Functional programming through the curriculum

Simon Thompson and Steve Hill

Computing Laboratory
University of Kent at Canterbury, U.K.
{S.J.Thompson,S.A.Hill}@ukc.ac.uk

Abstract. This paper discusses our experience in using a functional language in topics across the computer science curriculum. After examining the arguments for taking a functional approach, we look in detail at four case studies from different areas: programming language semantics, machine architectures, graphics and formal languages.

1 Introduction

We first explore our reasons for using a functional programming language, Miranda, as a vehicle for teaching a number of topics across the computer science curriculum. Figure 1 gives an overview of the courses we look at. We then give an overview of the paper itself.

Why?

We see five major reasons for using a functional language in a variety of components of a computer science course.

A common language First, a functional language provides a common medium in which we can express many of the ideas which come into a variety of courses. For example,

- rules in operational semantics;
- functions in a denotational semantics;
- sets and other data structures (which appear in many situations, such as non-deterministic automata (NFAs) and executable versions of model-based specifications);
- hardware description, and abstract machines of various sorts;
- geometric transformations and their composition, which appear in courses on computer graphics.

This language is familiar, so that the student can concentrate on any new ideas being expressed, rather than the particular way in which they are written down. There is anecdotal evidence for this from one of the author's experience: in covering a simple imperative language and its structured operational semantics

Degree programme	Computer Science and related subjects, which in the English system take three years to complete.
Students	Students are specialists, who have suitable qualifications in mathematics, or are taught the equivalent material in the first year of their programme.
Courses:	
Introductory programming	Year 1: 25 1 hour lectures + 20 classes.
Formal languages	Year 2: 8-10 lectures[1] + assessment work.
Machine architectures	Year 2: 10 lectures[1] + assessment work.
Formal Semantics	Final year: 10 lectures[1] + assessment work.
Computer Graphics	Final year: 5 lectures[1] + assessment work.

For each course assessment is by means of a mixture of continuous assessment (typically allocated 25% of the marks) and written examination.

Fig. 1. Summary of the degree programme and courses

the students preferred the (rather more cluttered) Miranda rendering to the rules written out in a natural deduction style.

It is easy for teachers to forget the overhead of learning another formal notation; our students are perhaps happier learning programming languages, which all follow the same ground rules, rather than more mathematical sorts of notation.

Of course there is a trade-off here; in restricting ourselves to a single (meta-) language in our studies we may limit some applications. One example might be denotational semantics, where our meta-language would be sequential.

High-level language The common language we have chosen is high-level; we gain advantages from this. In particular, the conciseness of the functional descriptions should help rather than overwhelm students.

For project work, the language supports rapid and accurate program development which is essential if students are to be able to perform substantial tasks with limited time available.

Static checking Our third reason is that the language is syntax- and type-checked: the descriptions we (or our students) write can be checked for syntactic correctness, and more importantly for type correctness. We use the types of the

[1] These lectures occur as parts of larger courses, typically taking 30 hours; the figures given here show the part allocated to the functional material discussed here.

language, particularly in giving semantic descriptions of programming languages; this point is discussed in more detail in Section 3.

Executable The fourth justification is that the language is executable. We gain, therefore

- executable semantic descriptions;
- prototypes of model-based specifications;
- machines and hardware which are directly executable.

Moreover, it is possible for students to test their solutions to exercises, as well as to embark upon larger-scale experiments.

Reinforcement Finally, using functional languages through the curriculum reinforces an initial exposure to functional programming. The ideas of lazy functional programming are subtle, and it would be naive of us to think that a first exposure would be sufficient for most students. In treating regular expressions and NFAs, for example, we find non-trivial instances of

- polymorphism: we use sets of various types of element;
- type abstraction: sets are a prime example;
- modularisation;
- higher-order functions: parsing regular expressions.

In first teaching Miranda we make links with imperative programming; these links can be strengthened as we continue to use the language.

Other issues In the longer term, we see the mathematical elegance of functional languages as affording opportunities for formal proof in a variety of areas, such as machine simulation and compiling. This is one area into which we hope to move in the future.

Finally, a rather more negative justification is that an isolated course in functional programming which is not followed up has a strong implicit negative message: "we teach you this stuff because we feel we ought to, but we don't use it ourselves"!

Overview of the paper

In the remainder of the paper we give a description of how we use functional programming in a number of areas, evaluating our approach as we go along. After giving a short description of how we introduce programming, we discuss in turn how we use a functional approach in covering the topics of programming language semantics, machine architectures, computer graphics and formal languages, before concluding the paper.

Some of the materials mentioned are available over the World Wide Web or by FTP; we detail this in the appropriate sections.

2 Learning to program

Functional programming has strong support at our institution, and we are able to draw on the expertise of some six lecturers and similar numbers of postgraduates and research staff. The topic is introduced in the first year with 25 lectures of basic material supported by a similar number of practical classes. The material is taught in parallel with 30 lectures and classes on the imperative language Modula-3.

In teaching functional programming we are mindful that our students also write imperative programs. We see the two approaches as complementary, with functional programming providing a valuable perspective on the imperative in a number of ways.

- A functional language is a useful *design* language for imperative programs, especially those which manipulate dynamic data structures. We can give functional list-processing programs which can be translated into an imperative language by adding the appropriate memory manipulating code.
- The different approach of functional programming can make plain what is happening in an imperative language: the different notions of 'variable' come to mind, for instance.
- A functional approach can also illuminate deficiencies in imperative languages, or alternative approaches which are unfamiliar to a more traditional programmer.

3 Semantics of Programming Languages

Since the inception of computing there has been interest in explaining in a clear and comprehensible way the behaviour of programs, that is giving a semantics to programming languages. The *denotational* school of Scott and Strachey, [5], aimed to give a mathematical model of (sequential, imperative) programs as functions from machine state to machine state. In order to find the appropriate structures to model these states and functions *domain theory* [8] was developed.

In retrospect, if not at the time, it is clear that the denotational semantics of a programming language can be factored into two parts.

- A functional model of the language is built, using an existing functional programming language — in this paper we shall use Miranda. Under this approach, the meaning of a command, for instance, is a function of the appropriate type: `stores -> stores`.
  In other words, the functional programming language is used as a semantic *meta*-language.
- The functional programming language itself is given a domain-theoretic semantics.

This separation makes clear the two quite different processes underlying the semantic description of the language.

- Using the basic notions of value, type, function and recursion we give a model of the more complex structures of an imperative language. These include
  - commands (as state transformers);
  - expression evaluation, which will in general have side-effects;
  - styles of parameter passing, with their corresponding styles of variable declaration ([6]);
  - different forms of binding: sequential or 'parallel', static or dynamic, and so forth.
- In the second stage, analyses of type, function and recursion have themselves to be given. It is only at this stage that the more technical aspects of domain theory need to be apparent.

This split shows that much can be gained by a student who only follows the first of these phases; s/he is able to see how the complex behaviour of a modern imperative language is rendered in simple (and hopefully familiar) terms.

The second phase, which involves further technicality, is optional. If it is examined, the first phase gives motivation for a closer examination of recursion in the definition of both functions and data, and so gives a clear reason for domains to appear. If the two phases are merged, it has been our experience that students find it more difficult to grasp what is going on; this is simply the lesson of 'divide and conquer' in the context of semantic descriptions rather than program development.

In the rest of this section we give an overview of our material on semantics in Miranda. This consists of descriptions of various aspects of a Pascal-like programming language together with an examination of its operational semantics, in the style of Plotkin. We discuss potential exercises and projects for students as we go along, and conclude with an evaluation of the approach advocated here, as well as looking at other advantages of the treatment.

The Miranda code and a reference document for the material can be found on the World Wide Web using the 'Further material' section given under the URL

`http://www.ukc.ac.uk/computer_science/Miranda_craft/`

or via anonymous FTP from the directory

`ftp://ftp.ukc.ac.uk/pub/sjt/Craft/`

Basic semantics

In writing the semantics we identify three stages. First we look at the base types we shall need to consider, then clarify the types of the major semantic functions, and finally we write the definitions of these functions.

Types First we have to establish how we model the programs themselves; we can use algebraic (or concrete) types to specify the structure of each syntactic category (commands, expressions and so on). The Miranda definition of `command` in Figure 2 shows how commands can be specified; note how the algebraic type

```
command ::= Skip |
            If_Then_Else b_expr command command |
            While_Do b_expr command |
            Sequence [command] |
            Assignment ident expr

values == num
lookup :: ident -> stores -> values
update :: stores -> ident -> values -> stores
```

```
command_value :: command -> stores -> stores
expr_value    :: expr -> stores -> values
nop_value     :: nop -> values -> values -> values
```

```
command_value Skip st = st

command_value (If_Then_Else e c1 c2) st
        = command_value c1 st , if  b_expr_value e st
        = command_value c2 st , otherwise

command_value (While_Do e c) st
        = command_value (While_Do e c) (command_value c st)
                , if b_expr_value e st
        = st    , otherwise

command_value (Sequence []) st = st
command_value (Sequence (c:cs)) st
        = command_value (Sequence cs) (command_value c st)

command_value (Assignment i e) st
        = update st i (expr_value e st)
```

Fig. 2. Basic denotational semantics

corresponds to a BNF-style syntax definition, and also that the type of commands is defined in terms of the types expressions `expr` and boolean expressions, `b_expr`.

Programs are to be modelled as functions from stores to stores, taking the machine state before executing the command to the state after the command terminates. We therefore need a type to model the store; at this level of the semantics we simply specify the signature required of the `stores` type, as is done in Figure 2; various implementations exist.

Typing the semantic functions Central to our approach is how we model commands; each command is seen as a function from `stores` to `stores`. The function interpreting commands, `command_value`, will therefore have type

```
command -> stores -> stores
```

The other declarations in the second part of Figure 2 show the value of typing the semantic functions in a separate phase, since these type declarations contain important information about the interpretation of various parts of the language. For example, we see that to give expressions a value we need a store (to interpret any variables in the expression), whilst to interpret a binary numerical operator (an object of type `nop`) no store is needed – operators have fixed values.

Were we to adapt the semantics to model a language with side-effects, this would be apparent in the type of `expr_value`; instead of returning an object of type `values` alone, the result would be of type `(values,stores)` in which the second component gives the state of execution after the expression evaluation has terminated.

Defining the semantic functions The definition of the functions themselves is straightforward; for commands we exhibit the definition in the final part of Figure 2. At this point it becomes clear that recursion is used in the modelling: a structural recursion runs along a `Sequence` of commands, while a potentially non-terminating recursion is used to interpret the `While_Do` loop.

Assessment In teaching this material we ask students to write definitions for themselves. It is instructive to look at `repeat` and `for` loops, as well as 'parallel assignment', `x,y:=e,f`. One obvious advantage for the student is that they can *check* their solutions for syntax and type errors using the Miranda system, and then for correctness by *executing* against example programs.

A second assessment building on the basic semantics is to add side-effects, which we do with the expression

```
Do_Return c e
```

whose effect is to execute the command `c` before evaluating the expression `e`. This requires students to think of changes to the types of the semantic functions before re-examining their definitions. Particularly instructive in this case is the parallel assignment command.

Extending the semantics

We have built a number of extensions of the basic semantics which illustrate various aspects of programming languages.

The definition mechanism An *environment* is used to keep track of the definitions in scope at any point during execution; this structure is quite separate

```
def_value     :: def -> env -> stores -> env
command_value :: command -> env -> stores -> stores
expr_value    :: expr -> env -> stores -> values
```

Fig. 3. Extending the denotational semantics

```
config ::= Inter command stores | Final stores

step :: config -> config

step (Inter (If_Then_Else e c1 c2) st)
  = (Inter c1 st)         , if b_expr_value e st
  = (Inter c2 st)         , otherwise
step (Inter (While_Do e c) st)
  = (Inter (If_Then_Else e (Sequence [c,While_Do e c]) Skip) st)
step (Inter (Assignment i e) st)
  = Final (update st i (expr_value e st))
```

Fig. 4. Basic operational semantics

from the store, which models the effect of commands on the machine state. The types of the main semantic functions are illustrated in Figure 3.

Abstraction: procedures and functions. There is considerable room for experimentation here.

- We treat different forms of parameter passing: value and reference as in Pascal, but with the possibility of adding others.
- We illustrate the difference between static and dynamic binding.
- We model recursive and non-recursive procedures.

Jumps We show the difficulty of interpreting languages with `goto` by extending the basic language with labels and jumps; the example illustrates the fact that modularity breaks down, with the interpretation function becoming a mutual-recursion involving the meanings of all the labels in the program.

In each of these cases there is room for students to experiment with the material, modifying or extending it and gaining feedback about the syntactic correctness of their work before executing it.

Operational semantics

An alternative semantic view is operational: we see the effect of a command as a series of execution steps for an abstract machine. Part of an operational model for our basic language is illustrated in Figure 4. The `configuration` of a machine is either

`Final st` : the machine has terminated in state `st`, or,
`Inter c st` : the command `c` is to be executed, starting at state `st`.

One `step` of execution takes one `config` to the next, and various cases of `step` are given in the figure.

On teaching this material, the rules were presented in functional form as well as more traditional 'deduction rule'; it became apparent that although the latter form was more abstract (and to us easier to read) the students preferred the Miranda version because the syntax was familiar, and so they were able to concentrate on the ideas, rather than on the surface syntax.

Conclusion

This section shows how a functional language is adequate for the functional description of many aspects of modern programming languages. Further details of this work are to be found in [7].

The advantages of this approach are threefold

- The semantics are presented in a language which is executable. In doing assessment work, students are able to check the syntax and typing of their work, before executing their solutions.
- The semantics are presented in a familiar language. Even if definitions are somewhat less elegant, readers can concentrate on the ideas rather than the syntax.
- The two phases of the semantics — going to a functional language; interpreting that language — are explicit here, and we have found this avoids some of the confusions of other expositions.

4 Machine Architectures

The work in this area arose from the need to provide a platform for the simulation of microprocessor architectures suitable for undergraduate students of the core computer science course. The problem was this: in the second year of our undergraduate programme, two groups of students study a digital systems course. The first group study Computer Systems Engineering which is oriented more towards electronics than the second group who are reading a Computer Science degree. Originally, the digital systems course contained a laboratory experiment which involved a fair amount of practical electronics. We decided that it was an unreasonable requirement that the mainstream computer scientists, especially those from largely mathematical or computing backgrounds, should have to perform this experiment. It was proposed, therefore, that these students be offered a software-based project as an alternative.

This provided an ideal opportunity for an experiment in using a functional platform, which we wanted to do for reasons discussed in the Introduction: in particular we wanted a concise yet precise description of machines, as well as a platform upon which to build project work.

We chose to provide simulations for two architectural styles - a register machine and a stack machine. Both machines share a common core which is extended to provide their peculiar instruction sets. The simulations are constructed in three levels.

– The core machine provides the basic architecture described by means of primitive transitions of machine state.
– The micro-code provides a specialisation of the core machine by implementing an instruction set in terms of the basic transitions.
– The assembly language interface is implemented by an assembler and loader which together construct an initial machine state. This is then run until the machine halts.

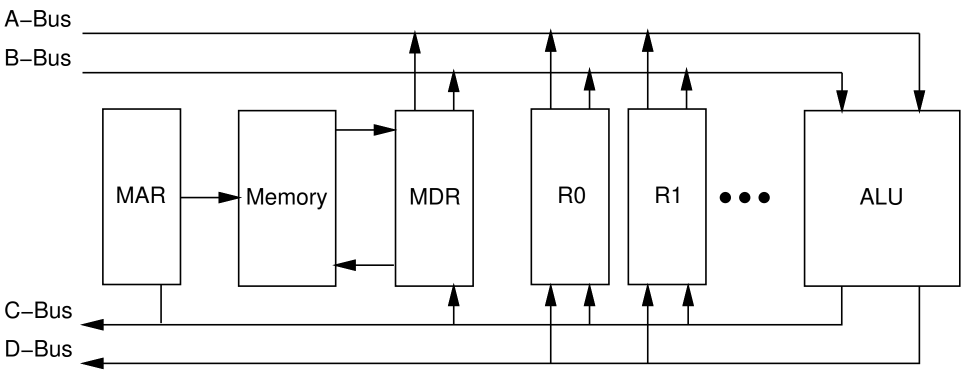

Fig. 5. Architecture of the Core Machine

Implementation

The core machine, depicted in Figure 5, provides a characterisation of a generic machine architecture. It comprises a type of "machine state" along with a set of permitted state transitions. These transitions are the only ones allowed. The style is similar to that adopted by Peyton Jones and Lester [3] for the description of abstract machines for the implementation of functional languages.

Ideally, the type of machine state would have been made abstract, but it is not possible to cover abstract datatypes in sufficient detail in the first-year functional programming course to allow this.

The machine was decomposed into the following parts:

– Memory - the memory is modelled as an association list between address and contents.

– Memory Interface - the memory interface comprises two special purpose registers - the memory address register (MAR) and the memory data register (MDR).
– Register File - the registers are modelled as an association list between register number and register contents. The core machine thus makes no commitments as to the number of registers available.
– Buses - the machine has four internal buses or data highways.
– Statistics - the statistics field is used to accumulate measures of the machine's performance.
– Halt Flag - this indicates if the machine has halted.

These components are conveniently represented in Miranda as a tuple.

```
address   == num
word      == num

memory    == alist address word
interface == (word, word)
registers == alist num word
buses     == (word, word, word, word)

machine   == (memory, interface, registers, buses, stats, bool)
```

The core machine is augmented by a set of transitions which define the valid actions a machine may make. Most transitions involve the movement of data from one place to another. Thus they also define the data paths that exist within the machine. Some example transitions are given:

```
transition == machine -> machine

regToAbus :: num -> transition

regToAbus n (m, i, r, (a, b, c, d), s, h)
          = (m, i, r, (a1, b, c, d), s, h)
            where a1 = aLookup n r

mdrToAbus :: transition

mdrToAbus (m, (mar, mdr), r, (a, b, c, d), s, h)
        = (m, (mar, mdr), r, (mdr, b, c, d), s, h)
```

The primitive transitions are combined via a small set of combinators. The most important `comma` is a version of function composition:

```
comma :: transition -> transition -> transition
```

```
(t1 $comma t2) m = t2 (t1 m)
```

and is used to construct the derived combinator:

```
do :: [transition] -> transition

do []     = id
do (t:ts) = t $comma do ts
```

The `switch` transition is more specialised. It allows a transition to be selected from a table according to the contents of a register. Its role mimics the operation of the mapping PROM in a micro-code engine. Similarly, it is often the case that a section of micro-code is parametrised on a register value and the function `passReg` is provided for this purpose.

```
switch  :: num -> alist num transition -> transition
passReg :: num -> (num -> transition) -> transition
```

We are now in a position to be able to define transitions which correspond more closely to the register transfer style. The first allows the contents of one register to be copied to another and might be written as:

$$R_s \rightarrow R_d$$

```
regToReg :: num -> num -> transition

regToReg rs rd
 = do [ regToAbus rs,
        aluCycle AluA,
        cbusToReg rd ]
```

Finally, some compound transitions for combining registers via the ALU are provided. These might be written in the register transfer style thus:

$$\begin{array}{l} \ominus R_n \rightarrow R_d \\ R_n \oplus R_m \rightarrow R_d \end{array}$$

The second of these transitions is presented:

```
op2 :: num -> aluOp -> num -> num -> transition

op2 rn op rm rd
 = do [ regToAbus rn,
        regToBbus rm,
        aluCycle,
        cbusToReg rd ]
```

Combinations of transitions are used to implement a fetch-execute cycle where each instruction is coded as a compound of basic or derived transitions.

The final stage of the simulation was to provide an assembly language, loader and functions to run programs to completion (*ie.* until the halt flag is set) and to print out statistics. Using a functional programming environment here was of great benefit. Programs were represented simply as lists of instructions which were themselves elements of an algebraic datatype. There was no need to have a concrete syntax for assembly language programs, nor parsing/unparsing functions. Instead, the syntax of lists and constructors is used directly, and the compiler provides adequate checking and error messages.

For simplicity, labels were not implemented, although in retrospect this was probably a mistake. Many of the errors that students encountered in their test data were due to incorrect jumps.

Assessment

Students perform a single sixteen-hour assessment based on the simulation. Their tasks include the following:

- Read the core machine definition and produce a schematic diagram similar to Figure 5.
- Implement the instruction sets of two similar machines and perform some optimisations on these machines.
- Write test programs for the machines, and collate performance statistics.

The first task provides a useful revision of Miranda syntax, this being the first functional programming that the students encounter after their first year course. It provides a useful revision exercise, as well as getting them to think about the machine architecture. The transitions are named such that a detailed understanding of their operation is not required.

Conclusion

The core definitions can be regarded as defining a meta-language or micro-code for the core machine. For the purposes of the simulation exercise, the students need only be proficient in a small subset of the Miranda language, namely the syntax of lists, function application and definition. Experience would suggest that the approach is successful. When students have problems with the work, it is most often to do with the implementation of their machine, rather than the details of functional programming.

However, we must be somewhat cautious. The groups that attempt this assessment are self-selecting. Any student who struggled with functional programming in the first year is unlikely to want to attempt this work. Between a half and a third of the CS cohort opt for the alternative hardware-based experiment each year.

From the point of view of the implementer, the simulator has been a great success. During the three years of its use, we have identified only a few minor bugs which were fixed in a matter of minutes. One was due to a typographical error and a couple of others were introduced when the simulation was modified to emulate a new architecture. Performance was not a problem for us since the students' test programs were quite small. Further details of this implementation can be found in [2].

5 Computer Graphics

In their text, Salmon and Slater [4] use a notation based on Standard ML to describe many features of higher-level graphics libraries. The reason for using a functional notation which they cite is conciseness. In particular the expression of values of simple datatypes is uncluttered and requires no explicit memory allocation.

Similar motivations lead to our use of Miranda in a final year course on computer graphics. We have found it to be a convenient language for the description of geometric transformations and building upon this hierarchical geometric models.

The first stage of this part of the course introduces the following notions:

- the abstract concept of a geometric transformation
- an implementation based on homogeneous transformation matrices where composition is achieved via matrix product
- an implementation based on functions where composition is achieved via functional composition

In the Miranda implementation, the homogeneous matrices are treated as an abstract data type with given implementations of the common transformations and matrix product. The function-based implementation is typified by definitions such as:

```
translate :: point2 -> point2 -> point2
translate (tx, ty) (x, y)
    = (x + tx, y + ty)

rotate :: num -> point2 -> point2
rotate t (x, y)
    = (x * cos t - y * sin t,
       x * sin t + y * cos t)
```

Such transformations can be combined naturally with function composition, but for consistency with the matrix notation (which uses row vectors for points), we chose a variant with the arguments reversed giving a natural left-to-right reading.

```
(t1 $o t2) p = t2 (t1 p)
```

The next stage in the course introduces the notion of a symbol (sometimes called a structure). Symbols are essentially parametrised (over transformation and possibly a graphics environment) graphical objects. We describe two approaches for representing symbols:

- a symbol is a function taking a transformation to a sequence of graphical commands, or
- a symbol is a list of graphical commands. An instance of the symbol is obtained by applying a transformation to each of the commands to obtain a sequence of graphical commands.

The final step is to construct hierarchical geometric models from the symbols. Again, we present two techniques.

- A hierarchy is constructed using functions parametrised on a (global) transformation. The transformation is applied to all graphical operations at this level. All children are invoked with augmented transformations which are the composition of the global transformation and any local transformations required to position them correctly within the model. For example:

  ```
  robot t p m =
    base m ++
    arm1 (a1 $o m) ++
    arm2 (a2 $o m)
    where
    a1 = rotate t $o
         translate 0 l1
    a2 = rotate p $o
         translate 0 l2 $o
         a1
  ```

  where `arm1`, `leg1` and `base` are the symbols which constitute the relevant parts of the robot.
- A hierarchy is constructed as a tree. Each node contains a symbol, a local transformation and a list (possibly empty) of children. A function is provided to instance a tree. It visits each node maintaining a current transformation which is the composition of any global transformation and all the local transformations on the path from the root to the current position in the tree.

  ```
  tree ::= Node symbol trans [tree]

  figure m =
    Node body m [
       Node arm arm1 [],
       Node arm arm2 [],
       Node leg leg1 [],
       Node leg leg2 [],
       Node head head1 []]
  ```

```
matches :: reg -> string -> bool

matches (Or r1 r2) st
  = matches r1 st \/ matches r2 st
matches (Then r1 r2) st
  = or [ matches r1 s1 & matches r2 s2 | (s1,s2)<-splits ]
    where
    splits = [ (take n st,drop n st) | n <- [0..#st] ]
```

Fig. 6. Regular expression matching

```
draw_tree m1 (Node sym m2 l) =
  sym m3 ++
  concat (map (draw_tree m3) l)
  where
  m3 = m2 $o m1
```

Here `arm`, `leg` and `head` are symbols, and `arm1` *etc.* are the local transformations which position these symbols within the model.

Conclusion

As with Salmon and Slater, we found the major advantage of the use of Miranda to be conciseness. The ease with which new datatypes can be defined and values of these types can be expressed makes the presentation of material of this nature much easier. Many imperative languages have a baroque syntax for literal values of anything but the predefined datatypes and this is both distracting and wasteful of space. When lists or trees are involved the notation becomes unwieldy and often impractical to present on, say, a single OHP slide. Miranda has a concise notation for the values of all algebraic datatypes, with a particularly concise notation for lists.

In their text, Salmon and Slater also use Pascal. They state that one should regard the Pascal as an implementation of the higher-level ML presentation. This is precisely one of the messages we try to give in our first year courses.

6 Formal Languages

In a short module on the processing of formal languages we cover regular expressions, and the different sorts of automaton used to recognise them, as described in [1], Chapter 3. We use Miranda as a description and implementation language for various of the ideas here. This material is also available through the URL

`http://www.ukc.ac.uk/computer_science/Miranda_craft/`

Matching

After describing regular expressions as a Miranda type, we are able to give a Miranda definition of when a string matches an expression, the function `matches` of Figure 6. The description is short, and more importantly *unambiguous*. In this context we are using Miranda as a formal specification language.

The system

In our system we give implementations of

- A type of NFAs, and a simulation of NFAs;
- a function transforming a regular expression into an NFA;
- a function making an NFA into a deterministic machine, a DFA;
- a function optimising a DFA by minimising its state set.

Much of the code can be re-used; we discuss a particular case in the next section.

Sets

The automata used to recognise matches are built from sets; we exploit the Miranda `abstype` mechanism to hide the particular implementation of the sets. Moreover, in different parts of the implementation we need to consider sets of different type: in the simple non-deterministic automaton we consider sets of numbers, in building a deterministic version we use sets of sets of numbers; polymorphism supports this sort of re-use.

Programming the system

Using a programming language forces us to consider both the details of the system and how it is built from its constituent parts. A functional language is sufficiently high-level that the details do not engulf the wider picture; for example, we need do no explicit memory management in a functional description. The Miranda language also has a module system, and this is most helpful in putting together the complete implementation.

As in earlier sections, the twin advantages of type/syntax checking and executability give us assurance that what we have written is sensible, as well as allowing students to experiment with the systems and their assessment work.

7 Conclusions

In the introduction to this paper we argued that there were considerable advantages to using a functional language as a teaching vehicle in a computer science degree. We illustrated our arguments with examples from four areas: semantics, architecture, graphics and formal languages. We believe that there are other parts of the degree in which a functional approach will be equally useful, specification animation and program verification being two obvious examples, and we hope to explore these and other topics in the years to come.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.
2. Steve Hill. The functional simulation of a simple microprocessor. Technical Report 17-94, UKC Computing Laboratory, 1994. Available by ftp from `unix.hensa.ac.uk` in the directory `pub/misc/ukc.reports/comp.sci/reports` as the file `17-94.ps.Z`.
3. Simon L. Peyton Jones. *Implementing Functional Languages.* Prentice-Hall, 1992.
4. Rod Salmon and Mel Slater. *Computer Graphics - Systems and Concepts.* Addison-Wesley, 1987.
5. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey approach to programming language theory.* MIT Press, 1977.
6. Robert D. Tennent. *Principles of Programming Languages.* Prentice Hall, 1979.
7. Simon Thompson. Programming language semantics using Miranda. Technical Report 9-95, Computing Laboratory, University of Kent at Canterbury, 1995.
8. Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

This article was processed using the LaTeX macro package with LLNCS style