

TOWARDS A CALCULUS OF LAZY INTERPRETERS

R A Frost and W C Saba, School of Computer Science, University of Windsor Ont N9B 3P4

Some time ago, Knuth suggested that executable attribute grammars might provide a viable declarative programming language [5]. More recently, a number of researchers have argued convincingly for a style of programming that focuses attention on a small collection of powerful higher order functions which capture common patterns of computation. Programs are expressed as compositions of partial applications of these functions. Some of the higher order functions may be defined recursively, but once that is done further explicit use of recursion and induction is avoided. This approach results in variable-free or nearly variable-free programs that may be readily transformed to more efficient forms using algebraic identities associated with the higher order functions. Bird [1] refers to this style as program 'calculation'. An integration of these two ideas would result in a calculus in which interpreters are values. Some initial progress has been made in developing such a calculus. A number of operators have been developed, some laws have been established, and a few interpreters have been constructed. The operators allow lazy, fully context-sensitive interpreters to be constructed as executable specifications. Interpreters may be composed resulting in a 'pipeline' style of programming.

1 INTRODUCTION

A number of functions have been developed that may be regarded as operators of a calculus whose values include interpreters. These operators have been implemented in the higher order lazy functional programming language Miranda* [6]. We begin by describing the calculus and then give examples of its use. We conclude with a discussion of planned future work.

2 A CALCULUS OF INTERPRETERS

In this section, we present an informal description of a first attempt at a calculus of interpreters. Formal definitions of the operators are given in Appendix A.

All definitions are given in a functional notation that requires only minor changes to be run in concrete lazy functional languages. The particular changes required to convert the definitions to executable Miranda code are given in Appendix B.

2.1 The Type Structure

2.1.1 The primitive types. There are two primitive types : `terminal` and `attribute`. Both of these types are primitive with respect to the calculus. However, for a given application, they are introduced using an algebraic type definition, eg :

```
terminal ::= CHARTERM char | WORDTERM [char] | NUMTERM num
attribute ::= VAL num | RES num | NOUNVAL [num] | SORT [char]
```

The first definition introduces four identifiers : the type-name `terminal` , and the constructors `CHARTERM`, `WORDTERM` and `NUMTERM`, which are of the following types respectively :

```
char    -> terminal
[char]  -> terminal
num     -> terminal
```

Given such definitions, the following are examples of values of type `terminal` and `attribute` respectively :

```
WORDTERM "two"          VAL 2
```

In order to simplify the following, we shall assume that all terminals are made from character strings. Under this assumption, there is no need to introduce the constructor `WORDTERM` and the type `terminal` can be introduced by the following type-synonym declaration :

```
terminal == [char]
```

* Miranda is a Trademark of Research Software Ltd.

2.1.2 The basic type structure

- If T is type, then $[T]$ is the type of lists whose elements are of type T .
- If T_1 to T_n are types, then (T_1, \dots, T_n) is the type of tuples with objects of these types as components.
- If T_1 and T_2 are types, then $T_1 \rightarrow T_2$ is the type of functions with arguments in T_1 and results in T_2 .

2.1.3 The type of interpreters. All interpreters are of type :

$[[[attribute], [terminal]]] \rightarrow [[[attribute], [terminal]]]$

That is, an interpreter is a function which maps a list of pairs of type $([attribute], [terminal])$ to a list of pairs of the same type.

Explanation :

1. Let us call a value of type $([attribute], [terminal])$ an 'atts_terms_pair'.
2. Each atts_terms pair (as , ts) that is in the list that is input to an interpreter is such that :
 - as is a context,
 - ts is a sequence of terminal symbols to be interpreted in the context as .
3. Each atts_terms_pair (as' , ts') in the list that is output by an interpreter is related to a single pair (as , ts) in the input such that :
 - as' = a subset of the union of as and the interpretation of some initial segment of ts ,
 - ts' = a list of the remaining uninterpreted terminal symbols in ts .
4. Interpreters return lists of atts_terms_pairs because a sequence of terminals may have more than one interpretation.
5. We have chosen to regard interpreters as accepting lists of atts_terms pairs for a number of reasons, one being that it simplifies their composition.

2.2 The interpreters fail and succeed

There are two distinguished interpreters, `fail` that always fails and `succeed` that always succeeds. The interpreter `fail` returns the empty list for all inputs. The interpreter `succeed` always returns its input as result.

2.3 Lists and Tuples

Lists of expressions may be made using square brackets and commas eg :

```
[VAL 1, VAL 2, VAL 3]
```

Lists may be appended using the polymorphic operator `++` which is of type $[*] \rightarrow [*] \rightarrow [*]$ eg :

```
[VAL 1, VAL 2] ++ [VAL 3] => [VAL 1, VAL 2, VAL 3]
```

The elements of a list must all be of the same type. A sequence of elements of mixed type is called a tuple, and is written using parentheses eg :

```
("one", [VAL 1])
```

2.4 The Operators

2.4.1 The operator term is of type :

$(terminal, [attribute]) \rightarrow interpreter$

That is, `term` takes as input a pair, consisting of a single terminal symbol followed by a list of attributes (the meaning of the terminal symbol), and returns an interpreter for that terminal symbol.

Example definitions of interpreters

```
us_bill = term ("billion", [VAL 10^9, DERIV "USA"])
uk_bill = term ("billion", [VAL 10^12, DERIV "UK"])
```

Example use of interpreters

```
us_bill [[[]], ["billion", "."]] => [[(VAL 10^9, DERIV "USA"), ["."])]
uk_bill [[[]], ["two", "."]]    => []
```

2.4.2 The operator ! is of type :

```
terminal -> interpreter
```

The operator ! takes a single terminal symbol as input and returns an interpreter that recognises that terminal symbol but does not interpret it. The attributes in the input to the interpreter, ie the 'inherited' context, are simply copied into the output from the interpreter.

Example definition of an interpreter

```
clbr = !")"
```

Example use of the interpreter

```
clbr [[(VAL 10), [")", "xx"]]] => [[(VAL 10), ["xx"])]
```

Notice that contexts such as [VAL 10] 'pass unchanged' through interpreters that are constructed using !.

2.4.3 The operator | is of type :

```
interpreter -> interpreter -> interpreter
```

The operator | is best described by giving its formal definition :

```
(p | q) input = p input ++ q input
```

That is, | takes two interpreters as input and returns an 'alternate' interpreter as result. This alternate interpreter applies both of the component interpreters to the input and appends their results.

Example definition of an interpreter

```
us_or_uk_bill = us_bill | br_bill
```

Example use of the interpreter

```
us_or_uk_bill [[[]], ["billion", "xx"]] => [[(VAL 10^9, DERIV "USA"), ["xx"]],
                                             [(VAL 10^12, DERIV "UK"), ["xx"])]
```

2.4.4 The operator excl_ is similar to | except that it only applies the second interpreter if the first fails. The operator excl_ can be used in place of | to improve the efficiency of certain types of interpreter.

2.4.5 The operator @ is of type :

```
interpreter -> interpreter -> interpreter
```

The operator @ composes its arguments in the opposite order to that which is usual in mathematical notation.

Example definition of an interpreter

```
br_numb = ! "(" @ numb @ ! ")"
```

Example use of the interpreter

```
br_numb [[[]], ["(", "2", ")", "xx"]] => [[(VAL 2), ["xx"])]
```

Such combinations of ! and @ can be used to 'peel' off those terminals in the input that serve only as syntactic markers. However, @ has other uses. For example, it allows interpreters to be pipelined. When a composite interpreter p1 @ p2 is given some input inp, the interpreter p1 is applied to it, then the results of this application are fed into p2. In particular, the attributes that are synthesised by p1 constitute the contexts for p2. Whatever terminal symbols in inp that are not 'consumed' by p1 are examined and, if recognised, are interpreted by p2.

So far, we have not considered any context dependent, or context sensitive, interpreters. We now introduce a construct that allows such interpreters to be defined.

2.5 Specification of Executable Attribute Grammars

Interpreters can be defined as executable attribute grammars. The construct used for doing this involves the symbol ψ followed by a 'production' then a set of 'attribute rules' :

1. The 'production' denotes the identity and the order of application of the component interpreters.
2. The 'attribute rules' denote :
 - a. how the inherited context attributes of the 'component' interpreters are to be calculated from the context attributes of the interpreter being defined together with the synthesised attributes that are returned by any of the other component interpreters,
 - b. how the attributes, that are to be returned by the interpreter being defined, are to be synthesised from the attributes in the context given as input to the interpreter and the synthesised attributes of any of the component interpreters.

The following are examples of the use of the ψ construct :

Example definitions of interpreters

```
add_two_nums =  $\psi$  (s1 numb ... s2 numb)
               [rule 2.1 (RES  $\uparrow$  lhs)  $\leftarrow$  add [VAL  $\uparrow$  s1,
                                                       VAL  $\uparrow$  s2]]

context_num =  $\psi$  (s1 numb)
               [rule 2.2 (RES  $\uparrow$  lhs)  $\leftarrow$  add [VAL  $\uparrow$  s1,
                                                       VAL  $\downarrow$  lhs]]

add_two_nums' =  $\psi$  (s1 numb ... s2 context_num)
                 [rule 2.3 (RES  $\uparrow$  lhs)  $\leftarrow$  same [RES  $\uparrow$  s2],
                 rule 2.4 (VAL  $\downarrow$  s2)  $\leftarrow$  same [VAL  $\uparrow$  s1]]

attribute functions
add [VAL x, VAL y] = RES (x + y)
same [x]           = x
```

These definitions may be read as follows :

1. The interpreter `add_two_nums` recognises a structure that consists of a sub-structure `s1` of 'type' `numb` followed by a sub-structure `s2` of 'type' `numb`. By semantic rule 2.1, the `RES` attribute that is synthesised (\uparrow) for the left hand side `lhs` (ie returned by `add_two_nums`), is equal to the result obtained by applying the attribute function `add` to a list of attributes containing the `VAL` attributes that were synthesised for the numbs `s1` and `s2`.
2. The interpreter `context_num` recognises a structure that consists of a structure `s1` of type `numb`. By semantic rule 2.2, the `RES` attribute that is synthesised for the left hand side (ie returned by `context_num`), is equal to the result of applying the attribute function `add` to a list of attributes containing :
 - a. the `VAL` attribute that was synthesised (\uparrow) for the `numb s1`,
 - b. the `VAL` attribute that was passed down as context (\downarrow) to the interpreter `context_num`.
3. The interpreter `add_two_nums'` recognises a structure that consists of a structure `s1` of type `numb` followed by a structure `s2` of type `context_num`.
 - a. By semantic rule 2.3, the `RES` synthesised for the left hand side `lhs` (ie returned by `add_two_nums'`), is equal to the result of applying the attribute function `same` to a list of attributes containing the `VAL` attribute that was synthesised for the `context-numb s2`.
 - b. By semantic rule 2.4, the `VAL` attribute inherited as context by the `numb s2` is equal to the result of applying the attribute function `same` to a list of attributes containing the `VAL` attribute that was synthesised for the `numb s1`.

Note that :

- The interpreters `add_two_nums` and `add_two_nums'` are equivalent to the interpreter `add_two_nums''` defined as follows :


```
add_two_nums'' = numb @ context_num
```


- All attribute functions are of type :
[attribute] -> attribute
ie they are all functions from lists of attributes to a single attribute.
- The order of the attribute rules within a definition is irrelevant.
- If the 'names' of the synthesised and inherited attributes are disjoint, then \uparrow and \downarrow can both be replaced by \dagger which is to be read as 'of'. This removes all procedurality in the definitions excepting that the attribute rules are used as left to right re-writes.

Example use of interpreters

```
add_two_nums [[[]],["1","2","."]] => [[(RES 3),["."]]]
context_numb [[(VAL 10),["2","."]]] => [[(RES 12),["."]]]
```

Although we have not given an example here, an inherited (context) attribute for an interpreter can be defined in terms of synthesised attributes returned by interpreters to its right in the production. That is, context sensitive interpreters may be constructed using ψ . There are no restrictions on the attribute dependencies that are allowed. It is up to the user to ensure that definitions do not contain circular dependencies. The only restriction in ψ structures is that the productions must not be left recursive.

2.6 Binding Powers

The following lists the operators in order of decreasing binding power :

Operator	Comments
term	prefix
!	prefix
	associative and commutative
excl_	associative and commutative
++	associative
@	associative

Function application is denoted by juxtaposition and is left associative. Function application is more binding than any operator.

2.7 Laws and Equivalences

The following are examples of laws and equivalences that hold in the calculus :

1. succeed is a left and right identity for @.
2. fail is a left and right identity for both | and excl_|.
3. For all p1 and p2,

$$p \mid q = p \text{ excl_} | \ q \quad ++ \quad q \text{ excl_} | \ p$$

4. For all interpreters p0, p1, p2, p3, p4, all attribute constructors A1, A2, A3, all attribute functions f, and all numbers k, l, m and n, the following equivalence holds :

```
p2 = p4
where
p2 =  $\psi$  (s1 p0 ... s2 p1)
      [rule k (A1  $\uparrow$  lhs) <- f [A2  $\uparrow$  s1, A3  $\uparrow$  s2]]
p4 =  $\psi$  (s1 p0 ... s2 p3)
      [rule l (A1  $\uparrow$  lhs) <- same[A1  $\uparrow$  s2],
      [rule m (A2  $\downarrow$  s2) <- same[A2  $\uparrow$  s1]]
p3 =  $\psi$  (s1 p1)
      [rule n (A1  $\uparrow$  lhs) <- f[A2  $\downarrow$  lhs, A3  $\uparrow$  s1]]
```

5. For all interpreters p, lists of attributes atts, terminals t, and lists of terminals ts :

$$(!t \ @ \ p)[(atts, (t:ts))] = p[(atts, ts)]$$

2.8 Auxiliary Functions and Abbreviations

The following are examples of useful auxiliary functions and abbreviations.

2.8.1 The function `mkint` is of type :

```
[terminal, [attribute]] -> interpreter
```

The input to `mkint` is a 'dictionary' comprising a list of pairs, each of which consists of a single terminal symbol followed by a list of attributes (the meaning of the terminal symbol), and returns as result an interpreter for that dictionary.

Example definition of an interpreter

```
numb = mkint [("one", [VAL 1]), ("two", [VAL 2]),
              ("billion", [VAL 10^9]), ("billion", [VAL 10^12])]
```

Example use of the interpreter

```
numb [{"two", "xx"}] => [{"VAL 2", ["xx"]}
numb [{"billion", "xx"}] => [{"VAL 10^9", ["xx"]},
                             {"VAL 10^12", ["xx"]}]
```

Interpreters that are constructed using `mkint` return a list of results, one for each terminal in the dictionary that is recognised.

2.8.2 The function `words` is of type :

```
[char] -> [terminal]
```

This function takes a list of characters and returns a list of terminals. For example :

```
words "man and dog" => ["man", "and", "dog"]
```

3 EXAMPLE 1 : A THEOREM PROVER FOR PROPOSITIONAL LOGIC

From now on, we shall refer to a program constructed in the calculus as a 'passage', standing for "program constructed as a specification of an attribute grammar".

The following example is a complete executable specification of a theorem prover for propositional logic. The passage is constructed using the operators of our calculus. The theorem prover works as follows : the expression that is entered is converted to clausal form by pushing the context, positive or negative, down to the atomic formulas, which are made into negative or positive literals according to the context. These literals are then made into conjunctive clause form sets containing a single disjunctive clause consisting of the single literal. The clause form sets are then conjoined or disjoined depending on the context in which they appear. Tautologous clauses are removed as they are formed. If and only if the expression given as input is valid, will the clause form set returned be empty.

```
// PASSAGE#1
// WE BEGIN BY DECLARING THE ATTRIBUTE AND TERMINAL TYPES
attribute ::= CCFSET [disjclause] | CONTEXT [char] | VAL [char]
disjclause ::= DISJCL [literal]
literal    == [char]

// This means that there are three types of attribute :
// (1) CCFSET (conjunctive clause form set) attributes. These are made from
// lists of disjclauses (disjunctive clauses) by applying the attribute
// constructor CCFSET
// (2) CONTEXT attributes. These are made from lists of characters by applying
// the attribute constructor CONTEXT
// (3) VAL attributes. These are made from lists of characters by applying
// the attribute constructor VAL
// Objects of type disjclause are made from lists of literals by applying
// the constructor DISJCL
```


``` // DEFINITION OF THE INTERPRETERS ```

```
var      = mkint      [{"a",[VAL "a"]}, {"b",[VAL "b"]}, {"c",[VAL "c"]},
                      {"p",[VAL "p"]}, {"q",[VAL "q"]}, {"r",[VAL "r"]},
                      {"s",[VAL "s"]}, {"t",[VAL "t"]}, {"u",[VAL "u"]}]]

// Note that we could have used an auxiliary function here rather
// than list all terminals of type var

wff      =  $\psi$       (s1 expr ... s2 !"")
                  [rule 1.1 (CCFSET  $\uparrow$  lhs)      <- same_as [CCFSET  $\uparrow$  s1],
                  rule 1.2 (CONTEXT  $\downarrow$  s1)      <- pos_context []]

expr      =  $\psi$       (s1 var)
                  [rule 2.1 (CCFSET  $\uparrow$  lhs)      <- make_ccfset [VAL  $\uparrow$  s1,
                  |      (!"(" $o compound $o !"")" )      CONTEXT  $\downarrow$  lhs]]

                  |  $\psi$  (s1 !"- " ... s2 expr)
                  [rule 2.1 (CCFSET  $\uparrow$  lhs)      <- same_as [CCFSET  $\uparrow$  s2],
                  rule 2.2 (CONTEXT  $\downarrow$  s2)      <- opposite [CONTEXT  $\downarrow$  lhs]]

compound = conj | disj | implication

conj      =  $\psi$       (s1 expr ... s2 !"and" ... s3 conj)
                  [rule 3.1 (CCFSET  $\uparrow$  lhs)      <- context_and [CONTEXT  $\downarrow$  lhs,
                  |      CCFSET  $\uparrow$  s1,
                  |      CCFSET  $\uparrow$  s3],
                  rule 3.2 (CONTEXT  $\downarrow$  s1)      <- same_as [CONTEXT  $\downarrow$  lhs],
                  rule 3.3 (CONTEXT  $\downarrow$  s3)      <- same_as [CONTEXT  $\downarrow$  lhs]]
                  |  $\psi$  (s1 expr)
                  [rule 4.1 (CCFSET  $\uparrow$  lhs)      <- same_as [CCFSET  $\uparrow$  s1],
                  rule 4.2 (CONTEXT  $\downarrow$  s1)      <- same_as [CONTEXT  $\downarrow$  lhs]]

disj      =  $\psi$       (s1 expr ... s2 !"or" ... s3 disj)
                  [rule 5.1 (CCFSET  $\uparrow$  lhs)      <- context_or [CONTEXT  $\downarrow$  lhs,
                  |      CCFSET  $\uparrow$  s1,
                  |      CCFSET  $\uparrow$  s3],
                  rule 5.2 (CONTEXT  $\downarrow$  s1)      <- same_as [CONTEXT  $\downarrow$  lhs],
                  rule 5.3 (CONTEXT  $\downarrow$  s3)      <- same_as [CONTEXT  $\downarrow$  lhs]]
                  |  $\psi$  (s1 expr)
                  [rule 6.1 (CCFSET  $\uparrow$  lhs)      <- same_as [CCFSET  $\uparrow$  s1],
                  rule 6.2 (CONTEXT  $\downarrow$  s1)      <- same_as [CONTEXT  $\downarrow$  lhs]]

implication =  $\psi$       (s1 expr ... s2 !"implies" ... s3 expr)
                  [rule 7.1 (CCFSET  $\uparrow$  lhs)      <- context_or [CONTEXT  $\downarrow$  lhs,
                  |      CCFSET  $\uparrow$  s1,
                  |      CCFSET  $\uparrow$  s3],
                  rule 7.2 (CONTEXT  $\downarrow$  s1)      <- opposite [CONTEXT  $\downarrow$  lhs],
                  rule 7.3 (CONTEXT  $\downarrow$  s3)      <- same_as [CONTEXT  $\downarrow$  lhs]]

// ATTRIBUTE FUNCTIONS : functions of type [attribute] -> attribute
same_as   [x]      = x

pos_context []      = CONTEXT "pos"

context_and [CONTEXT "pos", x, y] = ccf_and x y
```

```

context_and [CONTEXT "neg", x, y] = ccf_or x y

context_or [CONTEXT "pos", x, y] = ccf_or x y
context_or [CONTEXT "neg", x, y] = ccf_and x y

opposite [CONTEXT "pos"] = CONTEXT "neg"
opposite [CONTEXT "neg"] = CONTEXT "pos"

make_ccfset [VAL v, CONTEXT "pos"] = CCFSET [DISJCL [ v]]
make_ccfset [VAL v, CONTEXT "neg"] = CCFSET [DISJCL [ negate_lit v ]]

// FUNCTIONS OF CLAUSAL FORM

// The function ccf_and forms the clausal conjunction of two conjunctive
// clause sets by simply uniting the disjunctive clauses in them.

ccf_and (CCFSET dcs) (CCFSET dcs') = CCFSET (mkset (dcs ++ dcs'))

// The function ccf_or forms the clausal disjunction of two conjunctive
// clause sets; tautologous clauses are removed as they are produced

ccf_or (CCFSET dcs) (CCFSET dcs') =
  CCFSET(mkset[newclause ! ( DISJCL c1 )<- dcs; ( DISJCL c2 )<- dcs';
    newclause <- [DISJCL (unite_clauses c1 c2)]; not_taut newclause])

not_taut (DISJCL c) = [l1 : l1 <- c; l2 <- c; l1 = (negate_lit l2)] = []

// The notation above is similar to set comprehension in set theory with
// '&' replaced by ';' and 'set membership' replaced by '<-'

negate_lit ('-' : x) = x
negate_lit y = ('-' : y)

unite_clauses c1 c2 = (sort . mkset) (c1 ++ c2)
// The operator '.' is function composition

// THE TOP LEVEL INTERPRETER
convert s = wff ([[]], (words s))

```

The following example illustrates how the function convert can be used to convert expressions to clausal form :

```

convert "--((p implies q) implies (r implies (s and t)))."
=> [[([CCFSET [DISJCL ["-r","p","s"],
  DISJCL ["-r","p","t"],
  DISJCL ["-q","-r","s"],
  DISJCL ["-q","-r","t"]]], [])]

```

The following example illustrates how the function convert can be used to test for validity. In general, if the formula to be converted is valid, it will be converted into the empty conjunctive clause form set. Therefore, to see if a formula F is a theorem of a theory which has a set of formulas S as proper axioms, you simply apply convert to "(S implies F)". If an empty clause set is returned, then F is a theorem of the propositional theory which has S as proper axioms, otherwise it is not :

```

convert "((p implies q) and p) implies q)." => [[([CCFSET [], [])]

```

4 EXAMPLE 2: A ONE PASS TREE PROCESSOR

The problem is to construct a program that is to accept a binary tree as input and which returns as result an equivalent tree except that all node values are equal to the maximum value in the original tree. For example :

```

"(3 (1 (4 - -) (5 - -)) (3 (4 - -) (2 - -)))."
=> "(5 (5 (5 - -) (5 - -)) (5 (5 - -) (5 - -)))"

```


With a little change in perspective, this problem can be thought of as a language processing problem. The input tree can be thought of as a sentence in some appropriately defined input language, and the output tree can be thought of as the result of interpreting the input tree. In the following, we construct a passage which takes a binary tree as input and which, in a single pass over the tree, returns the required result. This example, demonstrates how lazy evaluation and the general attribute dependencies that are allowed when using ψ can be used to create completely declarative executable specifications of programs that are often regarded as being inherently procedural.

```
// PASSAGE#2
// WE BEGIN BY DEFINING THE ATTRIBUTE AND TERMINAL TYPES
attribute ::= VAL num | RESULT [char] | MAX num | REPVAL num

// WE NOW DEFINE THE INTERPRETERS

item = mkint(["1", [VAL 1]], ("2", [VAL 2]),
             ("3", [VAL 3]), ("4", [VAL 4]),
             ("5", [VAL 5]) ]
// We could have defined and used an auxiliary function to define
// item rather than list all of the terminals

rooted_tree =  $\psi$  (s1 tree)
               [rule 1.1 (RESULT  $\uparrow$  lhs) <- same_as [RESULT  $\uparrow$  s1],
               rule 1.2 (REPVAL  $\downarrow$  s1) <- convert [MAX  $\uparrow$  s1]]

tree =  $\psi$ 
      (s1 !"-")
      [rule 2.1 (MAX  $\uparrow$  lhs) <- constant_0 [],
      rule 2.2 (RESULT  $\uparrow$  lhs) <- null_tree []]

      |  $\psi$  (s1 ! "(" ...s2 item...s3 tree...s4 tree...s5 !")")
      [rule 3.1 (MAX  $\uparrow$  lhs) <- maximum [VAL  $\uparrow$  s2,
                                         MAX  $\uparrow$  s3,
                                         MAX  $\uparrow$  s4],
      rule 3.2 (RESULT  $\uparrow$  lhs) <- make_tree [REPVAL  $\downarrow$  lhs,
                                             RESULT  $\uparrow$  s3,
                                             RESULT  $\uparrow$  s4],
      rule 3.3 (REPVAL  $\downarrow$  s3) <- same_as [REPVAL  $\downarrow$  lhs],
      rule 3.4 (REPVAL  $\downarrow$  s4) <- same_as [REPVAL  $\downarrow$  lhs]]

// We COMPLETE THE SPECIFICATION BY DEFINING THE ATTRIBUTE EVALUATION FUNCTIONS
same_as [x] = x
convert [MAX x] = REPVAL x
constant_0 [] = MAX 0
null_tree [] = RESULT "-"
maximum [VAL x, MAX y, MAX z] = MAX (max [x, y, z])
make_tree [REPVAL x, RESULT y, RESULT z]
          = RESULT "(" ++ (show x) ++ " " ++ y ++ " " ++ z ++ ")"
```

We can explain this completely declarative executable specification as follows :

1. A rooted_tree is a tree. Rule 1.1 states that the synthesised RESULT attribute that is associated with a rooted_tree is equal to the synthesised RESULT attribute associated with the tree that comprises the rooted tree.
2. Rule 1.2 states that the inherited replacement value, ie the REPVAL attribute that is inherited, by a tree that comprises a rooted_tree is equal to the synthesised maximum value that is associated with that tree, ie the MAX attribute.
3. A tree can be either a null_tree or can comprise an open bracket followed by an item, two component trees and a closing bracket. Rule 2.1 states that if a tree is a null tree, the synthesised MAX attribute associated with it is MAX 0. Rule 2.2 states that if a tree is a null tree, the synthesised RESULT attribute associated with it is RESULT "-".
4. If a tree is not a null_tree, then according to rule 3.1, the synthesised MAX attribute associated with it is equal to the result obtained by applying the function maximum to the synthesised VAL attribute associated with the

item and the synthesised MAX attributes associated with the two component trees.

5. If a tree is not a null_tree, then according to rules 3.2, the synthesised RESULT attribute associated with it is equal to the result obtained by applying the function make_tree to the inherited REPVAL attribute associated with the tree and the synthesised RESULT attributes associated with the two component trees.
6. If a tree is not a null_tree, then according to rules 3.3 and 3.4, the inherited REPVAL attribute associated with its two component sub-trees are both equal to the inherited REPVAL attribute associated with the tree itself.

The following is an example of an application of the interpreter rooted_tree :

```
rooted_tree [[[[], words "(3 (1 (4 - -) (5 - -)) (3 (4 - -) (2 - -)))".]]]
=> [[([RESULT "(5 (5 (5 - -) (5 - -)) (5 (5 - -) (5 - -)))", ["."])]]
```

The tree example was presented because it concisely demonstrates two features of the operators : (i) they allow lazy executable attribute grammars to be built, and (ii) they allow fully general attribute dependencies to be specified. We believe that the operators constitute the first environment to be built with these two properties.

5 AN EXAMPLE ILLUSTRATING HOW THE OPERATORS CAN BE USED TO CONSTRUCT A SIMPLE NATURAL LANGUAGE QUERY EVALUATOR THAT USES INTENSIONAL, EXTENSIONAL AND UNKNOWN VALUES.

This example illustrates how the operators can be used to construct a simple natural language query evaluator in which intensional and extensional attributes are used in the calculation of answers. The example also illustrates how the operators can be used to implement a simplistic approach to the accommodation of unknown values.

The interpreter given below takes as input a question such as "planets spin?" or "people think?" and returns a result that is obtained by first of all examining the intensional value INT w associated with the common noun and the intensional value INT x associated with the intransitive verb. If the pair (INT w, INT x) is a member of the second order sub_set_relation, then the answer to the question is "yes", otherwise the extensional values EXT y and EXT z, associated respectively with the common noun and the intransitive verb, are examined. If the extensional values are known, then, if the set z includes the set y, the answer is "yes", otherwise the answer is "no". If either of the extensional values are unknown, the answer is "unknown".

```
// PASSAGE#3
attribute      ::= INT_VAL [char] | EXT_VAL ext_val | ANSWER [char]
ext_val        ::= KNOWN intrans_or_cnoun_val | UNKNOWN
intrans_or_cnoun_val ::= INTRANS_VAL [entity_rep] | CNOUN_VAL [entity_rep]
entity_rep     ::= E num

common_noun    = mkint
  [ ("planets", [INT_VAL "planet_set", EXT_VAL (KNOWN (CNOUN_VAL planet_set))]),
    ("people", [INT_VAL "people_set", EXT_VAL (KNOWN (CNOUN_VAL people_set))]) ]

intrans_verb   = mkint
  [ ("spin", [INT_VAL "spin_set", EXT_VAL (KNOWN (INTRANS_VAL spin_set))]),
    ("think", [INT_VAL "think_set", EXT_VAL UNKNOWN]) ]

question       = ψ (s1 common_noun ... s2 intrans_verb)
  [rule 1.1 (ANSWER lhs) <- int_ext_calc[INT_VAL ↑ s1,
                                           INT_VAL ↑ s2,
                                           EXT_VAL ↑ s1,
                                           EXT_VAL ↑ s2]]

int_ext_calc [INT_VAL w, INT_VAL x, EXT_VAL y, EXT_VAL z]
  = ANSWER "true", member sub_set_relation (w,x)
  = check y z, otherwise
  where
    check (KNOWN c) (KNOWN i) = ANSWER "true", (includes i c)
    check (KNOWN c) (KNOWN i) = ANSWER "false", ~ (includes i c)
    check any any' = ANSWER "unknown"
    includes (INTRANS_VAL a) (CNOUN_VAL b) = (a -- b) = []
```



```

planet_set = [E 9, E 10, E 11, E 12, E 13, E 14, E 15, E 16, E 17]
people_set = [E 54, E 55, E 56]
spin_set   = [E 9, E 10, E 11, E 12, E 13, E 14, E 15, E 16, E 17]

sub_set_relation = [("people_set", "think_set")]

```

This executable specification can be explained informally as follows : The first three lines state : (i) that both intensional values and answers are 'made from' character strings, (ii) that extensional values can either be of type known or unknown, (iii) that known extensional values are made from values of intransitive verbs or common nouns, (iv) that values of intransitive verbs and common nouns are made from lists of entity representations, and (v) that entity values are made from numbers. The dictionary lists are such that the extensional value of the word "think" is unknown. The definition of the complex interpreter `question` states that the answer is computed by applying the attribute function `int_ext_calc` to the intensional and extensional attributes of the `common_noun` and `intrans_verb` components. The function `int_ext_calc` begins by looking at the intensional values and the second order relation `sub_set_relation` to see if it can return the answer "true" immediately. If not, it checks the extensional values to see if they are both known. If so, a set inclusion test is carried out and an appropriate answer returned. If either of the extensional values is unknown, then the answer returned is "unknown". The following examples illustrate the use of the interpreter :

```

Qu 1) question [] [([]), words "people think?"] => [(["ANSWER "true"], ["?"])]
Qu 2) question [] [([]), words "planets spin?"] => [(["ANSWER "true"], ["?"])]
Qu 3) question [] [([]), words "planets think?"] => [(["ANSWER "unknown"], ["?"])]

```

The answer to Qu 1 was obtained immediately from the intensional values , the answer to Qu 2 involved a set inclusion test, and the answer to Qu3 was returned because no intensional or extensional value was known.

6 INTENSIONAL TRANSFORMATIONS

We are currently examining ways in which to extend the calculus so that intensional transformation rules can be stated declaratively and then implemented automatically by the interpreters. At first, it would appear that transformation rules which are defined in terms of the concrete syntax of the input language can be incorporated quite easily. For example, consider the following rule which is relevant to example #1 :

$p \text{ implies } q \Rightarrow \neg p \text{ or } q$

This rule could be incorporated into the specification of the implication interpreter as follows :

```

implication = TRANSFORM (s1 expr ... s2 !"implies" ... s3 expr)
               TO expr (["-"] ++ s1 ++ ["or"] ++ s3)

```

The structure `TRANSFORM X TO I Y` would be such that whenever the interpreter `implication` were applied to an input structure matching `X`, the result(s) returned would be obtained by applying the interpreter `I` to the structure `Y`. Unfortunately, there are a number of problems with this simplistic approach :

1. If the grammar contained a number of 'related' transformation rules, there would be no guarantee that all possible transformations according to the rules would be carried out. For example, if we were to add a rule for cancelling double negations to the interpreter `expr`, in addition to the rule for implications given above, then all double negations in the input would be transformed. However, some of the double negations that arise through the transformation of implications, will be missed.
2. There are a number of reasons why transformation rules should be expressed in terms of the abstract syntax of the input language. To accommodate this, the method for implementing transformations would have to access attribute values.

The extension of the calculus to include a satisfactory approach to transformation is an area that we are actively investigating. If successful, the resulting automatic implementation of declaratively specified transformation rules could be used :

1. To implement efficiency improving transformations in, for example, database query processors.

2. To generate all possible transformations of input given some set of transformation rules. This could be achieved by having multiple productions for each syntactic construct : one that does not transform the construct but simply returns the construct as its result, and others that transform the construct in various ways according to the transformation rules. The generated transformations could then be tested until one is found that meets certain criteria. The production rules could be ordered to reflect relevant heuristics. The lazy evaluation strategy would make this a plausible approach to solving certain types of constraint satisfaction problems.
3. To implement Chomsky type transformations in natural language processors. The ability to perform intensional and extensional processing in one integrated environment may facilitate the investigation of syntactic and semantic theories of natural language processing.

7 CONCLUDING COMMENTS

The programming environment that we have built may be thought of as a step towards a realisation of a suggestion made by Knuth [5] in 1971 that executable attribute grammars might provide a viable declarative programming language. The notion that such a language could be realised as a calculus of interpreters was influenced by Bird's notion of program calculation [1]. The implementation of the operators of the calculus has been greatly facilitated by the availability of Turner's higher order, pure, lazy functional programming language Miranda[6].

Clearly, we need to do more work on formalising the calculus. In the examples given in this paper, we have not used the calculus to 'calculate' anything other than the output from interpreters. Ideally, one should be able to methodically transform definitions of interpreters to obtain other forms that may be more efficient with respect to given criteria. This is another area of research that we are currently pursuing. To date, we have informally transformed a number of interpreters to more efficient forms. For example, the theorem prover, given in section 3, was derived informally from a more inefficient form. The details of the derivation are given in [2]. However, we have not yet identified an appropriate method to formalise such transformations. It would appear that an appropriate method would have to integrate a number of algebras related to transformations at various levels in the syntactic/semantic spectrum.

Earlier versions of the operators described in this paper have been used to build a natural language interpreter which is based on Montague's approach to the interpretation of natural language [3 and 4]. The interpreter is capable of answering first order, non-modal, non-intensional questions about the planets, their moons, and the people who discovered them. (Note that the notion of intensionality in natural language is somewhat different from that assumed in section 6). The operators have also been used to build a translator from an executable hardware specification language to a netlist format suitable for input to a VLSI layout package. We intend to continue experimentation in these areas and in particular to study the use of the calculus in the investigation of intensionality in natural language processing.

The authors acknowledge the assistance of N.S.E.R.C. of Canada, and the support provided by The University of Windsor. Thanks also go to Stephen Karamatos of The University of Windsor, who contributed to the work described in this paper.

REFERENCES

1. R. S. Bird, Algebraic identities for program calculation. *The Computer Journal*, 32 (2), 123 - 126 (1989).
2. R. A. Frost, 'Use of Algebraic Identities in the Calculation of Programs Constructed as Executable Specifications of Attribute Grammars' ACM/SIGSOFT International Workshop on Formal Methods in Software Development. Napa, California, (May 1990).
3. R. A. Frost and J. Launchbury, Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32 (2), 108 - 122 (1989).
4. R. A. Frost, and W. Saba, (in print) 'A database interface based on Montague's approach to the interpretation of natural language'. *IJMMS - International Journal of Man-Machine Studies*.
5. D. E. Knuth, Semantics of context free languages. *Mathematical Systems Theory* 2 (2), 127 - 146 (1968).
6. D. Turner, A non-strict functional language with polymorphic types. *Proc. IFIP Int. Conf. on Functional Programming Languages and Computer Architecture*, Nancy, France. Springer Lecture Notes in Computer Science vol. 201. (1985).

Appendix A : Formal definitions of operators

```

fail x      = []

succeed x   = x

term pair input =
  concat (map (interp_term pair) input)
  where
    interp_term (terminal,atts) (inh,[]) = []
    interp_term (terminal,atts) (inh,(u:us)) = [(atts,us)], u = terminal
                                           = []           , otherwise
    concat = foldr (++) []

! x ((atts,(t:ts)):rest) = (atts,ts):!x rest, t = x
                        = !x rest           , otherwise
! x any                  = []

(p | q) input = p input ++ q input

(p excl_| q) input = q input, x = []
                  = x,otherwise
                  where x = p input

(p @ q) x = q (p x)

mkint [] = fail
mkint (p:ps) = term p | mkint ps

```

The formal definition of the ψ structure, when converted to executable Miranda, consists of about 75 lines of code. It is not presented here but can be obtained by writing to the authors at the address given on the first page of this paper.

Appendix B : Conversion to Miranda

The following minor textual replacements will convert the definitions given in the paper to Miranda code.

<i>Symbol in Paper</i>	<i>Suggested Replacement</i>
!	uterm
	\$orelse
@	\$o
↑	\$u
↓	\$d
ψ	rstc
...	++

In addition, since operators are less binding than function application, constructs such as `s1 !"&"` would be replaced by `s1 (uterm "&")`.