*

### A Reposition operation

In order to be able to carry out random access operations, we need an operation that moves the current position to any point with the contents of the file.

Positioning at 0 puts the current position before the first element in the file, and the maximum valid position will be at  $size(contents)$

```
Reposition(pos) =
  PRE
    pos : NAT & pos <= size(contents)
  THEN
    processed := contents /\ pos ||
    remainder := contents \\/ pos
  END
```

j++i

### Position and Length enquiry operations

It will be useful to have an enquiry operation, *Position*, that will return the current position in the file.

```
pos <-- Position =
  BEGIN
    pos := size(processed)
  END
```

and a *Length* operation that returns the length of the file

```
length <-- Length =
  BEGIN
    length := size(contents)
  END
```

## 3 Multiple File System

### A File System

We now want to model a file system consisting of a collection of named files.

We will need to model a set of named files and a set of "opened" files that can be used with file operations, such as those we've already specified for the single anonymous file.

j++i

### The Set of Names

We will assume that the set of names, used for identifying files, is a set *NAME*, passed as a parameter to the machine, which we will call *FileSystem*.<sup>2</sup>

Hence the machine header is as follows:

```
MACHINE    FileSystem(NAME)
```

and the machine *SEES* the machines FileTYPE and ByteTYPE.

```
SEES      FileTYPE, ByteTYPE
```

---

<sup>2</sup>Given the comment made earlier about global sets, it is more likely that NAME should be a deferred set in a machine, rather than a parameter.

### Modelling a set of objects

We have to model a set of opened files.

In the *SingleFile* specification we modelled a single open file using three components or attributes: *contents*, *processed* and *remainder*.

Essentially, what we did there was to model a single object. We now have to model a set of objects, or if you like, a *class*.

Consider modelling a class, *CLASS*, in which each object has attributes  $attr_1, attr_2, \dots, attr_n$  of type  $X_1, X_2, \dots, X_n$ , respectively.

We use a set *CLASS* to represent all possible instances, and *class* ( $\subseteq$  *CLASS*) to represent all instantiated members of *CLASS*.

The attributes are represented by functions  $attr_1 \in class \rightarrow X_1, attr_2 \in class \rightarrow X_2, \dots, attr_n \in class \rightarrow X_n$ .

### Modelling a set of opened files

To represent all possible opened files we will use a set *OPENFILE*.

You can think of the elements of this set as being *pointers* to opened files.

We will then use a variable *openfiles* ( $\subseteq$  *OPENFILE*) to represent the actual set of opened files.

The attributes, *contents*, *processed*, and *remainder*, become functions:

$$contents \in openfiles \rightarrow FILE$$

$$processed \in openfiles \rightarrow FILE$$

$$remainder \in openfiles \rightarrow FILE$$

### The Named files

Files will be referenced by *names*, except when they are opened when they will be referenced by *file-pointers*.

We assume that each file has a different name, so we can use a function to map from *NAME* to *FILE*. We will use the variable *filesystem* to contain that mapping.

$$filesystem \in NAME \rightarrow FILE$$

We also need to keep a mapping from *openfiles* to *NAME*, and we assume each member of *openfiles* maps to a different member of *NAME*, thus this relation is a total injective function.

$$filename \in openfiles \rightarrow NAME$$

### Maintaining the constraint between components

In the *SingleFile* specification we had a constraint

$$contents = processed \frown remainder$$

We now need to do this for each opened file so we need to quantify the above constraint across all the opened files:

$$fle.(fle \in openfiles \Rightarrow contents(fle) = processed(fle) \frown remainder(fle))$$

Using New Zealand phonetic spelling of “file”.

## The Signature of FileSystem

```
MACHINE FileSystem(NAME)
SEES    FileTYPE, ByteTYPE
SETS    OPENFILE
VARIABLES
    openfiles, filesystem, filename,
    contents, processed, remainder
INVARIANT
    filesystem : NAME +-> FILE &
    openfiles  <: OPENFILE &
    filename   : openfiles >-> NAME &
    contents   : openfiles --> FILE &
    processed  : openfiles --> FILE &
    remainder  : openfiles --> FILE &
    !fle.(fle : openfiles => (
        contents(fle) = processed(fle) ^ remainder(fle)
    ))
```

## The Signature of FileSystem

```
INITIALISATION
    openfiles, filesystem, filename := {}, {}, {} ||
    contents, processed, remainder := {}, {}, {}
```

## The operations

We can promote the operations that we had for the *SingleFile* by observing that each operation must now have a file argument, and making the appropriate changes to accommodate the fact that *contents*, *processed* and *remainder* are now functions.

## Read and Write

```
bytes <-- Read(file,length) =
    PRE file : openfiles & length : NAT
    THEN bytes := remainder(file) /\ length ||
        remainder(file) :=
            remainder(file) \\/ length ||
        processed(file) := processed(file) ^
            (remainder(file) /\ length)
    END;
Write(file,bytes) =
    PRE file : openfiles & bytes : FILE
    THEN contents(file) := processed(file) ^
        (remainder(file)<+ bytes) ||
        processed(file) := processed(file) ^ bytes ||
        remainder(file) := remainder(file) \\/
            (size(bytes))
    END;
```

## Reposition

```
Reposition(file,pos) =
    PRE file : openfiles & pos : NAT &
```

```

        pos <= size(contents(file))
    THEN processed(file) := contents(file) /\ pos ||
        remainder(file) := contents(file) \/ pos
    END;
pos <-- Position(file) =
    PRE file : openfiles
    THEN pos := size(processed(file))
    END;
length <-- Length(file) =
    PRE file : openfiles
    THEN length := size(contents(file))
    END;

```

### The Open operation

The preceding operations are those promoted from the simpler model of a single anonymous file. Those operations use file-pointers, but so far we have not explained how those file-pointers come into existence.

An Open operation takes a *name* as an argument and returns a *file – pointer* that must be used in other file operations. The *name* may be the name of an existing file in the *filesystem* or it may be a new name. In the former case the *contents* of the opened file becomes a copy of the existing file; in the latter the *contents* is initially empty.

In all cases the opened file is positioned at the beginning of the file, with *processed* being empty and *remainder* being equal to *contents*.

### The Open operation

```

file <-- Open(name) =
    PRE name : NAME & name /\ ran(filename) &
        openfiles /= OPENFILE
    THEN ANY file
        WHERE file : OPENFILE - openfiles
        THEN
            IF name : dom(filesystem)
            THEN
                contents(file) := filesystem(name) ||
                remainder(file) := filesystem(name)
            ELSE
                contents(file) := <> ||
                remainder(file) := <>
            END ||
        END ||

```

### The Open operation

```

        processed(file) := <> ||
        filename(file) := name ||
        openfiles := openfiles \/ {file} ||
        file := file
    END
END;

```

### The Close operation

A Close operation takes an opened file and returns the *contents* component to the *filesystem*.

Notice that until the Close operation is completed no modification of a file is reflected in the original named file, if any, in the file system.

### The Close operation

```
Close(file) =
  PRE file : openfiles
  THEN filesystem(filename(file)) :=
    contents(file) ||
    filename := {file} <<| filename ||
    contents := {file} <<| contents ||
    processed := {file} <<| processed ||
    remainder := {file} <<| remainder ||
    openfiles := openfiles - {file}
  END;
```

### The Abort operation

Since a Close operation will, in general, change the file system, it is necessary to have an operation that enables us to abandon the file without returning the *contents* to the *filesystem*.

The Abort operation does this. It is the same as Close except for not changing *filesystem*.

### The Abort operation

```
Abort(file) =
  PRE file : openfiles
  THEN filename := {file} <<| filename ||
    contents := {file} <<| contents ||
    processed := {file} <<| processed ||
    remainder := {file} <<| remainder ||
    openfiles := openfiles - {file}
  END;
```

### The Delete operation

We can now specify operations on the file system.

Such an operation is Delete that deletes a named file from *filesystem*.

The file to be deleted must not currently be opened.

### The Delete operation

```
Delete(name) =
  PRE name : dom(filesystem) & name /: ran(filename)
  THEN filesystem := {name} <<| filesystem
  END;
```

### BackUp and Restore operations

We will specify a BackUp and Restore operation.

*BackUp* will copy the state of the *filesystem* to some external medium. Since we are not modelling the external medium, we cannot specify what happens externally. Instead we specify that there must be no effect on this system, by saying that BackUp behaves like *skip*, the substitution that does nothing.