Programming Language Semantics using Miranda

Simon Thompson

ABSTRACT

This paper explains the use of the functional programming language Miranda as a vehicle for describing the semantics of imperative programming languages. In particular we give a Miranda denotational description of a substantial subset of a Pascal-like language, describing a number of variants of the semantics, including parameter passing by value-result, dynamic binding of values to names and a simple semantics of jumps.

We also give an executable operational semantics of our basic language, as well as a compiler for this language into a simple stack machine, which is itself modelled in Miranda.

Introduction

This paper gives a concise description of various semantic features of Pascal-like languages (Pascal, Modula-2, Modula-3, Ada and so on) written using functions described in the Miranda[1] programming language. Further details about Miranda can be found in [Thompson]. The paper is intended to provide a reference document for the Miranda semantics we have implemented, rather than a general introduction to programming language semantics. We expect that the material here would be used as a part of a course in which the principles of semantics were explained.

The approach we adopt to semantics is best described by [Tennent] which gives an excellent overview of the way in which programming languages work. In particular we use the following ideas.

- Tennent discusses two kinds of value for a variable name: the l-value and the r-value. The l-value of a variable is the storage location denoted by the variable, in other words the interpretation of the variable when it appears on the left-hand side of an assignment. The r-value is the value stored in that location — the value the variable has when it appears on the right of an assignment.
- He also makes a separation of the model of values into two parts: an environment which associates names with their values — in particular associating a location with a variable name — and a store which associates each location with the value stored therein. (In the rest of the paper we will use the terms ‘store’ and ‘state’ interchangeably.)

The paper is structured as follows. In Section 1 we give an introduction to the syntax of our simple ‘`basic`’ language, and we follow this with its semantics in Section 2. Section 3 extends the language to include definitions and block structure, whilst in Section 4 we treat some variants of the language, including a number of parameter passing mechanisms, a semantics for jumps and an explanation of side-effects.

Section 5 gives a description of an operational semantics of the `basic` language, and Section 6 shows how the language can be compiled and executed on a simple stack-based abstract machine which is itself described in Miranda.

---

[1] Miranda is a trademark of Research Software Limited.

We make some observations about the approach of the paper in Section 7, and in an Appendix we give an informal explanation of the foundations of a domain-theoretic explanation of recursion.

The code for the functions is to be found under the URL

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

This code is divided into a number of directories, within which a uniform naming convention is observed; for instance, the file named

```
command_values.ins
```

will always contain the function(s) interpreting commands.

I am very grateful to Howard Bowman and Steve Hill for their comments on a draft of this paper.

1 Abstract Syntax in Miranda

Before we can give a formal description of the semantics of various languages, we need some way of representing the parsed form of sentences in a formal way. As we have chosen Miranda as our formal language, we need to represent parse trees, or the 'abstract syntax' of our languages in Miranda.

Our first example language is a simple single-typed language with global integer variables. The control constructs are those of most Pascal-like languages, with a null command (`skip`) added.

```
skip
if .. then .. else ..
while .. do ..
repeat .. until ..
 .. ; .. ; ..
 .. := ..
```

Parse trees can be represented using algebraic types — our type of commands will be

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

Looking at a `While_Do` node, for instance, we see that it has two sub-trees, the first a boolean expression (`b_expr`) and the second a command. Observe that we have used a Miranda list type in our abstract syntax for a sequence of commands.

The representation we use bears a strong resemblance to BNF — we use `::=` as a symbol, but more significantly the general form of the expressions mirrors BNF, | delimiting alternatives and `[..]` indicating '0 or more occurrences of'. In effect, we can go mechanically from BNF to Miranda.

Our language contains integer and boolean expressions (`expr, b_expr`), which are built from unary and binary operations and binary relations, in the standard way. For completeness we list their formal representation now.

```
expr ::= Var ident |
         Num num |
         Apply_nop nop expr expr |
         Apply_monop monop expr
```

```
b_expr ::= Bool bool |
           Apply_rel rel expr expr |
           Apply_bop bop b_expr b_expr |
           Apply_monbop monbop b_expr

nop ::= Add | Subtract | Multiply
monop ::= Abs | Minus
rel ::= Greater | Greater_eq | Equal
bop ::= And | Or
monbop ::= Not
ident ::= Name [char]
```

It is simple to define pretty printing functions for this syntax. The functions are defined by recursion over the structure of the expressions in the way that we would expect, and show in a graphic way how the algebraic types contain the parsing information. For example, in printing an `expr` in fully parenthesised form we write

```
e_pretty (Num n) = show n
e_pretty (Apply_nop op e1 e2)
  = "("++ e_pretty e1 ++ nop_pretty op ++ e_pretty e2 ++ ")"
```

among the equations of the definition of `e_pretty`.

In the distribution, the syntactic types are to be found in the files

```
abstract_syntax.m
```

2 The denotational semantics of commands

2.1 Introduction

This section starts our elucidation of the denotational semantics of various linguistic features. We are eventually aiming to provide a full description in Miranda, but we shall try to reach this gradually, concentrating on the main features of the semantics, and ignoring some of the more mundane operations. The files containing the Miranda sources for the functions here are contained in the directory

```
basic
```

which forms part of the distribution.

In the discussion of each semantic feature we shall go through three stages:

- Identify the data types upon which the semantics is based, such as `stores` and `env`.
- Identify the types of the semantic functions we shall define.
- Define the semantic functions themselves.

This separation of concerns is as important here as it is in programming — it gives us some control over how we perform a complex task.

2.2 The data types

We are looking at an extremely simple language here. In particular, names or identifiers are bound to the same locations for the whole of a program, as there are no local variable declarations, or indeed any definitions at all. Because of this we don't need an environment to describe bindings, so we can think of `stores` thus:

```
stores == ident -> values
```

where `values == num`. A member of the type `stores` associates values with identifiers directly. Note that this circumstance is only temporary; for the more complex languages we see below we shall use an environment and store respectively to model definitions and storage. In the distribution, the types listed here are to be found in the file

```
semantic_types.m
```

2.3 The types of the semantic functions

The main function of interest is that which interprets commands. The crux of the denotational approach is that we see the effect of a command as a function, which takes the store before the command is performed as input and which returns the store after the command is performed. This means that we interpret each command as being of type `stores -> stores`, and the function which gives commands values will be

```
command_value :: command -> stores -> stores
```

The values of expressions depend on the values held in the store, and so their interpretation functions will be:

```
expr_value :: expr -> stores -> values
b_expr_value :: b_expr -> stores -> bool
```

We shall want to manipulate stores, and in particular to perform two operations. First, in evaluating expressions we shall need to `lookup` values of identifiers:

```
lookup :: ident -> stores -> values
```

In interpreting assignment commands (and indeed all commands, indirectly) we need to be able to update the value stored in a particular variable:

```
update :: stores -> ident -> values -> stores
```

Another approach to modelling the store would declare the type as an `abstype` carrying the operations to lookup, update and initialise a store.

These are the major functions that we shall use — others we need give meaning to operations, relations etc. Their types can be found in the file

```
sem_fun_types.ins
```

This file is inserted (textually) rather than included as a module; this is because it it not possible to separate type declarations from their corresponding function definitions in Miranda.

2.4 The function definitions themselves

The major definition is that of

```
command_value :: command -> stores -> stores
```

The definition is written using a case analysis over the type `command`; the definition is found in the file `command_values.ins` We go through this definition now, case by case.

```
command_value Skip st = st
```

The effect of the command `Skip` is to leave the state `st` unchanged.

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

In interpreting `(If_Then_Else e c1 c2)` first evaluate `e` in the state `st`. If its value is `True`, then output the state resulting from performing `c1` in state `st`, that is `command_value c1 st`; otherwise perform `c2`, that is output `command_value c2 st`.

```
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
```

The while-loop is modelled thus: first evaluate the condition `e` in state `st`, that is, find `b_expr_value e st`. If the result is `False` then do not modify the store, so we return `st` as result. Otherwise, we find the result of performing `c` in `st`, that is evaluate

```
command_value c st
```

and pass the result back to the function which interprets the loop, i.e.

```
command_value (While_Do e c)
```

Note that we have used recursion to explain the (iterative) `while`-loop. In a similar way we explain the `repeat`-loop:

```
command_value (Repeat_Until c e) st
        = command_value (Repeat_Until c e) st'
                , if ~ (b_expr_value e st')
        = st'   , otherwise
          where
          st' = command_value c st
```

Observe that the condition is evaluated in the state `st'`, the state resulting from performing command `c` in state `st`: we perform the command before making the first test, in other words.
How are we to explain the effect of a sequence of commands `c1;c2;..;ck`? There are two cases, depending whether the list is empty or not. An empty list has the effect of a null command,

```
command_value (Sequence []) st = st
```

while the effect of the list `Sequence (c:cs)` is that of `c` followed by `Sequence cs`.

```
command_value (Sequence (c:cs)) st
        = command_value (Sequence cs) (command_value c st)
```

The point comes when we have to explain our primitive command, that of assignment. Using the update function and `expr_value` this is easily done:

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

In our definitions we have used the expression evaluation functions and the two store manipulation functions. Now is the time to define them. First we define the latter. Recall that

```
stores == ident -> values
```

so that to look up the value of an identifier in a store we simply apply the store to the identifier:

```
lookup :: ident -> stores -> values
lookup i st = st i
```

The result of updating a store, `update st i val` should yield the same values as `st`, except when evaluated on `i`, when the value `val` is to be returned:

```
update :: stores -> ident -> values -> stores
update st i val j = st j , if i ~= j
                  = val  , otherwise
```

Note that in this definition we are defining the function `update st i val` by saying how it behaves when applied (to `j`).

2.5 Definition of the other functions

Now we explain how expressions are evaluated. The values of variables are given by `lookup`, number literals are interpreted in the obvious way, and in composite expressions we apply the operator to the values of the subexpressions which form its arguments:

```
expr_value :: expr -> stores -> values
expr_value (Var i) st = lookup i st
expr_value (Num n) st = n
expr_value (Apply_nop f e1 e2) st
        = (nop_value f) (expr_value e1 st) (expr_value e2 st)
expr_value (Apply_monop g e) st
        = (monop_value g) (expr_value e st)
```

Boolean expressions are interpreted in the same way as expressions, using the usual recursive algorithm.

```
b_expr_value :: b_expr -> stores -> bool
b_expr_value (Bool tv) st = tv
b_expr_value (Apply_rel re e1 e2) st
        = (rel_value re) (expr_value e1 st) (expr_value e2 st)
b_expr_value (Apply_bop bo b1 b2) st
        = (bop_value bo) (b_expr_value b1 st) (b_expr_value b2 st)
b_expr_value (Apply_monbop mbo b) st
        = (monbop_value mbo) (b_expr_value b st)
```

Finally the operators have to be given values. Note that since `nop_value` takes no argument of type `stores`, their values are independent of the machine state, as we would hope! We show the function `nop_value` as an example; the others are similar.

```
nop_value :: nop -> (values -> values -> values)
nop_value Add = (+)
nop_value Subtract = (-)
nop_value Multiply = (*)
```

The file containing these definitions is

```
expr_values.ins
```

2.6 Conclusion

We have shown how to model commands as store transformation functions. To re-iterate, a command can be thought of as a function which takes the store before execution as input and produces as output the store after execution. We showed how to model various primitive, alternative and iterative constructs.

It is interesting to review the dependencies between the major semantic functions. In interpreting programs (which are commands) by means of the function `command_value`, we need to interpret expressions and boolean expressions, so we call `expr_value` and `b_expr_value` which itself calls `expr_value`.

The reader might like to think about how in this semantics to interpret

```
if .. then ..
for <ident> := <expr> to <expr> do <command>
```

and the parallel assignment command

```
v1,v2,..,vk := e1,e2,..,ek
```

which is intended to assign the values of `e1,e2,..,ek` to the variables `v1,v2,..,vk` simultaneously.

In the distribution directory, `basic`, can be found the file putting together the various semantic files:

```
semantics.m
```

The top-level file

```
top_level.m
```

also includes an examples file and the pretty printing functions. It is this file which should be loaded into Miranda for the semantics to be executed.

3 Definitions and abstractions

3.1 Introduction

In Section 2 we discussed the semantics of commands and expressions, or at least that part of expression semantics which deals with side-effect-free expressions. Here we consider two more categories of program component; we investigate how to model definitions of constants, variables and procedures, and how to treat the abstraction implicit in a procedure definition. (We shall mention the final category of sequencers in Section 4.6.) The directory containing the sources of this material is

```
full
```

We can first get an informal idea of what we shall be doing by explaining how we are going to augment our language to include these features. We shall keep the single data type of numbers, so as not to complicate the semantics overmuch.

Definitions will take the forms:

```
const c (27+32) ;
var eric ;
procedure fred ( x , joe , jack ) ;
  .. procedure_body ..
```

Procedures are called in the standard way, by naming them and listing their actual parameters. For instance, a valid procedure call is given by:

```
fred ( 27 , jack , eric + sid ) ;
```

We treat parameters as value parameters; we shall discuss other approaches in a subsequent section. Our final new control construct is the block which is the means by which we introduce local definitions:

```
begin block
  const c 17 ;
  procedure fred (y);
    .. body of fred ..
  const d (33.3+c) ;
  <command>
end block
```

The `<command>` is the body of the block, and the definitions in the list of definitions are local to the block. The Pascal scoping discipline says roughly that an identifier is in scope after its definition, so that it can be used both in subsequent definitions and (recursively) within the definition itself. In the model we discuss here we adopt this, except that we don't allow recursive definitions. A later section studies recursive procedures.

How do we add to our abstract syntax? We augment the definition of `command` thus:

```
command ::= .. | .. | ..
            Block [def] command |
            Call ident [expr]
```

The first addition is the syntax for a command block, and the second for procedure call — it includes the procedure identifier and the list of actual parameters. We describe the type `def` for definitions so:

```
def ::= Const ident expr |
        Variable ident |
        Procedure ident [ident] command
```

A definition can take one of three forms, a constant, variable or procedure definition. Note that the variable definition does not explicitly mention the location bound to the name; the location will be supplied by the storage allocation mechanism of the implementation. The procedure definition has three components — the name of the procedure, the list of names of the formal parameters and the command which is the body of the procedure.

Linked to our discussion of scopes above we can point out a highly desirable property of denotational semantics. This property is completeness. Because we write a function which describes the input-output relation, there is no room for ambiguity to creep in, as could easily happen in an informal description. What we get in our case is a particular, well-defined, action which may not be that action we (implicitly) expected, but which does at least prescribe the language in a definite way. This must be better than a system where a feature may be un- or under-defined.

3.2 The types

As we now have local definitions, we shall have to keep track of both how names are bound to locations (using the environment) and how values are associated with locations (using the store). The values associated with constants will be numbers and the values associated with variables will be locations. What will be the value associated with a procedure name?

We conceive of procedures as functions which take as input both the starting state and the parameter values, and which return the termination state. In other words, the type of procedures, `proc` will be given by

```
proc == ( stores , [values] ) -> stores
```

Our type of values will be

```
values ::= Number num |
           Loc locations |
           Abstract proc
```

The type of locations – the storage locations in memory – is modelled by `num`.

What will be the types of stores and environments? Remember that we want these to constitute a record of associations between names or locations and values. How this is done is irrelevant. For instance, environments are (or should be) defined by the action of the operations

```
find :: ident -> env -> values
```

which finds the value associated with an identifier in an environment;

```
bind :: ident -> values -> env -> env
```

which updates an environment with a new name-value binding, and

```
free :: env -> locations
```

which returns location which is unused in the environment. This function is used when interpreting variable declarations, at which point a name is associated with a 'new' piece of storage. We want to conceive environments as an abstract data type, in fact. It is clear that we can implement the environment type using lists (rather than the functions we used in Section 2 to model stores)

```
env == [(ident,values)]
```

the binding of a value `v` to an identifier `i` being registered by the presence of the pair `(i,v)` in the list.

Arguing similarly, we can implement stores as

```
stores == [(locations,values)]
```

(We could equally well use the implementation of Section 2, in fact.) These definitions are in the file

```
semantic_types.m
```

3.3 The types of the semantic functions

In giving a value to an expression containing, for instance, both constants, whose values are held in an environment, and variables, whose r-values are located in a store, we need to supply information about both environment and store. We must modify the types of the expression evaluation functions thus:

```
expr_value :: expr -> env -> stores -> values
b_expr_value :: b_expr -> env -> stores -> bool
```

In a similar way, a command yields a store transformation function only after we have interpreted the names used by the command. We must interpret commands in an environment:

```
command_value :: command -> env -> stores -> stores
```

What is the effect of performing a definition? We keep a record of definitions in the environment, and so,

```
def_value :: def -> env -> stores -> env
```

(We need the `stores` parameter to interpret the expressions which may lie on the right hand sides of definitions, e.g.:

```
const c (x+9.76) ;
```

where `x` is a variable. Note that we only resolve our constant definitions at run-time, unlike Pascal and related languages for which the evaluation is done at compile-time.)

As an abstract type, how does the store appear? We retain the functions

```
update :: stores -> locations -> values -> stores
lookup :: locations -> stores -> values
```

Our final semantic task is to interpret procedures like

```
procedure joe(y) ;
  x := y + x ;
```

It is usual to interpret the free names in a procedure, like `x`, in the environment in which the procedure is defined. This is called static binding; in contrast we can resolve the names in the prevailing environment when the procedure is called; this is called dynamic binding, and is discussed in Section 4.4. If `x` is a variable we only resolve its l-value in the definition environment, we find its r-value from the state in which the procedure is invoked. To re-iterate, state information is passed to the procedure body on call, and so no state information is needed in giving the procedure a value, as a state transition function:

```
proc_value :: [ident] -> command -> env -> values
```

In conclusion, note that we now interpret constructs relative to (at most) two objects, the state and the environment. We have had to modify our storage model to include an explicit allocation function. We introduced two new major semantic functions, `def_value` and `proc_value`, with the obvious intention. The declarations are found in

```
sem_fun_types.m
```

3.4 The definition of the semantic functions.

Here we define the new semantic functions, and explain how we extend `command_value` to interpret our new commands `Call` and `Block`. Finally we explain how the definitions of Section 2 are modified to accommodate our change from the simple store model to the store/environment model.

Perhaps this is a good place at which to make an aside. One useful way of looking at denotational semantics is simply as a translation from a general (in this case imperative) programming language to a functional one. Some of the novelties (peculiarities?) which appear in the semantics can be explained by this. For example, because we don't have a state in a functional language, we have to pass explicit state (and environment) information to command, expressions etc. If we want to repeat a particular operation, this will be done by composing functions, recursion or iteration along a list. The functions `map` and `foldr, foldl` are iterators of this sort. Once we see that many of the definitions are standard functional 'idioms' it should be a lot easier to see the precise effect of the semantics.

3.4.1 Definitions

Definitions are intended to bind and we see that happening in the definition of

```
def_value :: def -> env -> stores -> env
```

which proceeds by cases over the form of definition.

```
def_value (Const id ex) en st
        = en'
          where
          en' = bind id val en
          val = (expr_value ex en st)
```

In a constant definition, we evaluate the right hand side, giving `val` and then `bind` the result to the identifier. Note that we do not affect the state.

```
def_value (Variable id) en st
        = en'
          where
          en' = bind id (Loc (free en)) en
```

In interpereting a variable declaration we allocate a new location, `free en` and bind the variable name to it.
Finally,

```
def_value (Procedure id l c) en st
        = en'
          where
          en' = bind id val en
          val = Abstract (proc_value l c en)
```

Find the value of the procedure using `proc_value` and bind it to the procedure name. We return the updated environment together with the unmodified store. Note that the procedure body is itself interpreted relative to the environment `en` in which the procedure under definition does not itself appear. This means that, as we mentioned earlier, our procedures are not recursive.

3.4.2 Commands

There is a strong correspondence between block entry and procedure invocation. As the former is simpler it seems sensible to look at how we interpret our new commands before we look at procedures. In the following we are going to omit some of the details of the definitions so that we get a clear view of the important points. The full (executable) details can be found in the directory `full` of the distribution. Recall that the function interpreting commands has the type

```
command_value :: command -> env -> stores -> stores
```

We look at the cases not covered earlier.

```
command_value (Call id elist) en st
        = val ( st , elist_value )
          where
          Abstract val = find id en
          elist_value = ...
```

We first find the value of `id` in `en`. This is of the form `Abstract val`, with `val :: proc`. We apply `val` to the current state, `st`, together with the values of the actual parameters, `elist_value`. We use `expr_value` in finding these values — essentially we map it along the list of actual parameters.

```
command_value (Block dlis c) en st
        = command_value c en' st
                where
                en'=def_list_value dlis en st
```

The function `def_list_value` is used to successively update the environment `en` by the list of definitions `dlist`. We therefore interpret the command `c` in an environment which includes all the definitions in `dlist`, as intended.

```
def_list_value :: [def] -> env -> stores -> env

def_list_value [] en st = en
def_list_value (d:ds) en st
                = def_list_value ds (def_value d en st) st
```

This function is defined so as to add definitions one at a time, so that the right hand side of the second definition is interpreted in an environment already containing the first definition.

3.4.3 Procedures

At last we can return to procedure values. The function we concentrate on is

```
proc_value :: [ident] -> command -> env -> proc
```

The function `proc_value` returns a `proc`, that is a function, and we define the effect of a function by looking at how it behaves on an argument, `(st,alist)`:

```
proc_value flist c en (st,alist)
        = command_value c en' st'
          where
          en' = def_list_value dlist en st
          st' = ...
```

We can see this definition as having two stages. First we treat the formal parameter list (`flist`) as a list of definitions (`dlist`) and incorporate these into the environment `en`. We then initialise these new variables to the values of the actuals, `alist`, resulting in the state `st'`; this shows that we are indeed passing parameters by value. It is in the state `st'` that we execute the command `c`.

The full definition reads

```
proc_value flist c en (st,alist)
        = command_value c en' st'
          where

          dlist = map Variable flist
          en' = def_list_value dlist en st

          st' = initialise st flist_vals alist

          flist_vals
             = map (get en') flist
                where
                get x y = loc
                           where (Loc loc) = find y x

          initialise st [] [] = st
          initialise st (f:fs) (a:as)
                = initialise (update st f a) fs as
```

3.5 Defining the rest of the functions.

In this section we give definitions of `expr_value` and the store manipulation functions. The definitions of `b_expr_value`, `find`, `bind` etc. should present no difficulties for the reader (who could find them in the `full` directory if desperate!) We define `expr_value` from the auxiliary function

```
expr_value0 :: expr -> env -> stores -> num
```

which returns a `num` rather than the `Number num` returned by `expr_value`.

Constants are evaluated by finding them in the environment `en`:

```
expr_value0 (Con i) en st
   = val
     where
     Number val = (find i en)
```

Variables are evaluated by first finding their l-value,`loc`, in the environment and then looking up the value stored in `loc` by the store `st`.

```
expr_value0 (Var i) en st
   = val
     where
     Loc loc = (find i en)
     Number val = (lookup loc st)
```

Operators and literals are evaluated exactly as before, except for the extra `env` parameter which has to be passed inwards.

How are the store and environment manipulated? The definitions are standard. `update` , `lookup` have the obvious definitions:

```
update x i val = (i,val):x
lookup i [] = error "lookup"
lookup i ((j,val):x) = val , i=j
                     = lookup i x
```

`free` finds the first location not mentioned in the list, and outputs this as the new location.

3.6 Conclusion

This section has shown how we treat the semantics of block-structuring and abstraction in a Pascal-like language. Although there is some overhead in learning a functional programming language like Miranda, once we have made the investment we can quickly write a concise, unambiguous and executable version of a programming language of substantial power.

Again it is interesting to review the dependencies between the major semantic functions. As before in interpreting programs (which are commands) by means of the function `command_value`, we need to interpret expressions and boolean expressions, so we call `expr_value` and `b_expr_value` which itself calls `expr_value`. We also need to interpret definitions, through `def_value` which in turn requires us to call `expr_value` (to interpret constants) and `command_value` (to interpret procedures); we therefore have a set of functions defined by mutual recursion.

In the following section we shall discuss the semantics of other features in a rather more discursive way.

4 Further Denotational Semantics

In this section we look at the semantics of some rather more complex constructs. An implementation of the functions in the first two sub-sections can be found in `full`.

4.1 Parameter passing by reference

When we pass parameters by reference, we bind the names of the formals to the l-values (locations bound to) the actuals. This is, in fact, a simpler mechanism than that for value, as we don't need to perform any allocation — all we do is bind:

```
ref_proc_value :: [ident] -> command -> env -> proc
ref_proc_value flist c en (st,alist)
        = command_value c en' st
          where
          en' = list_bind en flist alist
list_bind en [] [] = en
list_bind en (f:fs) (a:as)
        = list_bind (bind f a en) fs as
```

On procedure call, that is when `ref_proc_value flist c en` is presented with the argument `(st,alist)`, we form an environment `en'` from `en` by binding the formals to the l-values of the actuals — the `list_bind` function achieves this. We then execute the body in this environment, together with the state from the call. An example of a swap procedure can be found in the distribution.

4.2 Recursive procedures (with reference parameters)

We interpret these exactly as we interpret reference procedures, except that in the environment in which we evaluate the body we want the procedure name already to be bound to its value — a clear case for a recursive description.

In interpreting non-recursive procedures we don't need to know the name to which the procedure is to be bound. Clearly we do in the recursive case, so we need to change the type of our interpretation function so that it takes this name as its first argument.

```
rec_proc_value :: ident -> [ident] -> command -> env -> proc
rec_proc_value id flist c en
        = val
```

`val` is a function, which itself is defined thus:

```
val (st,alist)
        = command_value c en'' st
          where
          en' = list_bind en flist alist
          en'' = bind id (Abstract val) en'
```

`en'` is defined in exactly the same way as for reference procedures — it is the result of binding the formals to the l-values of the actuals. `en''` results from it by binding the procedure name `id` to its value `val`, exactly as the informal description.

How would we interpret a collection of mutually recursive procedures? The same technique will work, except that in the interpretation of the body of each procedure, all the names have to be bound to their ultimate values.

4.3 Parameter passing by value-result

A variant of the reference parameter passing mechanism is value-result, in which we use the value mechanism, but assign the final values of the local variables to the l-values of the actuals on procedure termination. This variant is found in the directory `value_result`.

4.4 Dynamic binding

Recall the example procedure from Section 3

```
procedure joe(y) ;
  x := y + x ;
```

We mentioned there that the l-value of `x` is found in the environment in which `joe` is defined; we could alternatively find its l-value in the prevailing environment when `joe` is called. This is called dynamic binding; details are to be found in the directory `dynamic`.

4.5 Expressions with side-effects

One feature which necessitates us rebuilding our semantics is the `function` mechanism for Pascal-like languages. We can handle its abstraction aspect in exactly the same way as we handled procedures, but a difficulty arises because these functions, which in general cause state changes, are invoked by expression evaluation. This means that all expressions can potentially change machine state, so that we have to revise our evaluation function to one of type:

```
expr_value :: expr -> env -> stores -> (values,stores)
```

Even if we had avoided storage allocation in the last section we would now have to allow definitions to affect the state, as definitions usually involve expression evaluation.

Let us return to the simple store model of Section 2 for this discussion — we lose nothing (except notational complexity!) by so doing.

```
expr_value :: expr -> stores -> (values,stores)
```

What happens when we evaluate an expression with two subexpressions, like

```
(Apply_nop op e1 e2)
```

in `st`?
Given that evaluating either expression `e1,e2` will potentially change `st` we can't evaluate both in `st`, as we would then not know how to modify `st`. We must decide to do one before the other, and we have to realise that this choice will have an effect, as evaluating an expression in two different orders will often give two different results. Left to right order is given by:

```
expr_value (Apply_nop op e1 e2) st
        = ( val , st'' )
          where
          val = nop_value op v1 v2
          (v1,st')  = expr_value e1 st
          (v2,st'') = expr_value e2 st'
```

More details can be found in the directory `dirty`.

4.6 Jumps

In the directory `jump` we model a simple language containing `gotos` by taking a program to be a list of labelled commands,

```
program == [ (label,command) ]
```

where to the type of commands we add a jump thus

```
command ::= ... as earlier ... | Goto label
```

A command can now produce one of two outcomes. It can terminate normally, or it can jump to a label, so we define,

```
outcomes ::= Ok stores | Jump label stores
```

We then define the semantic functions; first for commands:

```
command_value :: command -> stores -> outcomes
```

We work as in Section 2, more or less. Example equations from the definition include

```
command_value Skip st = Ok st

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

command_value (Goto l) st = Jump l st
```

The top-level function in the semantics is

```
program_value :: program -> stores -> stores
```

where

```
program_value p = label_value p (first_label p)
```

It is only at the level of a program that we can describe a simple transition from `stores` to `stores` — we give a meaning to each label by giving a meaning to the command associated with it. A mutual recursion ensures that if the command terminates normally, we pass control to the next label, whilst if it terminates with a jump, we jump to that label. The crucial definition is, therefore

```
label_value :: program -> label -> stores -> stores
```

and

```
label_value p l st
    = follow (command_value (command_for p l) st) l
      where
      follow (Jump l' st') l
        = label_value p l' st'
      follow (Ok st') l
        = st'                    , if last l labels
        = label_value p (next l labels) st'     , otherwise
```

where various auxiliary functions manipulate labels in the obvious way.

This semantics shows the complexity of using jumps — we can only resolve the meaning of a whole program as a single unit, since from any point we can jump to any other.

4.7 Errors

Some programming errors, like division by zero, can only be detected at run-time, and we should explain how these are to be handled. There are really two approaches.

The first, simpler, technique is to abort on error. This can be modelled using the Miranda `error` function which causes a Miranda error. If we think about this more carefully, what this implies is that all calculations and computations containing errors are formally undefined. We can actually do better than that, and return an explicit error value. For instance we can define a type of numbers with an error value thus:

```
enum ::= OK num |
         Error
```

and redefine division on `enum`.

```
ediv n m = Error , n = Error \/
                   m = Error \/
                   m' = 0
         = OK (n'/m')
           where n = OK n'
                 m = OK m'
```

We output `Error` if either of the inputs is erroneous or if division by zero is attempted, otherwise we perform the ordinary division.

Because `Error` is a proper value, we can trap and handle it. Such a facility is going to be necessary in describing most languages, which use more sophisticated approaches than the "dump and run" of `error`.

5 Operational semantics

An alternative way of explaining the semantics of the simple language described in

```
basic/abstract_syntax.m
```

is to think of the effect of a command as a sequence of transitions between machine states. (The details for this section are found in the `basic_opr` directory.) We can think of the configuration of a machine as either having halted in a particular state, or as being in a particular state with a command to be executed. In Miranda,

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

Expressions are interpreted just as in Section 2. The function interpreting commands is

```
computation :: config -> [config]
```

and we get a computation by repeatedly taking the next step in the machine:

```
computation con
        = [con]                          , if isFinal con
        = con : computation (step con)   , otherwise
```

So, we have to define the function

```
step :: config -> config
```

The most important cases follow.

A skip command terminates immediately:

```
step (Inter Skip st)
        = Final st
```

In performing an alternative command, we choose the appropriate command to continue with:

```
step (Inter (If_Then_Else e c1 c2) st)
        = (Inter c1 st)         , if b_expr_value e st
        = (Inter c2 st)         , otherwise
```

In interpreting a loop, we say:

```
step (Inter (While_Do e c) st)
        = (Inter (If_Then_Else e (Sequence [c,While_Do e c]) Skip) st)
```

What is happening here? We explain one command in terms of another — an alternative between the two cases. In the first we do the command, followed again by the while loop; in the other case, we skip.

In interpreting a sequence of commands we have three cases:

```
step (Inter (Sequence (c:rest)) st)
        = Final stf                             , if isFinal con' & rest = []
        = (Inter (Sequence rest) stf)           , if isFinal con'
        = (Inter (Sequence (c':rest)) st')      , otherwise
          where
          con' =  step (Inter c st)
          (Final stf)    = con'
          (Inter c' st') = con'
```

In the first case we have one command which terminates in one step; in the second the first command terminates in one step, so we move to the remainder in the next step. In the final case, the first command gives rise to the configuration

```
(Inter c' st')
```

in one step, so we continue to execute `c'` from state `st'` before executing the commands in `rest`.

An assignment terminates in one step:

```
step (Inter (Assignment i e) st)
        = Final (update st i (expr_value e st))
```

The other cases of the function are straightforwardly defined.

6 Compiling

A simple stack machine and compiler for the `basic` language are given in the `machine` directory. The machine uses a stack of values — numbers, booleans and locations (addresses of positions in the store) — to calculate values of expressions, and has a store in which to record values.

The description in the directory consists of three components.

6.1 The code

The code for the machine is given by an algebraic data type in Miranda, contained in the file `code.m`.

```
m_code
       ::= Push_num num |
           Do_nop nop |
           Do_monop monop |
           Contents ident |
           Push_bool bool |
           Do_rel rel |
           Do_bop bop |
           Do_monbop monbop |
           Lval ident |
           Pop |
           Assign |
           Copy |
           Label label |
           Goto label |
           Gofalse label |
           Gotrue label |
           Halt
```

The `Push_num` places a number at the top of the stack. To 'do' an operator, the appropriate number of arguments are taken from the stack, the operator is applied to them and the result returned to the stack. The `Lval` instruction places a location at the top of the stack, and `Pop` removes the top item of the stack. The `Assign` instruction removes a location and value from the stack, and performs the appropriate assignment. The various jump operations replace the code to be executed by the code found by jumping to the appropriate label.

A formal version of this brief explanation is found in the next section, where we give a Miranda description of the machine.

6.2 The machine

The machine is described in `machine.m`. The stack of the machine is modelled as a list of items,

```
stack == [item]
```

where each item is either a number, a Boolean or a location:

```
item ::= N num | B bool | L ident
```

All the instructions, bar the jumps and halt, pass control to the next instruction in the instruction sequence. We can therefore explain their behaviour simply by explaining their effect on the stack and store. This is the purpose of the

```
execute_one :: ( m_code , stack , stores ) -> ( stack , stores )
```

function. In our definition, we follow the ordering of the algebraic type definition. To push a number, we say

```
execute_one ( Push_num n , sta , sto )
        = ( N n : sta , sto )
```

whilst a typical 'do' operation works thus, taking its arguments from the stack and returning the result there

```
execute_one ( Do_nop f , N v2 : N v1 : sta , sto )
        = ( N (nop_value f v1 v2) : sta , sto )
```

We get the contents of a location thus:

```
execute_one ( Contents ide , sta , sto )
        = ( N (lookup ide sto) : sta , sto )
```

and we assign to a variable as follows:

```
execute_one ( Assign , L ide : N n : sta , sto )
        = ( sta , update sto ide n )
```

A label is simply treated as a ‘null’ operation,

```
execute_one ( Label l , sta , sto )
        = ( sta , sto )
```

The other cases can easily be reconstructed by the reader, who can also find them in the file.

Program execution is modelled by

```
execute ::  [m_code] -> ( [m_code] , stack , stores ) ->
            ( [m_code] , stack , stores )
```

The triple `( [m_code] , stack , stores )` represents an intermediate configuration of the machine, consisting of

- a code sequence to be executed;
- a stack of values; and
- a state of the store.

The effect of executing the program is to give a final such configuration.

The interesting cases are given by the jumps. For instance,

```
execute pro ( (Gotrue l) : re , B b : sta , sto )
        = execute pro ( re’ , sta , sto )       , if b
        = execute pro ( re  , sta , sto )       , otherwise
          where
          re’ = follow l pro
```

In the case that the top of the stack is `True`, we make the jump; the function `follow` finds the target of the jump, and execution resumes there; in the other case, execution continues from the point after the jump instruction. `Goto` and `Gofalse` are modelled in a similar way. The `Halt` instruction halts execution, and any other instruction causes a single step of execution:

```
execute pro ( ins : re , sta , sto )
        = execute pro ( re , sta’ , sto’ )
          where
          ( sta’ , sto’ ) = execute_one ( ins, sta , sto )
```

Now we have a formal model of our machine, we can give the compilation algorithm for the `basic` language.

6.3 Compiling the language

To compile an expression we give the function

```
compile_expr :: expr -> [m_code]
```

A variable has its contents placed on the stack,

```
compile_expr (Var ide) = [ Contents ide ]
```

and a number is pushed:

```
compile_expr (Num n)    = [ Push_num n ]
```

The substantial case is of an operator; for instance, a binary operator gives the code:

```
compile_expr (Apply_nop f e1 e2)
        = compile_expr e1 ++ compile_expr e2 ++ [ Do_nop f ]
```

which we can see corresponds to the way we executed a numeric operator above. Compiling the other numeric and boolean expressions is similar.

The code for a skip instruction is null,

```
compile_command Skip = []
```

and for an assignment

```
compile_command (Assignment ide e)
        = compile_expr e ++ [ Lval ide , Assign ]
```

In compiling structured commands such as `If_Then_Else` we produce code containing jumps. For example,

```
compile_command (If_Then_Else be c1 c2)
        = compile_b_expr be ++ [ Gofalse newlab ] ++
          compile_command c1 ++ [ Goto newlab' , Label newlab ]
          ++ compile_command c2 ++ [ Label newlab' ]
```

We first evaluate the condition, hence the code

```
compile_b_expr be
```

If the result is false, we jump to the code for the 'else' case , `c2` — hence the `Gofalse`; otherwise, we do the code for the 'then' case, and jump over the `c2` code to `newlab'`.

For the `While_Do` we have

```
compile_command (While_Do be c)
        = [ Label newlab ]  ++ compile_b_expr be ++ [ Gofalse newlab' ] ++
          compile_command c ++ [ Goto newlab , Label newlab' ]
```

and a similar translation gives the code for the Repeat_Until `command.`

We have glossed over exactly how the new labels `newlab` etc. are supplied to the compiler. In fact our function is of type

```
compile_command :: command -> labeltree -> [m_code]
```

where the second argument is a tree of labels, generated separately to provide the appropriate labels. In the case of the `If_Then_Else` command, the labeltree will be

```
Iftree newlab newlab' tree1 tree2
```

providing the two new labels and the trees to be used in the component commands. The definition then reads

```
compile_command (If_Then_Else be c1 c2)
                (Iftree newlab newlab' tree1 tree2)
        = compile_b_expr be ++ [ Gofalse newlab ] ++
          compile_command c1 tree1 ++ [ Goto newlab' , Label newlab ] ++
          compile_command c2 tree2 ++ [ Label newlab' ]
```

in which it can be seen how the labels are distributed. Full details are to be found in the `compile.m` file.

6.4 Conclusion

Putting the three files together, we have a machine to compile and execute our programs. We can compare this with the semantics given earlier, and indeed we can prove that the implementation is correct. For example, we can prove that executing the code for an expression results in a stack with the value of the expression at the top — the proof is by structural induction over the complexity of the expressions.

In a similar way we can prove that our commands are implemented correctly; in this case fixed-point induction has to be used.

7 On the approach of this work

The approach to explaining semantics which we have introduced here has a number of advantages.

- The semantics is presented in a familiar language. In presenting the operational semantics given in Section 5 I was surprised to find that students were happier with the Miranda description rather than the more abstract and less cluttered rule-based version. Each different type of formal language imposes a learning overhead, and so it can be more effective to use a familiar language, even at some cost in elegance.

- The semantics is type checked and executable – we can interact with it and can therefore be sure that it works as we wish. We can validate it, in other words. Again, for students, it is a bonus to be able to experiment with the semantics as well as to read it. From another point of view, the approach allows us to prototype language features.

- The semantics makes a clear distinction between the functional description of the imperative (which this semantics gives) and a domain-theoretic description of recursion. Other approaches can tend to conflate the two, leading to confusion.

Clearly there are disadvantages here too. The approach leaves some issues implicit – what the exact form that product (tuple) types take is a case in point – and it can also make it difficult to specify clearly or abstractly certain parts of languages. Notwithstanding this, we feel that it provides a useful learning tool in explaining abstract ideas in an approachable way.

Appendix – Explaining Recursion

The aim of this appendix is to show how recursion can be given a sensible, mathematical explanation. The essence of a recursive description is that it uses the object being defined within the description itself: it is self-referential and self-referential definitions can be troublesome.

> In a certain town every man is clean shaven. Some men shave themselves and others are shaved by (male) barbers. One barber is special, in that he shaves every man who does not shave himself (and no others). Who shaves the barber?

If the barber shaves himself, then by his description he must be someone who does not shave himself, yet if he is such a person, then he must, by his description, shave himself! This is one form of Russell's paradox, and it illustrates the problems that self-reference can generate.

The way that we interpret recursion is by producing a system of approximations to the object defined. We are familiar with this technique from the numerical solution of equations, and so we first recall what we do there.

We look at equations of the form

$$x = f(x) \qquad (1)$$

such equations form a larger class than might be imagined at first sight; we can rewrite

$$x^2 = 2$$

as

$$x = (x^2 + 2)/2x \qquad (2)$$

for example. Note the similarity of this to a recursion equation. It defines a solution(x)in terms of itself ($(x^2 + 2)/2x$) — the only difference is in the kind of object that we are defining: here we define a number , or collection of numbers, whereas a recursion equation usually defines a function.

How do we solve something like (2)? As we said, we use (2) to give us a sequence of approximations, thus:

$$x_0 = start$$
$$x_{n+1} = (x_n^2 + 2)/2x_n$$

and in general for an equation like (1),

$$x_0 = start$$
$$x_{n+1} = f(x_n)$$

The solution we generate is given by the limit of the sequence $<x_n>_n$. The equation (2) gives a sequence of approximations to the square root of 2. The solution itself is none of these approximations — it is an infinite decimal.

Now, how are we to ‘‘solve’’ or explain a recursive definition like

```
fac p = 1  ,  p <= 0
      = p * fac (p-1)
```

or

```
qsort []    = []
qsort (a:x) = qsort (less a x) ++
              [a] ++
              qsort (more a x)
```

where `less,more` are auxiliary functions with the obvious definitions?
We do it in exactly the same way that we did for our square root. We take fac_0 $qsort_0$ to be suitable starting values (more below) and generate successive approximations thus:

```
fac_{n+1} p = 1  ,  p <= 0
            = p * fac_n (p-1)

qsort_{n+1} []    = []
qsort_{n+1} (a:x) = qsort_n (less a x) ++
                    [a] ++
                    qsort_n (more a x)
```

We use the definition to generate the next approximation from the previous one, and we hope that the sequence takes us to a solution of the equations. What should our starting functions be? The obvious choice is to start out with nothing, in other words we take fac_0 $qsort_0$ to be completely undefined: whenever we try to find one of their values, our computation does not terminate, (we go into a ‘‘black hole’’, as it were.)
What do the approximations look like?

```
fac_0 p = undef
```

```
fac₁ p = 1  ,  p <= 0
       = p * fac₀ (p-1)
```

so

```
fac₁ p = 1  ,  p <= 0
       = undef

fac₂ p = 1  ,  p <= 0
       = p * fac₁ (p-1)
```

so

```
fac₂ p = 1  ,  p <= 0
       = 1  ,  p = 1
       = undef
```

Similarly,

```
fac₃ p = 1  ,  p <= 1
       = 2  ,  p = 2
       = undef
```

and so on.
We can see that

- The approximations never disagree on the values that they do define — it is only that some approximations may give a value and others not.
- The further we go along our approximation sequence, the more information we derive. (Strictly, we should make the weaker claim that we never lose information.)

These properties can be proved to hold of any sequence of approximations thus generated, and the properties are sufficient to imply that there is a unique smallest function defined by a recursion equation. Formally we define it by saying

```
fac k = l
```

if and only if for some `n` ,

```
facₙ k = l
```

In our analogy of equation solving we were led to the view of a real number, like the square root of 2, being given by a sequence of approximations. Based on this idea, we build the model of the real numbers given by Cauchy or Dedekind. In a similar way, we can build a mathematical structure of functions, called a domain, out of our function approximations and their ‘‘limits’’, which we obtain from sequences like $\langle \mathtt{fac}_n \rangle_n$. A gentle introduction to this work is provided by [Schmidt] or [Winskel].

We have show that there is a sensible way for us to give meaning to the recursion equations which form the heart of Miranda and therefore of our semantics. The techniques we have used are based on numerical techniques, and make sense of ‘‘nonsensical’’ definitions like

```
f x = f (x+1)
```

or our ‘definition’ of the barber — the objects in question are simply `undef`ined, a situation which we are used to in computing.

Bibliography

Schmidt, D.A., *Denotational Semantics*, Allyn and Bacon, 1986.

Tennent, R.D., *Principles of Programming Languages*, Prentice-Hall, 1981.

Thompson, S.J., *Miranda The Craft of Functional Programming*, Addison-Wesley, 1995.

Winskel, G., *The Formal Semantics of Programming Languages*, MIT Press, 1993.