A FUNCTIONAL PROGRAMMING LANGUAGE WITH CONTEXT FREE GRAMMARS AS DATA TYPES

by

Violet R. Syrotiuk

Technical Report 85-12

August 1985

A Functional Programming Language with Context Free Grammars as Data Types

by

Violet Rose Syrotiuk

B.Sc., The University of Alberta, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming

to the required standard

.....

.....

THE UNIVERSITY OF BRITISH COLUMBIA

October 1984

Table of Contents

A User's Manual for the Functional Programming Language Kaviar

1. Introduction

Kaviar is a functional programming language which has context free grammars as data types. Applications most natural for such a language are those involving symbolic manipulation. Examples of these might include symbolic integration or differentiation, transformation of grammars and so on.

Backus-Naur Form, abbreviated as BNF, is a well know meta-language for specifying the concrete syntax of a programming language. The intent of this manual is to present the BNF of *Kaviar* and explain how its constructs are built and the operation of the interpreter.

For a discussion of the evolution of the idea of using context free grammars for the specification of data types consult [Syrotiuk,1984].

1.1. Lexical Conventions

There are five classes of tokens: identifiers, keywords, constants, strings, and other separators. Each class is discussed below.

1.1.1. Identifiers

An identifier is a sequence of lower case letters and digits; the first character must be an alphabetic letter. The dash – counts as a letter.

1.1.2. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

type
sub-type
is-
default

Other identifiers that are reserved, because they are the names of predefined functions which may not be redefined are:

read	le	add
eq	gt	subtract
ne	ge	multiply
lt		divide

1.1.3. Constants

There are only two kinds of constants available: the integer constants of the predefined type *Nat,* and the two constants of the predefined type *Bool,* T and F, for true and false.

1.1.4. Strings

A string is a sequence of characters surrounded by double quotes, as in "...".

1.1.5. Separators

Blanks, tabs, newlines and comments, collectively called *white space,* are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.

The characters /* introduce a comment, which terminates with the characters */. Comments may be nested and may appear anywhere in a *Kaviar* program.

1.2. The BNF of Kaviar

The following sections present the BNF of *Kaviar* in understandable chunks along with examples and an informal explanation.

1.2.1. A Kaviar Program

A *Kaviar* program is a sequence of type definitions, followed by a sequence of function definitions.

<Program> ::= <Definitions> { <Invocation_Exprs> }

<Definitions> ::= <Type_Defns> <Function_Defns>

The type definitions precede function definitions only to enhance the speed of the type checking capabilities of the interpreter.

Once all the definitions have been completed, the invocation expressions follow, enclosed within braces.

The next section describes the definition of types.

1.2.2. Type Definitions

Context free data types as well as sub-types, which may be non-context free, are accomodated by *Kaviar*.

<Type_Defns> ::= <Context_Free_Types> <Sub_Data_Types>

The context free types are defined by a context free grammar, while sub-types are defined by predicates.

Since sub-types are, essentially, restrictions on context free types, they follow the definition of these types to simplify ensuring the predicate operates over some previously defined context free type.

Considered first will be the definition of context free types.

1.2.2.1. Context Free Data Types

Each context free type definition can be viewed as defining only one of the nonterminals N_i, $1 \leq i \leq k$, $N = \{N_1, N_2, \ldots, N_k\}$ of a grammar, G. Recall that there is no restriction on the number of alternatives a single nonterminal may have. That is, a non-

terminal N_i may appear on the left hand side of arbitrarily many production rules.

In *Kaviar,* each alternative is further named by a *parser.* This facilitates, when given a terminal string ω, derived from a nonterminal $\{ \omega \mid N_i \rightarrow^* \omega \}$ finding which alternative of N_i ω was derived from. The parser names for each alternative of N_i must be distinct so that the alternative used in deriving ω can be uniquely determined.

As an example, consider the familiar LISP symbolic expression, or S-expression. Its recursive definition is:

> Basis: An atom is an S-expression.
> Recursive Construction Rule: If s_1 and s_2 are S-expressions, so is $(s_1.s_2)$.
> Closure: Only those values that result from finitely many applications
> of the recursive construction rule to the basis values are S-expressions.

An equivalent context free grammar over the atoms x and y is:

> s-expr → atom
> s-expr → (s-expr . s-expr)
> atom → x
> atom → y

As just described, each alternative of a given nonterminal, N_i, is further named by a parser. A nonterminal symbol will begin with <, followed by the nonterminal name, N_i, a colon, the parser name and then the closing >. Terminal symbols are strings and so will be enclosed within double quotes. Then, the grammar for S-expressions becomes:

> <s-expr:atom> → <atom:a>
> <s-expr:list> → "(" <s-expr:car> "." <s-expr:cdr> ")"
> <atom:x> → "x"
> <atom:y> → "y"

Since this form is lengthy, it is abbreviated by

```
type s-expr
{
     <atom> → <atom:a>;
     <list> → "(" <s-expr:car> "." <s-expr:cdr> ")"
}

type atom
{
     <x> → "x";
     <y> → "y"
}
```

and this is exactly the form the BNF for context free types in *Kaviar* generates. The interpreter converts this abbreviated form to the lengthier form.

The BNF for a context free type, then, is:

```
<Context_Free_Types> ::= <Context_Free_Type> |
                         <Context_Free_Type> <Context_Free_Types>

<Context_Free_Type> ::= type <Type_Name> { <Alternative> }

<Type_Name> ::= <Identifier>

<Alternative> ::= <Production> |
                  <Production> ; <Alternative>

<Production> ::= < <Parser> > → <Rewrite_Rule>

<Rewrite_Rule> ::= <Null_Rule> |
                   <Non_Null_Rule>

<Null_Rule> ::= λ

<Non_Null_Rule> ::= <Terminal> |
                    <Nonterminal> |
                    <Terminal> <Non_Null_Rule> |
                    <Nonterminal> <Non_Null_Rule>

<Terminal> ::= <String>

<Nonterminal> ::= < <Type_Name> : <Selector> >

<Selector> ::= <Identifier>
```

This allows for an arbitrary number of context free data types.

Notice that nonterminal symbols on the right hand side of the production rule are composed of two parts: a type name and a selector. The type name must be the name of a context free type, already defined or to be defined by <Context_Free_Type>, or

one of the two predefined types, *Nat* or *Bool*. The selector is an identifier. Its use will become clear when recursive functions are discussed. Selector names for all the nonterminals appearing on the right hand side of a given alternative must be distinct.

Sub-type definitions follow context free type definitions. The next section describes these types.

1.2.2.2. Sub-Data Types

Suppose N_i is a context free type and

$$P(x : N_i) : \text{Bool}$$

is a defined total predicate. By

$$R = P(N_i)$$

a decidable subset

$$\{ x \mid x \in L(N_i) \cap P(x) = T \}$$

of $L(N_i)$ can be specified. Then R is called a ***sub-type*** of N_i. That is, a sub-type restricts a context free type N_i to only those values which satisfy the predicate P. These sub-types can be non-context free.

In *Kaviar*, a sub-type is specified as:

```
<Sub_Data_Types> ::= <Sub_Data_Type> |
                     <Sub_Data_Type> <Sub_Data_Types>

<Sub_Data_Type> ::= sub-type <Type_Name>
                    ( <Context_Free_Type_Name> )
                    = <Predicate> ;

<Context_Free_Type_Name> ::= <Identifier>
```

This allows for any number of sub-types the user wishes to define.

As mentioned already, the sub-type is a decidable subset of a context free type hence <Context_Free_Type_Name> must be the name of one of context free types. The BNF of the predicate that must be satisfied is given by:

<Predicate> ::= (<Predicate>) |
¬ <Predicate> |
<Function_Name> |
(<Predicate>) ∩ <Predicate> |
(<Predicate>) ∪ <Predicate> |
¬ <Predicate> ∩ <Predicate> |
¬ <Predicate> ∪ <Predicate> |
<Function_Name> ∩ <Predicate> |
<Function_Name> ∪ <Predicate>

<Function_Name> ::= <Identifier>

That is, the predicate is an arbitrary logical formula. The functions referenced in the formula are restricted to be unary over a context free data type, being the same type the sub-type is defined over. In addition, these functions must have as their result type the predefined type *Bool*. This is so that the predicate itself can evaluate to true or false.

Consider the following context free type ***number*** defined as being a natural number:

type number
{
<n> → <Nat:n>
}

and suppose the subset to be specified is, say, a number corresponding to the months of the year.

$$\{ x \mid x \in L(\text{number}) \cap ((x \geq 5 \cap x \leq 12) = T) \}$$

that is, that is *x* is a ***number,*** but is also within the range one to twelve. This sub-type may be specified as:

sub-type month(number) = ge-one ∩ le-twelve;

So the functionality of ***ge-one*** and ***le-twelve*** must be:

ge-one(x : number) : Bool
le-twelve(x : number) : Bool

Now that the construction of types has been dealt with, a discussion of how to define functions that operate over these types appears in the next sections.

1.2.3. Function Definitions

Function definitions in *Kaviar* have the form:

```
<Function_Defns> ::= <Fcn_Defn> |
                     <Fcn_Defn> <Function_Defns>

<Fcn_Defn> ::= <By_Recursion> |
               <By_Unification>
```

Functions may be defined by one of two methods: by recursion, or by unification. Each method will be discussed in turn, accompanied by an example.

1.2.3.1. Recursive Functions

The BNF of recursive functions is:

```
<By_Recursion> ::= <Function_Name> ( <Typed_Variable_List> )
                   : <Result_Type> { <Case_Stmts> }

<Typed_Variable_List> ::= <Identifier> : <Type_Name> |
                          <Identifier> : <Type_Name> , <Typed_Variable_List>

<Result_Type> ::= <Identifier>

<Case_Stmts> ::= <Case_Stmt> |
                 <Case_Stmt> ; <Case_Stmts>

<Case_Stmt> ::= is- <Parser> ( <Identifier> )
                → <Recursive_Exprs>

<Parser> ::= <Identifier>

<Recursive_Exprs> ::= <Recursive_Expr> |
                      <Recursive_Expr> <Recursive_Exprs>

<Recursive_Expr> ::= <Number> |
                     <String> |
                     <Identifier> |
                     <Truth_Value> |
                     <Selector> < <Identifier> > |
                     <Function_Name> ( <Rec_Par_List> )

<Rec_Par_List> ::= <Recursive_Exprs> |
                   <Recursive_Exprs> , <Rec_Par_List>
```

Suppose a recursive function was to be defined which takes an arbitrary S-expression and reverses it. The functionality of the function would be:

reverse(s : s-expr) : s-expr

Ideally, a <Case_Stmt> should be given for each alternative of the type of the identifier in the parameter list to take care of any derivation of that type. The type *s-ezpr* was defined as:

```
type s-expr
{
      <atom> → <atom:a>;
      <list> → "(" <s-expr:car> "." <s-expr:cdr> ")"
}
```

which has two alternatives. If the value of *s* is an atom, reversing an atom yields itself so the <Case_Stmt> would be:

is-atom(s) → s

If the value of *s*, however, is a list, reversing a list consists of concatenating the reverse of the tail end of the list, given by the derivation of the nonterminal <s-expr:cdr>, with the reverse of the head of the list, given by the derivation of the nonterminal <s-expr:car>. Hence, the <Case_Stmt> would look like:

is-list(s) → "(" reverse(cdr<s>) "." reverse(car<s>) ")"

where the selectors *car* and *cdr* select the values corresponding to the derivations of their nonterminals. The operation of selectors will be discussed in more detail in section §1.2.3.1.2.

So the complete function is:

```
reverse( s : s-expr ) : s-expr
{
      is-atom(s) → s;
      is-list(s) → "(" reverse( cdr<s> ) "." reverse( car<s> ) ")"
}
```

If the function defined has more than one parameter a <Case_Stmt> which operates over any of the identifiers could be given, but only the <Recursive_Exprs> of the first applicable parser found will be evaluated. The notion of what an applicable parser is will be discussed in the next section, §1.2.3.1.1.

Of course, each function defined must be given a unique name. *Kaviar* permits no operator, or function, overloading. The identifiers present in the <Typed_Variable_List> are the only ones that may appear within the function. The type of a given identifier in this list may be a context free type name, a sub-type name or one of the predefined type names. Likewise for the result type of the function. Before entry to the function each parameter is checked to ensure it is of the correct type. If it is not an error will report the value at fault.

1.2.3.1.1. Interpreting a Recursive Function

If all the parameters of a function are of the correct type, the interpreter proceeds to determine the <Case_Stmt> to execute. Suppose the function *reverse* is invoked as:

reverse("(" "x" "." "y" ")")

The terminal string "(" "x" "." "y" ")" ∈ L(s-expr) so for the duration of the function the identifier *s* has this value. The parsers used in the body of a recursive function must be valid for the type of the identifier it operates over. For example, the type of *s* is *s-expr* so the only parsers that may operate on *s* are those that appeared in the definition in the type *s-expr*. Looking back at the definition of *s-expr* the available parsers are found to be *atom* and *list*.

If the type of the identifier is a sub-type, the applicable parsers are those of the context free type the sub-type is defined over. For example, if an identifier had type *month* which is defined over the context free type *number*, the only applicable parser would be *n*.

If the type of the identifier is the predefined type Bool, two parsers are available for use. They are *true* and *false* just as if the type Bool had been defined as:

```
type Bool
{
      <true> → T;
      <false> → F
}
```

No parsers are provided for type Nat because there are an infinite number of them.

The interpreter steps through the <Case_Stmts> of the function one by one until the applicable one is found. How is a <Case_Stmt> determined applicable?

First, the <Parser> of the <Case_Stmt> is extracted, call it *parser*. Then the alternative with this parser name is extracted from the type definition. This might be

<type:parser> → β

Then, the interpreter tries to determine whether the value of the identifier, the parser is operating over, can be derived starting from this rule. If so, the <Case_Stmt> is then termed applicable. The interpreter would then go on to execute the <Recursive_Exprs> of this <Case_Stmt>.

Otherwise, the interpreter would move on to the next <Case_Stmt> to determine if it is applicable. If the interpreter exhausts all of the <Case_Stmts> without finding one to be applicable an error will result.

Supposing an applicable parser is found, the next section describes how its <Recursive_Exprs> are evaluated by the interpreter.

1.2.3.1.2. Interpreting Recursive Expressions

There are six types of recursive expressions. The expressions occurring are evaluated in sequence from left to right and the results concatenated together. The final result must be of type <Result_Type>. Numbers of type *Nat,* truth values of type *Bool* and strings require no interpretation. The interpretation of an identifier is the value assigned to it on entry to the function.

Selectors must operate on the same identifier as the parser, and can only be the

ones present in the alternative of the parser. For example, the alternative *list* of type *s-expr* is:

<list> → "(" <s-expr:car> "." <s-expr:cdr> ")"

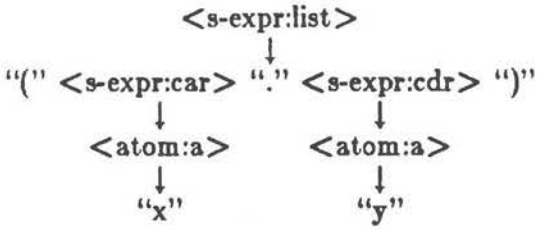
On entry to *reverse,* the value of *s* was set to "(" "x" "." "y" ")". The interpreter can determine that this value was derived from the alternative <list>, so the only selectors that may be active in the <Case_Stmt> with parser *list* are *car* and *cdr* since they are the ones that appear in the definition of the alternative. Recall, each nonterminal on the right hand side of the production rule has the form:

< <Type_Name> : <Selector> >

The <Selector> may be used as a function that operates over the value of the identifier the parser was applicable to. The effect of a selector, as used in the following extract from the function *reverse*

is-list(s) → "(" reverse(cdr<s>) "." reverse(car<s>) ")"

is to select, from the value of the identifier, the subtree corresponding to the derivation of the nonterminal. The type of the subtree will, of course, be <Type_Name>. For example, the derivation tree of the value of *s* is:



The selector returns the value of the subtree. So *car*<*s*> = "*x*" and *cdr*<*s*> = "*y*".

If the expression is a function invocation, its parameters are first evaluated and then the function is called. The result of the function is then concatenated to the partial result.

When all of the expressions on the right hand side of the arrow have been evaluated the resulting terminal string is checked to see if it is of type <Result_Type> and if so, it is returned, otherwise an error is reported.

The evaluation of

reverse("(" "x" "." "y" ")")

would then proceed as follows. The parser *is-list(s)* would be found applicable. Its expressions are:

"(" reverse(cdr<s>) "." reverse(car<s>) ")"

The string "(" becomes the partial result. The parameter to *reverse, cdr<s>*, is evaluated, yielding "y", so reverse("y") is invoked. It returns "y" as a result. This result is concatenated onto the partial result, the partial result now becoming "(" "y". Continuing in this fashion the final result is the terminal string "(" "y" "." "x" ")", which is of type *s-expr*.

Now that the operation of recursive functions is understood, the next sections describe the second type of function definition – by unification.

1.2.3.2. Unification Functions

Functions may, alternatively, be defined by unification. The BNF for these types of functions is:

```
<By_Unification> ::= <Function_Name> ( <Type_Name> )
                    : <Result_Type> { <Unif_Clauses> }

<Unif_Clauses> ::= <Clause> |
                   default → <Unif_Exprs>

<Clause> ::= <Schema> → <Unif_Exprs> |
             <Schema> → <Unif_Exprs> ; <Clause> |
             <Schema> → <Unif_Exprs> ; default → <Unif_Exprs>

<Unif_Exprs> ::= <Unif_Expr> |
                 <Unif_Expr> <Unif_Exprs>

<Unif_Expr> ::= <Number> |
                <String> |
                <Identifier> |
                <Truth_Value> |
                <Function_Name> ( <Unif_Par_List> )

<Unif_Par_List> ::= <Unif_Exprs> |
                    <Unif_Exprs> , <Unif_Par_List>

<Schema> ::= <String> |
             <Identifier> |
             <String> <Schema> |
             <Identifier> <Schema>
```

Unification can be intuitively looked at as being pattern matching. In *Kaviar,* unification is restricted since matching is done at only the top-most level. In other words, the pattern must be at the <Rewrite_Rule> level. None of the nonterminals in the <Rewrite_Rule> are expanding when trying for a match.

All unification functions are unary since otherwise it would be impossible to tell what the <Schema> of a given <Clause> should be unified against.

The next section describes how to build a <Schema> so that the interpreter will be able to successfully unify expressions.

1.2.3.2.1. Finding a Unifier

Finding a unifier consists of finding an expression for each identifier symbol occurring in the <Schema> so that when these expressions are substituted for their respective identifiers the value of the original parameter results. To do this successfully, the

<Schema> must mimic the form of the alternatives of the type.

Ideally, there should be a <Clause> for each alternative of the type so that all possible derivations of the type may be handled. The <Schema> is built by specifying a string for every terminal symbol in the <Rewrite_Rule> and an identifier for every nonterminal symbol.

The definition of S-expressions was given as:

```
type s-expr
{
      <atom> → <atom:a>;
      <list> → "(" <s-expr:car> "." <s-expr:cdr> ")"
}
```

A unification function which reverses arbitrary S-expressions would have the functionality:

reverse(s-expr) : s-expr

so the schemas of the clauses should mimic the alternatives of *s-expr* since this is the type of the parameter.

The right hand side of the alternative <atom> has only one nonterminal, so the schema would consist of only one identifier. Reversing an atom yields itself so the <Clause> would be:

atom → atom

The right hand side of the second alternative, <list>, contains terminal and nonterminal symbols. Working from left to right, simply copy terminal symbols and assign nonterminal symbols distinct identifiers. So for <list> the <Schema> would be:

"(" s_1 "." s_2 ")"

Reversing a list requires concatenating the reverse of the tail end of the list with the head of the list. Thus the complete function would be:

```
reverse( s-expr ) : s-expr
{
        atom → atom;
        "(" s1 "." s2 ")" → "(" reverse( s2 ) "." reverse( s1 ) ")"
}
```

The interpreter finds a unifier by going through the <Unif_Clauses> of the function one by one trying to find which <Schema> the value of the parameter can be unified against.

The alternative the parameter value was derived from is what the <Schema> is compared to. For example, the alternative the terminal string "(" "x" "." "y" ")" was derived from is:

<list> → "(" <s-expr:car> "." <s-expr:cdr> ")"

Identifiers can be unified against nonterminal symbols and terminal strings must match exactly.

If unification can take place, the interpreter proceeds to evaluate the <Unif_Exprs> to the right of the arrow, binding any identifiers in the <Schema> to the value of the subtree the nonterminal they stand for derives. It makes sense that the only identifiers that can appear in the <Unif_Exprs> are those that appeared in the <Schema>.

If the reserved word *default* is found instead of a <Schema> the <Unif_Exprs> to the right of the arrow will be evaluated immediately.

If no <Schema> can be unified against the interpreter will indicate failure.

Assuming a unifier is found, the interpreter goes on to evaluate the unification expressions. The next section describes this.

1.2.3.2.2. Evaluating Unification Expressions

The unification expressions are evaluated in the exact same manner as recursive expressions, except for the fact that the selector functions cannot be used.

Once all the types and functions are defined there must be some way of using all these definitions. The next section describes the invocation expressions.

1.2.4. Invocation Expressions

The BNF of the invocation expressions is:

```
<Invocation_Exprs> ::= <Invoc_Expr> |
                       <Invoc_Expr> ; <Invocation_Exprs>

<Invoc_Expr> ::= Nl |
                 <String> |
                 <Identifier> |
                 read ( <Typed_Variable_List> ) |
                 <Function_Name> ( <Parameter_List> )

<Parameter_List> ::= <Parameters> |
                     <Parameters> , <Parameter_List>

<Parameters> ::= <Parameter> |
                 <Parameter> <Parameters>

<Parameter> ::= <String> |
                <Number> |
                <Identifier> |
                <Truth_Value> |
                <Function_Name> ( <Parameter_List> ) |
```

Input may be prepared for a *Kaviar* program by placing sentential forms in a data file, each one terminated by the character ".". The values may be read from the data file by issuing a *read(* $x_1 : t_1$,..., $x_n : t_n$ *)* invocation expression. The value read will be associated with the identifier x_i if its type is indeed t_i.

For example, the data file for the read statement:

read(s_1 : s-expr, s_2 : s-expr)

might look like:

"(" "x" "." "y" ")" .
"y" .

When a string is given as an invocation expression it will simply be written to standard output. Similarly, when an identifier is given the value associated with it by some read statement will be written to standard output.

When *Nl* is encountered a newline will be written to standard output.

As in interpreting recursive and unification expressions, parameters to a function are first evaluated, if necessary, and then the function is invoked. Upon return, the result of the function will be written to standard output.

1.2.4.1. Preparing Input for a Kaviar Program

Since all terminal symbols, except the constants of type *Nat* and *Bool,* must be quoted in a data file, typing all the quotes can be quite tedious. A program named *prepare* is provided to rid this annoyance.

Prepare reads from standard input and writes to standard output. It quotes all terminal symbols except constants of the predefined types and the character, ".", which separates terminal strings from one another. It leaves quoted symbols untouched.

Suppose a data file containing terminal strings of type *s-expr* is required. Then input to *prepare* might look like:

```
( x "." ( y "." x ) ) .
( y "." x ) .
```

Note that since the "." is used to separate symbolic expressions from one another it must be quoted to that it is not mistaken for the character that separates terminal strings from one another. The output of *prepare* would be:

```
"(" "x" "." "(" "y" "." "x" ")" ")" .
"(" "y" "." "x" ")" .
```

1.3. Built in Functions

There are several built in functions available. The basic arithmetic functions all have two parameters of type *Nat* with result type *Nat.* They are:

```
add( x, y )       /* x + y   */
subtract( x, y)   /* x - y   */
mutiply( x, y )   /* x * y   */
divide( x, y )
```

Divison is integer division in which the fractional part is truncated. The expression ***divide(x,y)*** produces the remainder when x is divided by y, and thus is zero when y divides x exactly.

The other built in functions take two arguments of type *Nat* and produce a *Bool* result. They are:

```
eq(x,y)     /* x = y */
ne(x,y)     /* x ≠ y */
le(x,y)     /* x ≤ y */
lt(x,y)     /* x < y */
ge(x,y)     /* x ≥ y */
gt(x,y)     /* x > y */
```

1.4. How to Run a Kaviar Program

A *Kaviar* source program is expected to come from standard input. All output is directed to standard output. There are three options available.

The *-q* option removes double quotes from values the invocation expressions write to standard output. This makes the output look tidier.

The *-s* option stops sub-type checking. In normal circumstances the interpreter verifies that something is a sub-type by evaluating a predicate. When this option is given it will bypass evaluating the predicate. Once a program is fully debugged this option might be used to enhance the speed of the interpreter.

And finally there is the *-d* option. If the *Kaviar* program has any read statements in it the interpreter expects that a data file has been prepared. The name of this file is to be given immediately after the *d* as in:

```
-dname
```

1.5. Implementation Restrictions and Details

The following are restrictions which have been arbitrarily set. To change these limits, the constants need only be changed and the interpreter recompiled.

No more than the first SIG_CHARS (12) characters of an identifier is significant, although more may be used.

At most MAX_CF_TYPES (50) and MAX_SUB_TYPES (50), context free and sub-types, respectively, may be defined in any one source program. Also, at most MAX_FCN (50) functions by either method may be defined.

The maximum number of parameters that a recursive function may be defined with is MAX_PARS (10).

The maximum length of a string is MAX_STR_LEN (256) counting the opening and closing quotes. There is currently no way of including double quotes within a string. The character % must not be used in strings since the Unix routine *printf* is used for output and the % introduces a conversion specification. Also, the character \ must not be used in strings since it introduces an escape sequence which could cause the interpreter to malfunction.

The interpreter does no checking for overflow in evaluating the built in functions. Behaviour of the interpreter in these cases is system dependent.

Although the symbol $\rightarrow$ has been used throughout the BNF for an arrow, the implementation recognizes an arrow as the concatenation of the characters = and >. Similarly, the symbol $\cup$ and $\cap$ have been used for logical *or* and *and,* respectively, but the implementation recognizes the characters | and & for these connectives.

1.6. Location of the Interpreter

The interpreter lives in the directory /ubc/guest/kanda/kaviar.

1.7. Sample Programs

Some sample programs are located in the directory /ubc/guest/kanda/tests.

Bibliography

[Syrotiuk,1984].

Syrotiuk, V.R., "A Functional Programming Language with Context Free Grammars as Data Types," *M.Sc. Thesis, Department of Computer Science, University of British Columbia*, October, 1984.

Abstract

Specifying data structures is important in programming languages. One approach uses initial algebras, but constructing specifications using them is difficult and hampered by restrictions. The recursive term algebra approach, uses recursive definition along with basic operations implied by the definition itself to induce an algebra of terms. Such a recursive definition may be transformed into an unambiguous context free grammar. These grammars are used to define data structures in a functional programming language named Kaviar. Provisions are made for non-context free data structures as well. This approach avoids the difficulties encountered in using the initial algebra method.

Table of Contents

List of Tables

List of Figures

Acknowledgements

My thanks to Drs. Akira Kanda and Harvey Abramson for their guidance, patience and cooperation, not to mention financial support. The enthusiasm and encouragement of my parents, Vera, Alan and Myron kept my motivation high. I could not ask for better friends than I have found in Steve, Denise, Brent and Bob. Gracias Eric, por lo tiempo pasado, y a lo tiempo que ha de venir.

Introduction

1. Framework

Specifications provide a means of abstracting ideas to reduce a problem to a more manageable and comprehensible size, making more apparent solutions to problems.

In programming languages, the function and procedure are commonly seen abstractions. Recently, an additional, powerful aid to abstraction has been identified – the data abstraction. A specification of a data structure should supply a representation independent characterization of the structure so the user need not be concerned with implementation details.

Six criteria which a technique for specifying such a characterization should satisfy are outlined.

The development of the initial algebra technique is described along with a programming language, OBJ, which uses initial algebras for specifying data types.

1.1. An Alternative Approach

In evaluating the initial algebra approach for the specification of data types it is found that such specifications are often very difficult to construct. Indeed, there are several restrictions imposed on the user.

These problems stimulated the search for an alternative method of specification. The recursive term algebra approach recursively defines not only a set of terms, but also basic operations over them. In other words, recursive definition induces an algebra of terms. The same algebra may be generated by an unambiguous context free grammar. Hence, context free grammars are used to specify data types.

The treatment of non-context free data structures is also discussed.

1.2. Implementation

A functional programming language, named Kaviar, has been designed and implemented. It uses context free grammars for the specification of data types. Sub-types, which may be non-context free, are defined by means of predicates which restrict some context free type.

Functions may be defined by one of two methods: by recursion, or by unification.

A full description of the language, some examples, and the operation of the interpreter is given.

1.3. Evaluation of the New Approach

The recursive term algebra approach is evaluated with respect to the criteria. Specifying data types using this approach overcomes the difficulties of the initial algebra method.

1.4. Overview

Chapter 2 provides the framework. Chapter 3 first evaluates the initial algebra approach. Then, the recursive term algebra approach is formally presented. The programming language Kaviar is described in Chapter 4. Chapter 5 evaluates this new approach. Finally, Chapter 6 proposes possible extensions to the method, an application of the method and some conclusions.

2. Formal Specifications of Data Abstractions

2. Introduction

This chapter introduces the notion of specifications in connection with abstract data types. Some criteria are established for evaluating the practical potential of specification techniques. One particular technique, the initial algebra approach is discussed along with a programming language based on it, OBJ.

2.1. The Role of Specifications

Programming is a problem solving activity. Consequently, a correct program implements a concept that exists in someone's mind. The concept can usually be implemented by many programs a situation depicted in Figure 1, but, realistically, only a small, finite number are of interest.

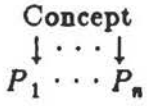


Figure 1: A concept and all programs
which implement the concept correctly.

Since programs can become quite complex, a *specification* of what a program is intended to do should be given before the program is actually coded. This philosophy is described by Figure 2.

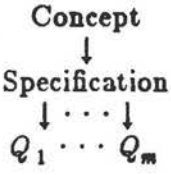


Figure 2: A concept, its specification and
all programs derived from the specification.

The purpose of the specification is to aid the understanding of the concept involved, and to increase the likelihood that the program, when implemented, will perform the intended function. So far, no mention of what exactly a specification is and how it can

satisfy these goals has been given. The next sections serve to answer these questions.

2.2. The Specification Unit

The nature of a specification is largely dependent on the program unit being specified. A specification of too small a unit does not correspond to any useful concept, just as the program comment

```
x := x + 1;     /* Increase x by 1 */
```

does not convey much information. The specification unit should correspond naturally to a concept, or *abstraction,* found useful in thinking about the problem to be solved.

In many of the existing programming languages there appear several implicit abstractions which are built into the language. The notions of arrays and records are examples of these, but perhaps the most commonly used kind of abstraction is the functional or procedural abstraction in which a parameterized expression or group of statements is treated as a single operation. When a function or procedure is made use of, the programmer is concerned only with what it does, not with the algorithm that is executed.

Multiprocedure modules are also important in system design. Here, procedures are grouped together because they interact in some way. They may share certain resources as well as information. Considering the entire group of procedures as a module permits all information about the interactions to be hidden from other modules. Other modules obtain information about the interactions only by invoking the procedures in the group. The purpose of hiding not only requires making visible certain essential properties of a module but making inaccessible certain irrelevant details that should not affect other parts of a system.

Although functions, procedures and modules offer a powerful aid to abstraction they alone are not enough. An additional kind of abstraction has been identified – the *data abstraction.* It is comprised of a group of related functions or operations that act

upon a particular class of objects. There is one constraint - the behaviour of the objects can be observed only by applications of the operations. As with other methods of abstraction, the programmer is not concerned with how the data objects are represented. Instead, the operations on the objects are regarded as atomic even though several machine instructions may be necessary to perform them.

Some examples of data abstractions are segments, processes, files, and abstract devices of various sorts, in addition to the more ordinary stacks, queues, and symbol tables.

Multiprocedure modules are also used in the implementation of data abstractions. Each procedure in the module implements one of the operations. The module as a whole implements the specified properties of the *abstract type*. The word abstract, as used here refers to a data type that is supposed to be independent of its representation, in the sense that details of how it is implemented and the representation of the type itself are to be actually hidden or shielded from the user. The user is provided with certain operations and so only needs to know what they are supposed to do, not how they are done.

Thus, a specification provides a means of describing the abstractions a programmer can make use of to solve the problem at hand. Since the need and importance of functional and procedural abstractions are well known while the need for data abstraction is, perhaps, less understood, the next section attempts to explain their necessity.

2.3. The Need for Data Abstraction

As an example of the problems which crop up when the data abstraction is ignored and the operations in the group are given input/output specifications independently of one another, consider the following specification for the operation PUSH for stacks of integers.

$$\text{PUSH} : \text{STACK} \times \text{INTEGER} \rightarrow \text{STACK}$$

The input/output specification defines the output value of PUSH (the stack object returned by PUSH) in terms of the input values of PUSH (a stack object and an

integer). This can be done by defining a structure for stack objects and then describing the affect of PUSH in terms of this structure. In Pascal, a stack of integers might be defined as:

```
type STACK = record
       TOP  : integer;
       DATA : array [1..100] of integer
end;
```

where the record selector TOP points to the topmost integer in the stack. Then, assuming PUSH is a function, the meaning of,

NEW_STACK := PUSH(OLD_STACK, i)

that is, a description of the result of its execution could be stated as [1]

for all j { 1 ≤ j ≤ OLD_STACK.TOP
→ NEW_STACK.DATA[j] = OLD_STACK.DATA[j]
& NEW_STACK.TOP = OLD_STACK.TOP + 1
& NEW_STACK.DATA[NEW_STACK.TOP] = i }

In words, and by example, if OLD_STACK is as shown in Figure 3, then executing, say

NEW_STACK = PUSH(OLD_STACK, 4)

results in Figure 4.

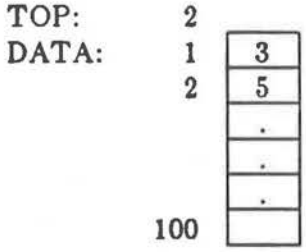


Figure 3: OLD_STACK before PUSH

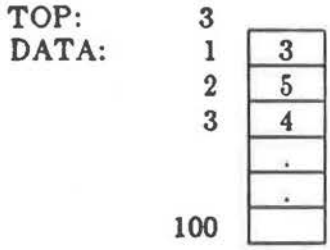


Figure 4: NEW_STACK after PUSH

That is, all of the entries in OLD_STACK are copied to corresponding index positions in NEW_STACK. The top of the stack in NEW_STACK becomes one greater than that of OLD_STACK and then the new top of stack entry is set to the integer input parameter, 4.

[1] Ignoring for the moment the behaviour of PUSH when the stack is full, that is, when OLD_STACK.TOP=100.

If the specification for POP is

POP: STACK → STACK × INTEGER

and, again, assuming POP is a function, the meaning of

i = POP(STACK)

could be stated as:

i = STACK.DATA[STACK.TOP] & STACK.TOP = STACK.TOP – 1

There are several things wrong with these specifications. The major flaw is that they do not describe the concept of stacklike behaviour. For example, a theorem stating that POP returns the value most recently PUSHed on the stack can only be inferred from the details of the specifications. This method of deriving theorems is undesirable because the programmer must get involved with the detail (which is really implementation information), rather than stating the concept directly.

Another problem is that the independence of the specifications of PUSH and POP is deceptive. A change in the specification of one is almost certain to lead to a change in the specification of the other. For example, in addition to being related through the structure chosen for the stack, the specifications of PUSH and POP are also related in their interpretation of this structure – the decision to have the selector TOP point to the topmost integer rather than, say, to the first available slot.

If a data abstraction such as STACK is specified as a single entity, much of the extraneous detail concerning the interactions between the operations can be eliminated, and the effects of the operations can be described at a higher level. Some specification techniques for data abstractions, such as the initial algebra and recursive term specification techniques to be presented in later sections, use input/output specifications to describe the effects of the operations. The difference is these specifications are expressed in terms of abstract objects with abstract properties instead of the very specific properties used in the example above.

In some instances it may not even be necessary to describe the individual operations separately. Instead, the effects of the operations can be described in terms of one another. For example, the effect of POP might be defined in terms of PUSH by

POP(PUSH(stack, i)) = i

which states that POP returns the value most recently PUSHed.

Now that the need for data abstractions has been established there are some properties of them that should be pointed out. The next section presents some of these.

2.4. Properties of Specifications of Data Abstractions

Specification techniques for data abstractions can be viewed as defining something very like a mathematical discipline. Just the way in which number theory arises from specifications, like Peano's axioms for the natural numbers, the discipline arises from the specification of the data abstraction. The *domain* of the discipline, the set on which it is based, is the class of objects belonging to the data abstraction, and the operations of the data abstraction are defined as mappings on this domain. The theory of the discipline consists of the theorems and lemmas derivable from the specifications.

The information contained in a specification of a data abstraction can be divided into two parts. Information about the actual meaning or behavior of the data abstraction is described in the *semantic* part. This description is expressed using a vocabulary of terms or symbols defined by the *syntactic* part.

The first symbols defined by the syntactic part of a specification identify the abstraction being defined and its domain. Usually, an abstraction has a single class of defined objects, and, typically the same symbol denotes both the abstraction and its class of objects. Thus the objects belonging to the data abstraction STACK are referred to as STACKs.

The remaining symbols introduced by the syntactic part name the operations of the abstraction and define their input/output specification. For example operations of the

stack abstraction might include:

CREATE : → STACK
PUSH : STACK × INTEGER → STACK
POP : STACK → STACK
TOP : STACK → INTEGER
EMPTY : STACK → BOOL

Some observations should be noted about this example. First, more than one domain appears in the specification. In practice, the specifications for many interesting data abstractions include more than one domain. Normally, only one of these, here STACK, is being defined with the remaining domains, here INTEGER and BOOL, and their properties are assumed to be known.

The second observation is that the group of operations can usually be partitioned into three categories. The first is those consisting of operations that have no operands belonging to the class being defined, but which yield results in the defined class. This includes the constants, represented as argumentless operations, such as CREATE. The second category consists of those operations, like PUSH and POP, which have some of their operands in, and yield their results in, the defined class. Those operations, TOP and EMPTY for stacks, whose results are not in the defined class make up the third category.

A third observation is that the input/output specification of an operation does not necessarily have to correspond to the way the operation would be programmed. For example, POP might be programmed in a way which removes a value from a stack, and returns both the new stack and the value.

The semantic part of the specification uses the symbols defined in the syntactic part to express the meaning of the data abstraction. Two different approaches are used in capturing this meaning: either an abstract model is provided for the class of objects and the operations are defined in terms of the model, or the class of objects is defined implicitly through descriptions of the operations.

Equipped with the idea of what a specification is, there remains the problem of devising a method of using them. The next section establishes some criteria for evaluating such methods.

2.5. Criteria for Evaluating Specification Methods

Liskov and Zilles [Liskov&Zilles,1978] have outlined a number of requirements an approach to specification should satisfy to be useful. The criteria, described below, include practical as well as theoretical considerations.

This first criterion should be satisfied by any specification technique.

1) *Formality.* A specification method should be formal. Specifications themselves should be written in a notation which is mathematically sound. This allows the formal specification techniques to be studied mathematically, so that other interesting questions such as the equivalence of two specifications may be set and answered. For those interested in program verification this is especially important.

The next three criteria address the construction of specifications.

2) *Constructibility.* It should be possible to construct specifications without much difficulty. Two aspects of the construction process are of interest here: the difficulty of constructing a specification in the first place, and the difficulty in knowing that the specification captures the concept.

3) *Comprehensibility.* Properties of specifications which determine comprehensibility are size and clarity. Clear, small specifications are desirable since they are usually easier to understand than larger ones.

4) *Minimality.* It should be possible, using the specification method, to construct specifications which describe the properties of the concept precisely and unambiguously in a way which adds as little extraneous information as possible. In

particular, a specification should say what functions a program should perform, but little, if anything, about how the functions are performed.

Flexibility and generality are issues addressed by the last two criteria.

5) *Range of Applicability.* There is a class of concepts associated with each specification technique which can be described in a natural and straightforward way, thus satisfying the criteria dealing with construction. Concepts outside of this class can only be defined with difficulty, if they can be defined at all. Clearly, the larger the class of concepts which may be easily described by a technique, the more useful the technique.

6) *Extensibility.* It is desirable that a minimal change in a concept result in a similar small change in its specification. This criterion especially impacts the constructibility of specifications.

A position has been reached where a specification technique may be examined. The technique of interest is an algebraic one. It is the precursor to the recursive term algebra technique introduced in Chapter 3.

2.6. Algebraic Preliminaries

Algebraic ideas have shown up in connection with abstract data types. Before introducing such a specification technique it is perhaps worthwhile reviewing some algebraic fundamentals. Those already familiar with these fundamentals can move on to §2.7 with no loss of continuity.

2.6.1. Signatures

In programming terms, a *signature* corresponds to a collection of declarations, declaring types, constants and procedures:

INTEGER, BOOL, STACK : type

CREATE : function() result STACK
PUSH : function(STACK,INTEGER) result STACK
TOP : function(STACK) result INTEGER
POP : function(STACK) result STACK
EMPTY : function(STACK) result BOOL

Notice that the types are just named, nothing is said of what they consist of. Bodies of procedures are not given. They can be viewed as being like the *forward references* of Pascal. Constants are regarded as functions with no arguments.

Henceforth, following usual mathematical practice, the word *sort* will be used for type, and *operator* for function names.

More formally, a signature Σ, is a pair $<S,\Omega>$ consisting of a set of sorts $S=\{s_1 \cdots s_n\}$, and a set of operators $\Omega=\{\omega_1 \cdots \omega_n\}$ each with given input and output sorts. That is, the operators have the form $S' \times S$. For example, the operations of the stack abstraction may include:

CREATE : $\rightarrow$ STACK
PUSH : STACK $\times$ INTEGER $\rightarrow$ STACK
TOP : STACK $\rightarrow$ INTEGER
POP : STACK $\rightarrow$ STACK
EMPTY : STACK $\rightarrow$ BOOL

Notice this data type, a stack of integers, involves several different sorts, including truth values of sort BOOL, integers of sort INTEGER, and stack states of sort STACK. All things of sort s_i are lumped together into a set called the *carrier* of sort s_i.

Unfortunately, signatures alone are not enough to specify a data type since they do not fully describe what is going on. For this, an algebra is needed.

2.6.2. Algebras

When an association between some particular set with the sort and some particular function with each operator is made an *algebra* is the result. That is, an algebra is a signature together with a function taking sorts to sets and another function taking operators to functions.

An *I-indexed family of sets* is a function from I to sets. If NATURAL=$\{0,1,2,...\}$ then a function mapping 0 to {a,b}, 1 to {a,c,d}, 2 to $\emptyset$ and so on is a NATURAL-indexed family of the set of letters.

Then, more formally, if Σ is a signature, a Σ-*algebra* A is an S-indexed family of sets, $|A|$, denoting the carrier of A, together with an $S^* \times S$-indexed family of functions $\omega_{u,s} : \Omega_{u,s} \rightarrow (|A|_u \rightarrow |A|_s)$ where $u \in S^*$, and $s \in S$.

An s-sorted Σ-algebra A is, essentially, a collection A_S of carriers, one for each s in a set S of sorts together with a collection of operations, that is functions, among them.

The index set S for the stack example above is {INTEGER,BOOL,STACK}. An S-sorted algebra A for this sort would have three carriers $A_{INTEGER}$, A_{BOOL}, and, A_{STACK}, and some of its operators, more precisely stated, are

$$\begin{aligned} &\text{PUSH} : A_{STACK} \times A_{INTEGER} \rightarrow A_{STACK} \\ &\text{TOP} : A_{STACK} \rightarrow A_{INTEGER} \\ &\text{CREATE} : \rightarrow A_{STACK} \end{aligned}$$

Each $\omega \in \Omega_{u,s}$ is called an operation of A and is named by ω. $\Omega_{\lambda,s}$ is the set of names of constants of A of sort s. For example, $\Omega_{\lambda,STACK} = \{CREATE\}$, $\Omega_{STACK,INTEGER,STACK} = \{PUSH\}$, $\Omega_{STACK,INTEGER} = \{TOP\}$ and $\Omega_{u,s} = \emptyset$ otherwise.

For the characterization of the data type to be independent of representation the algebra must be initial. Before discussing what this means the notions of homomorphism and isomorphism should be understood.

2.6.3. Homomorphisms and Isomorphisms

Suppose there are two algebras which are rather similar in that evaluating in one gives analogous results to evaluating in the other. If in the first algebra evaluating an expression like *plus(x,times(x,y))* with variable x bound to a and y bound to b gives result c, then evaluating the same expression in the second algebra with the correspond-

ing binding, x to $f(a)$, y to $f(b)$, should give the corresponding result $f(c)$. It is then said that f is a *homomorphism* from the first algebra to the second one.

The integers modulo n consists of a set of n elements $\{0,1,...,n-1\}$ and an operation called addition modulo n which may be described as follows. Imagine the numbers 0 through $n-1$ as being evenly distributed on the circumference of a circle. To add two numbers h and k, start at h and move clockwise k additional units around the circle. $h+k$ will be the end point.

For example, the integers modulo 6 can be transformed into the integers modulo 3 by the function f,

$$f = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 0 & 1 & 2 \end{pmatrix}$$

as may be verified by comparing their tables below.

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

Replace x by f(x) →

+	0	1	2	0	1	2
0	0	1	2	0	1	2
1	1	2	0	1	2	0
2	2	0	1	2	0	1
0	0	1	2	0	1	2
1	1	2	0	1	2	0
2	2	0	1	2	0	1

Eliminating duplicate information, for example 2+2=1 appears four separate times in the table, gives:

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

The dictionary says that two things are *isomorphic* if they have the *same structure*. Consider the two groups G_1 and G_2 with their operations denoted by + and •, respectively.

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

•	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

G_1 and G_2 are different but isomorphic. Indeed, if in G_1 0 is replaced by e, 1 by a and

2 by b, then G_1 coincides with G_2. In other words there is a one-to-one correspondence

$$\begin{pmatrix} 0 & 1 & 2 \\ \downarrow & \downarrow & \downarrow \\ e & a & b \end{pmatrix}$$

transforming G_1 to G_2. It is called an isomorphism from G_1 to G_2.

So, if two groups are isomorphic, this means there is a one-to-one correspondence between them which transforms one of the groups into the other. Now if G and H are any groups, it may happen that there is a function which transforms G into H although this function is *not* a one-to-one correspondence. In this case, the function is a homomorphism.

More exactly, if G and H are groups, a homomorphism from G to H is a function $f : G \rightarrow H$ such that for any two elements, a and b in G:

$$f(ab) = f(a)f(b)$$

If G and H are groups, a bijective[2] function $f : G \rightarrow H$ with the property that for any two elements a and b in G

$$f(ab) = f(a)f(b)$$

is called an isomorphism from G to H.

Now with some idea of what homomorphisms and isomorphisms are, it is possible to return to the problem of finding a unique characterization of a data type using algebras.

2.6.4. Word Algebras and Initial Algebras

For any algebra there is a *word algebra* with its elements for a given sort consisting of all the expressions in the signature denoting elements of that sort. For example, for the number signature:

$$S = \{NUMBER\}$$

[2] A function f : A→B is called bijective if it is both injective and surjective. That is, if each element of A has exactly one partner in B and each element in B has exactly one partner in A.

$\Omega_{\lambda,NUMBER} = \{ZERO,ONE\}$
$\Omega_{NUMBER,NUMBER,NUMBER} = \{PLUS,TIMES\}$
$\Omega_{u,s} = \emptyset$, otherwise

the set of numbers of the word algebra would be "ZERO", "ONE", "PLUS(ZERO,ZERO)" and so on. The operations build larger expressions from smaller ones. The expressions could be reverse Polish, represented by trees or whatever. The representation is not the issue because all these expression algebras will be isomorphic. The key idea is that for any given interpretation of the operators there is a unique value for each expression. The interpretation is just another Σ-algebra, *A*, and the evaluation of the expressions is just a unique homomorphism from the word algebra to *A*. An algebra having this unique homomorphism property is called the *initial* *S*-algebra.

In other words, *I* is an initial Σ-algebra if and only if for any Σ-algebra, *A*, there is a unique homomorphism $f:I \rightarrow A$.

The word algebra is initial because there is just one way of creating the given expression using the operators described. Thus "TIMES(ZERO,PLUS(ZERO,ONE))" can only be created by applying the operator TIMES to the two expressions "ZERO" and "PLUS(ZERO,ONE)".

Initiality characterizes the isomorphism class of an object. This means it characterizes an object abstractly, that is, independent of representation or only in terms of its structure.

As another example, the data type of natural numbers, NATURAL, can be characterized by the initial Σ-algebra:

$S = \{NATURAL\}$
$\Omega_{\lambda,NATURAL} = \{0\}$
$\Omega_{NATURAL,NATURAL} = \{SUCC\}$
$\Omega_{u,s} = \emptyset$, otherwise

The basic idea is that every natural number and all further operations of interest upon natural numbers can be expressed in terms of just two basic ones, SUCC and 0. Initiality provides the key to unique characterization. There is no committment to

thinking of integers as strings of decimal, or binary, or Roman characters. To know the natural numbers, is to know the operations occurring in them and how they are combined. Further operations are obtained by recursive definitions expressed algebraically with equations, and based on initiality.

Given all of these definitions the next section describes how a specification for a data type is constructed.

2.6.5. Defining Specifications using Initial Algebras

In the initial algebra approach a *specification* is defined to be a triple <S,Ω,E>, where S is the set of sorts, Ω is the set of operators of the type and E is a set of equations which the algebra is to satisfy. The initial algebra that satisfies only what E requires and no more is chosen.

For example, the operators and equations of the stack abstraction are given below:

CREATE : → STACK
PUSH : STACK × INTEGER → STACK
POP : STACK → STACK ∪ {ERROR}
TOP : STACK → INTEGER ∪ {ERROR}

TOP(PUSH(s, i)) = i
POP(PUSH(s, i)) = s
TOP(CREATE) = ERROR
POP(CREATE) = ERROR

The set of equations specify when two expressions yield the same value. If an equation contains variables, such as *s* and *i*, the equation holds whenever each variable is replaced by an expression of the correct type. For example, the third equation

POP(PUSH(s, i)) = s

specifies that

POP(PUSH(CREATE, 1)) = CREATE , and
POP(PUSH(PUSH(CREATE, 1), 2) = PUSH(CREATE, 1)

The equations can be used in combinations to derive all the expressions which compute a particular value forming the equivalence class representing that value.

Using this approach, two expressions yield distinct results unless there is some sequence of applications of the equations which shows the expressions to be equal.

The definitions of some data abstractions require the use of conditional equations, that is, equations which do not hold for all possible substitutions of expressions for the variables, but which hold only for substitutions which satisfy some condition. The specification below for integer sets illustrates conditional equations.

EMPTY : → SET
INSERT : SET × INTEGER → SET
REMOVE : SET × INTEGER → SET
HAS : SET × INTEGER → BOOLEAN

INSERT(INSERT(s, i), j) = if i=j then INSERT(s, i)
else INSERT(INSERT(s, j), i)
REMOVE(INSERT(s, i), j) = if i=j then REMOVE(s, j)
else INSERT(REMOVE(s, j), i)
REMOVE(EMPTY, j) = EMPTY
HAS(INSERT(s, i), j) = if i=j then TRUE else HAS(s, j)
HAS(EMPTY, j) ≡ FALSE

The second equation is a good example of a conditional equation. There are two distinct outcomes that are possible when a REMOVE follows an INSERT. If the integer being REMOVEd is the same as the one being INSERTed, then that INSERT has no effect. If, however, the integer being REMOVEd is distinct from the integer INSERTed, then the REMOVE and the INSERT can be permuted.

2.6.6. For Further Help

If these brief explanations have not succeeded in their intent, the following references could be consulted for further examples and elucidation: [Meseguer&Goguen,1983], [Burstall&Goguen,1982], [Goguen,Thatcher&Wagner,1978], and [Goguen,Thatcher,Wagner&Wright,1975].

An overview of a programming language developed on these algebraic ideas will now be given.

2.7. The Programming Language OBJ

In OBJ [Goguen&Tardo,1979], algebras are the method for specification of data types. This is based on the notion that algebraic isomorphism captures independence of representation and the idea that initial algebras embodies it.

OBJ resembles more traditional languages in permitting definitions of operations and then evaluation of expressions. It differs in ways including: operations are defined in modules (objects) and have a different character in that operator symbols do not denote procedures in the usual sense, because they are defined implicitly by algebraic equations (using the initial algebra model). There is no assignment function, nor any assignable variables. There are no side effects, statements, or gotos. It is very strongly typed, supports sub-types and handles coercions, overloaded operators and errors in a rigorous, systematic manner.

The next section presents how data types, or objects, are defined in OBJ.

2.7.1. The Structure of OBJ Objects

The syntax for objects is inspired by the notation for presenting initial many sorted algebras. It is intended that an object denote a particular many sorted algebra.

An OBJ object is a means for declaring an abstract data type. There are three built in objects, INT for integers, BOOL for booleans and ID for identifiers. User defined objects begin with keyword OBJ and end with JBO. Objects are subdivided into sort, operation and equation sections. The sorts and operators together constitute a signature. Each operator declaration indicates its argument sorts, its target sort and its syntactic form. The equation section begins with a declaration of the variables to be used, including their sorts. A skeleton of the structure of an object is:

```
OBJ <name>
     SORTS <new sorts> / <old sorts>
     OK-OPS ...
     FIX-OPS ...
     ERR-OPS ...
     VARS ...
     OK-EQNS ...
     EQNS ...
     ERR-EQNS..
JBO
```

The SORTS line gives the sort names involved in the object. There are three different types of operators; OK-OPS are used for ordinary situations, FIX-OPS are for recovery situations and ERR-OPS are for exceptional or error situations.

The next section describes how these operators may be defined.

2.7.2. Definition of OBJ Operators

Operators are used in expressions, where an expression is either a constant operator, or an operator together with its arguments (which are also expressions). Every expression has a sort, the sort of its operator. Operators are declared in a manner commonly used in algebra:

G : INT, INT → INT
CREATE : → INT

The syntax of the operator is determined by its form to the left of the colon. Prefix form is assumed, so G would be invoked as G(2,G(4,5)). It is possible to customize the syntax of operators using underscores to indicate where the respective arguments go. For example, if an operator were defined as,

PUSH _ ON _ : INT, STACK → STACK

it would be invoked as,

PUSH 2 ON CREATE

Coercions are permitted with the form:

$_ : S_1 \rightarrow S_2$

indicating S_1 is to be a sub-sort of S_2. Note that the ability to customize syntax is not only a syntactic issue. The coercion above means that all items, but not necessarily operations, of sort S_1 are also of sort S_2 which is clearly a semantic statement.

Operating overloading, that is, distinct operators with the same syntactic form, are permitted:

```
_ + _ : BOOL, BOOL → BOOL
_ + _ : INT, INT → INT
```

Specifying the signature has now been taken care of, but what of the equations? The next section describes their form.

2.7.3. The Form of OBJ Equations

Each variable used in the equations must be listed, following the keyword VARS, together with its sort, as, for example:

```
VARS
    I, J : INT;
    R1, R2 : RECORD;
```

Equations are pairs of expressions of the same sort separated by an equal sign, such as (G(0,J)=J). Each equation is of some particular sort and its constituent expressions must be unambiguous.

Conditional form is allowed for equations. The condition must be of sort BOOL and appears following an IF in the form:

```
( G(I,J) = J IF (I=0) )
```

The conditional equation applies only if its conditional expression evaluates to true.

An example of specification of the factorial function is:

```
OBJ FACTORIAL
    SORTS / INT BOOL
    OK-OPS
        F : INT → INT
    ERR-OPS
        NEG-ARG : → INT
    VARS
```

```
        I : INT
    OK-EQNS
        ( F(0) = 1 )
        ( F(I) = I * F(I-1) IF (I>0) )
    ERR-EQNS
        ( F(I) = NEG-ARG IF (I<0) )
JBO
```

Now with some knowledge of how specifications are built in OBJ, the next section describes evaluation.

2.7.4. Evaluation in OBJ

OBJ evaluates, that is derives the value of, expressions by using the equations as rewrite rules. To apply a rule, first find a unifier must be found. This means that the interpreter tries to find an expression for each variable symbol occurring on the left side of the rule so that when these expressions are substituted for their respective variables, the original expression results. If the expression is conditional, evaluating the condition after substituting the expressions of the unifier for the variables must yield true.

When OBJ finds a rule with which it can unify an expression, it replaces the expression with one it obtains from the right side of the rule by substituting the expressions of the unifier for their respective variable symbols. In most cases, OBJ applies rules first to the innermost nested sub-expressions until no more rules apply, that is, until those sub-expressions are reduced or in what is called normal form. For example, evaluation of F(3) omitting trivial steps, proceeds as:

```
F(3)
3 * F(2)
3 * ( 2 * F(1) )
3 * ( 2 * ( 1 * F(0) ) )
3 * ( 2 * ( 1 * 1 ) )
3 * ( 2 * 1 )
3 * 2
6
```

Evaluation is complicated by the error facility. It has been proven [Goguen,1977] the above evaluation order correctly implements specifications containing error operators, provided all ERR-EQNS are tried before OK-EQNS.

OBJ evaluation computes, for any given expression, a reduced expression representing its equivalence class, and this expression is taken as a representative of the entire class. For example, the reduced value of expression F(3) is the expression 6.

In visualizing the connection between OBJs operational semantics and its initial algebra semantics, consider initial algebra semantics in terms of "word" algebras. With no equations, each word or expression, would denote a distinct value in the intended semantic domain. The equations identify those expressions which are to have equivalent meaning in the intended semantic domain.

2.7.5. Other Features of OBJ

Hidden operators are supported by OBJ. An operator may be declared hidden by placing (HIDDEN) after its syntactic definition. Hidden operators cannot be used outside the object in which they were declared, but may be used freely within it.

There are several evaluation modes some of them being: PERMUTING which remembers intermediate values resulting from successful rule applications to an expression, and RUN-WITH-MEMORY where all intermediate values for all evaluations are remembered, as though all operators were PERMUTING.

2.8. Introducing a New Technique

Now that an algebraic specification technique has been observed, along with a programming language based on this technique, Chapter 3 will evaluate this method with respect to the criteria outlined earlier. It will then supply an alternative specification technique – one based on recursive term algebras.

3. An Alternative Approach

3. Introduction

This chapter discusses the problems with the initial algebra approach to specification within the context established by the criteria given in Chapter 2. Then an approach based on recursive definition will be outlined along with how this method of definition can be realized as an unambiguous context free grammar.

3.1. Evaluation of the Algebraic Approach

The algebraic method of specification of data abstraction is, essentially, non-constructive. It characterizes a type by certain properties. In review, a data type is defined by its operations only. The operations are given as mappings satisfying certain relations. For an algebraic specification the axioms are allowed to be of a very restricted form only. For example, only those function and predicate symbols are admitted which stand for explicit operations on the data type. The existential quantifier does not occur and the axioms have the form of equations with left and right-hand sides built according to specific rules [Guttag,1975]. This method will now be evaluated with respect to the six criteria outlined in §2.5.

3.1.1. Formality and Constructibility

First off, there is no question as to the formality of the algebraic approach to the specification of data types. Underlying it is a well developed mathematical theory. Constructibility is, however, an issue, a difficulty admitted by the designers of OBJ [Goguen&Tardo,1979]. Problems that commonly arise are presented below.

3.1.1.1. The Problem of Infinite Sets of Equations

A well known argument against non-constructive methods is that it is very difficult

to find the characterizing set of axioms, or, in the case of algebraic axioms, the set of relations between the operations of the type. For the algebraic specification, it might not only be difficult but impossible to find a presentation of a type by operations and relations.

The question is whether the method is generally applicable, that is, is it possible to find such a finite specification for an arbitrary data type? An example adapted from [Majster,1977] to show that this is not the case is presented. The operators for the integer stack abstraction and its equations are:

CREATE : → STACK
PUSH : STACK × INTEGER → STACK
POP : STACK → STACK ∪ {ERROR}
TOP : STACK → INTEGER ∪ {ERROR)

TOP(PUSH(s, i)) = i
POP(PUSH(s, i)) = s
TOP(CREATE) = ERROR
POP(CREATE) = ERROR

To allow traversal of the stack stepwise from the top to the bottom and the ability to return to the top element from an arbitrary position the following additional operations are required. They perform, respectively, movement down the stack by one position, yield the contents of the stack at the current position, and position to the top of the stack.

DOWN : STACK → STACK ∪ {ERROR}
READ : STACK → INTEGER ∪ {ERROR}
RETURN : STACK → STACK ∪ {ERROR}

Now a finite characterizing set of relations between sequences of operations must be found. Needed first are[1]:

POP(CREATE) = ERROR
READ(CREATE) = ERROR
DOWN(CREATE) = ERROR
POP(DOWN(s)) = ERROR
PUSH(DOWN(s)) = ERROR
POP(PUSH(s, i)) = if PUSH(s, i)=ERROR then ERROR else s

[1] Assume the result of an operation applied to ERROR is ERROR.

READ(PUSH(s, i)) = if PUSH(s, i)=ERROR then ERROR else i

These seven relations, however, are not yet sufficient, as, for example, they do not express the fact that for a stack object consisting of n elements the $(n+1)$ fold application of DOWN would cause an error.

Hence, the relation[2]

$$(DOWN)^{k+1}(PUSH)^{k}(i_1, \ldots, i_k)=ERROR$$

is needed for $k=0,1,2,3\ldots$, that is, infinitely many relations.

Application of RETURN always causes positioning to the top element of the stack. Again, infinitely many equations of the form

$$RETURN(DOWN)^{m}(PUSH)^{n}(i_1, \ldots, i_n)=(PUSH)^{n}(i_1, \ldots, i_n)$$

for all $m \geq 1$ and $m < n$ are needed. That is, the position in the stack by first PUSHing n elements onto it, Figure 5(a), then moving DOWN the stack m positions $(m<n)$, Figure 5(b), followed by a RETURN, Figure 5(c), is the same as just PUSHing the n elements.

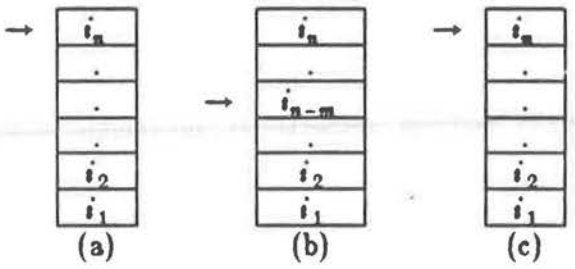


Figure 5: Sequence of stack configurations in performing $RETURN(DOWN)^{m}(PUSH)^{n}(i_1, \ldots, i_n)$

The specification is still not complete. The effect of the READ operation is only determined for objects of the form PUSH(s, i), that is, stacks that have had integers PUSHed onto them. An arbitrary stack object is,

$$(DOWN)^{q}(PUSH)^{r}(i_q, \ldots, i_r)$$

with $q<r$, hence the equations,

---

[2] Take $(PUSH)^{n}(i_1, \ldots, i_n)$ to be short hand for $PUSH(\cdots PUSH(CREATE, i_1)\cdots i_n)$

$$READ(DOWN)^q\ PUSH(s,i)=READ(POP)^q\ PUSH(s,i)$$

for $q>0$ are required. What this says is that given an arbitrary stack, of size r, the same element is retrieved in moving DOWN the stack q times $(q<r)$ and then performing the READ as when those q elements are POPped off the stack and then READ.

Three infinite sets of equations have already been produced. The problem is in determining whether any such equations have been omitted. Many other examples of specifications that require infinite sets of equations to express equivalences can be easily constructed.

Infinitely many relations might not be a handicap as long as the relations could be given in a parameterized form such as

$$A(m,n): RETURN(DOWN)^m(PUSH)^n(i_1,\ldots,i_n)=(PUSH)^n(i_1,\ldots,i_n)$$

Note, however, that in the theory developed for the algebraic technique the finiteness of the set of axioms is an important assumption.

3.1.1.2. Problems when the Church-Rosser Property is Absent

In the algebraic approach, as pointed out by Klaeren [Klaeren,1980], there is neither a general procedure which decides whether a certain set of algebraic axioms determines a unique operation, nor is it possible to give computation rules for the evaluation of operations. An obvious idea is to interpret the axioms as rewriting rules as done in the OBJ language. This, however, can lead to problems because systems of algebraic equations do not, in general, possess the Church-Rosser property [Meseguer&Goguen,1983], meaning it may be possible to get different results if the equations are applied in different ways.

It is possible, however, to give sufficient restrictions on the form of the axioms [Guttag&Horning,1978] to make sure that they define a unique total operation which can be computed by using the axioms as rewriting rules.

3.1.2. Comprehensibility and Minimality

Considering the problems just mentioned it appears the comprehensibility and minimality of the specification suffer greatly. In an attempt to avoid the problems of infinite sets of equations and to give sufficient restrictions to make the operations total the size of the specification grows and becomes unclear. Quite often avoiding inconsistency results in somewhat unnatural specifications. The user becomes trapped into giving these details and attention to the problem to be solved strays.

3.1.3. Range of Applicability and Extensibility

Using the algebraic approach an uncountable domain, such as the real numbers, cannot be defined without using an uncountable set of primitive constructors.

The onus is on the user not to introduce operations that lead to the generation of infinite sets of equations. In addition, if the axioms are interpreted as rewriting rules, the form of the axioms must be restricted to make sure they define a unique total operation if consistent results are expected.

Overall then, the addition of a new operation to Ω may mean augmenting the set of equations considerably. Hence, extending a specification may involve much effort.

3.1.4. Conclusions of the Evaluation

There appear to be some difficulties in using non-constructive techniques for the specification of data types, to which the initial algebra approach, as a member of this class, is subject.

In particular, the difficulties surrounding the construction procedure limits this technique to those well versed in algebra.

The next section introduces an alternative approach, using recursive term algebras, for which it will be shown that these difficulties can be either reduced or completely

eliminated.

3.2. The Recursive Term Algebra Approach

In order to get a non-constructive specification of a data type one usually has to use some constructive description to find the relations or axioms. Constructive methods explicitly state the appearance of objects and how the operations affect them. A constructive algebraic method, using recursive term algebras, will now be investigated. It is based on the work of Kanda and Abrahamson [Kanda&Abrahamson,1983].

3.2.1. Term Algebras versus Initial Algebras

In the initial algebra approach, a specification is defined to be the triple $<S,\Omega,E>$, where S is a set of sorts, Ω is the set of operators of the type, and E is the set of equations which the algebra has to satisfy.

To guarantee the uniqueness of the specification $<S,\Omega,E>$ the initial algebra which satisfies E as the specified data type is chosen. This specification method is based on the philosophy that carrier sets (sets obtained as the interpretation of sorts) of a data type should be induced from the equational properties of the operations.

An alternative specification method based on an alternative philosophy accepts only recursively defined sets. Recursive definition not only defines a set of terms but also basic operations over them. That is, it induces an algebra of terms.

How do these methods compare?

Consider the following recursive definition of the natural numbers, NATURAL.

Basis : 0 is a NATURAL.
Recursive Construction Rule : If n is a NATURAL then so is s(n).
Closure : Only those values that result from finitely many applications of the recursive construction rule to the basis function values are NATURALs.

This definition gives a set of terms over {0,s,(,),}. The operations of NATURAL from the recursive definition may now be induced. Notice closure ensures if *n* is a

NATURAL then either it is 0 or it is *s(m)* for some unique NATURAL *m*. This property gives a predicate, IS-ZERO on terms such that[3]:

IS-ZERO(n) = T, if n=0
= F, otherwise

The recursive construction rule shows how to compose and decompose terms. This gives rise to the following two operations on terms:

SUCC(n) = s(n)

PRED(n) = m, if n=s(m)
= undefined, otherwise

An algebra over the terms of NATURAL has been obtained.

Note partial operations like PRED[4] are bound to show up, consequently partial algebras. To overcome any problems this might possibly cause, it is only necessary to agree that undefinedness propagates throughout expressions. All operations of term algebras can be made total by adding *undefined* to each sort and making undefinedness propagation explicit. For example:

IS-ZERO(n) = T, if n=0
= undefined, if n=undefined
= F, otherwise

SUCC(n) = s(n), if n≠undefined
= undefined, otherwise

PRED(n) = m, if n=s(m)
= undefined, otherwise

Term algebras canonically obtained from recursive definition, as above, will be called *recursive term algebras.*

Comparing this with the initial algebra specification below:

$S = \{NATURAL\}$
$\Omega_{\lambda,NATURAL} = \{0\}$
$\Omega_{NATURAL,NATURAL} = \{s\}$
$\Omega_{u,s} = \emptyset$, otherwise

[3] Assume recursive term algebras have the sort BOOL as a carrier.

[4] PRED is a partial operation because it is not possible to perform PRED(0).

$E = \emptyset$

the set of natural numbers {0, s(0), s(s(0)),...} and the successor function can be given as the interpretation of NATURAL and *s*, respectively. Notice that the predecessor function cannot be obtained because the initial algebra will not give any operations not already in Ω. To get PRED in the initial algebra specification, a new initial algebra $<S,\Omega',E'>$ must be given, as,

$S = \{NATURAL\}$
$\Omega'_{\lambda,NATURAL} = \{0\}$
$\Omega'_{NATURAL,NATURAL} = \{s,p\}$
$\Omega'_{u,s} = \emptyset$, otherwise

which includes the symbol *p* for the predecessor operation. This, however, is not good enough because there are now too many terms representing the same natural number. For example, *s(0)*, *s(p(s(0)))*, and *p(s(s(0)))* all denote one. So a set of equational axioms is needed which enforces different terms denoting the same object to be equal. Since the initial algebra approach only works for total algebras, the predecessor operation must be totalized with the aid of undefined elements. The set E′ of axioms that is sufficient is

s(undefined) = undefined

p(0) = undefined
p(undefined) = undefined

ok(0) = T
ok(s(n)) = ok(n)
ok(undefined) = F

IF-EQUAL(T, n, m) = n
IF-EQUAL(F, n, m) = m
IF-EQUAL(undefined, n, m) = undefined
IF-EQUAL(ok(n), p(s(n)), undefined) = IF-EQUAL(ok(n), n, undefined)
IF-EQUAL(ok(n), s(p(s(n))), undefined) = IF-EQUAL(ok(n), s(n), undefined)

Really, all E′ is doing is characterizing the following four properties:

1) PRED(0) is undefined
2) undefinedness propagates
3) PRED(SUCC(n)) = n
4) if $n \neq 0$ then SUCC(PRED(n)) = n

The initial algebra satisfying E′ interprets NATURAL, p and s to be the set of natural numbers, and the predecessor and successor functions respectively.

In recursive term algebra specification, one need only make sure that a recursive definition of terms provides a unique representation for each element of the intended set. Then, a sufficient collection of operations follows automatically.

It is well known that there are some structures for each element of which it is impossible to provide a unique representation such as a finite power set. A discussion on how to handle these structures can be found in §7 of [Kanda&Abrahamson,1983].

In contrast, in the initial algebra specification, enough operations for the data type to be specified have to be chosen. Then by equational axioms, different terms are forced to denote the same object to form an equivalence class.

The equations E of the specification should not be used to define operations, but only for expressing certain properties such as commutativity, associativity and so on. The set Ω should be kept as small as possible.

As presented, the recursive term algebra approach might be criticized to be lacking in abstractness since the specification is dependent on term representations. The view taken here is that abstractness is finitely establishable properties of a finitely decidable collection of finitely examinable (concrete) objects. From this viewpoint, the method is sufficiently abstract.

Given that it has been decided to use recursive term algebras for the specification of data types, the next sections present how this form of definition can be realized as an unambiguous context free grammar.

3.3. Data Types as Context Free Grammars

The three basic elements of a recursive function definition are: a *basis* which states that certain values are, by definition, values of the function for given arguments, a

recursive construction rule which tells how to determine other values of the function from known values, and an understanding (or statement) that the function takes on only those values that result from finitely many applications of the recursive construction rule to the basis function values.

A formal grammar is a four-tuple G=(N,T,P,Σ) where:

N is a finite set of nonterminal symbols
T is a finite set of terminal symbols
N and T are disjoint, $N \cap T = \emptyset$
P is a finite set of productions
Σ is the start symbol, $\Sigma \notin (N \cup T)$

Each production in P is an ordered pair of strings, α, β,

$$\alpha = \phi A \psi$$
$$\beta = \phi \omega \psi$$

in which ω, ϕ and ψ are possibly empty strings in $(N \cup T)^*$ and A is Σ or a nonterminal letter. A production is usually written as:

$$\alpha \rightarrow \beta$$

Given this, a context free grammar G=(N,T,P,Σ) is a formal grammar in which all productions are of the form

$$A \rightarrow \omega \quad \begin{cases} A \in N \cup \{\Sigma\} \\ \omega \in (N \cup T)^* - \{\lambda\} \end{cases}$$

The grammar may also contain the production Σ→λ, where λ represents the empty string. That is, in context free grammars only one nonterminal is allowed to appear on the left hand side of the production. The right hand side of the production contains a rewrite rule that can be formed by concatenating an arbitrary number of symbols chosen from the nonterminals and terminals.

If G is a formal grammar, a string of symbols in $(N \cup T)^* \cup \{\Sigma\}$ is known as a sentential form. If $\alpha \rightarrow \beta$ is a production of G and $\omega = \phi\alpha\psi$ and $\omega' = \phi\beta\psi$ are sentential forms it is said that ω' is immediately derived from ω in G. This relation is indicated by writing $\omega \rightarrow \omega'$. If $\omega_1, \omega_2, \ldots, \omega_n$ is a sequence of sentential forms such that

$\omega_1 \rightarrow \omega_2 \rightarrow \ldots \rightarrow \omega_n$ it is said that ω_n is derivable from ω_1. This relation is indicated by writing $\omega_1 \rightarrow^* \omega_n$. The sequence $\omega_1, \omega_2, \ldots, \omega_n$ is called a derivation of ω_n from ω_1 according to G.

The language L(G) generated by a formal grammar G is the set of terminal strings derivable from Σ:

$$L(G) = \{ \omega \in T^* \mid \Sigma \rightarrow^* \omega \}$$

If $\omega \in L(G)$, it is said that ω is a string, a sentence, or a word in the language generated by G.

Given these definitions, how is recursive definition connected to context free grammars?

The terminal productions can be viewed as providing the basis of the recursive definition whereas the nonterminal production rules may be seen as the recursive construction rule. Closure follows automatically from the definition of $\rightarrow^*$.

For example, the basis for the recursive definition of the natural numbers, NATURAL, is:

Basis: 0 is a NATURAL.

From this the terminal production rule,

NATURAL $\rightarrow$ 0

may be built since zero is the only basis function value. The recursive construction rule for NATURAL is:

Recursive Construction Rule: If n is a NATURAL then so is s(n).

A nonterminal production rule describes this construction process. It is:

NATURAL $\rightarrow$ s(NATURAL)

The grammar, G, containing these two production rules gives a set of terms over {0,s,(,)}. In other words, the language generated by G is the set of terminal strings,

$$L(G) = \{ \omega \in T^* \mid \Sigma \rightarrow^* \omega \}$$

where T={0,s,(,)} and Σ=NATURAL.

Formally speaking then, recursive definition of sets is an unambiguous context free grammar. This has a few interesting results.

First, the data types are algebras over context free languages. Theoretically, an unambiguous context free grammar is important because it provides a computationally complete set of basic operations over the language it generates. This means that from the basic operations of term algebras, by iterative use of function composition, conditional definition and general recursive definition, it is possible to recursively define all partial computable functions over the terms recursively defined. If a decidable set is non-context free, there is no obvious way of providing a computational basis from a grammar of the set.

Second, algebras generated by unambiguous context free grammars are not just many sorted algebras. Each nonterminal N of a grammar G corresponds to a sort of the algebra *Alg(G)* generated by G. The carrier set of $Alg(G)_N$ of this sort in *Alg(G)* is the language $L(N) = \{ \omega \mid N \rightarrow^* \omega \}$. Therefore, if G has a production rule $N_1 \rightarrow N_2$ then $L(N_2) \subset L(N_1)$.

The importance of non-context free decidable data structures should not be neglected. The next section describes how these should be handled.

3.4. Sub-Data Types

In formal language theory it is well known that any decidable set S is a decidable subset of some context free set, CF. Also, it is easy to observe that a function $f:S \rightarrow CF$ is computable if and only if it is the restriction of a computable function $f:CF \rightarrow CF$. This indicates that the computability over S is inherited from that over CF, thus the data structure S should be treated within the structure of CF.

3.5. A Language Based on this Approach

An alternative specification technique for data structures in which recursive definition can be realized as unambiguous context free grammars has been given. In the next chapter, Chapter 4, this idea will be developed into a programming language, taking into account sub-data types as well.

4. The Functional Programming Language Kaviar

4. Introduction

The functional programming language that has been developed with context free grammars as data types has been named Kaviar.

Backus-Naur Form, abbreviated as BNF, is a well know meta-language for specifying the concrete syntax of a programming language. The intent of this chapter is to present the BNF of Kaviar and explain the operation of the interpreter.

4.1. Lexical Conventions

There are five classes of tokens: identifiers, keywords, constants, strings, and other separators. Each class is discussed below.

4.1.1. Identifiers

An identifier is a sequence of lower case letters and digits; the first character must be a letter. The dash - counts as a letter.

4.1.2. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

type
sub-type
is-
default

Other identifiers that are reserved, because they are the names of predefined functions which may not be redefined are:

read	le	add
eq	gt	subtract
ne	ge	multiply
lt		divide

4.1.3. Constants

There are only two kinds of constants available to the user: the integer constants of the predefined type *Nat,* and the two constants of the predefined type *Bool,* T and F, for true and false.

4.1.4. Strings

A string is a sequence of characters surrounded by double quotes, as in "...".

4.1.5. Separators

Blanks, tabs, newlines and comments, collectively called *white space,* are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.

The characters /* introduce a comment, which terminates with the characters */. Comments may be nested and may appear anywhere in a Kaviar program.

4.2. The BNF of Kaviar

The following sections present the BNF of Kaviar in understandable chunks along with examples and an informal explanation.

4.2.1. A Kaviar Program

A Kaviar program is a sequence of type definitions, followed by a sequence of function definitions.

<Program> ::= <Definitions> { <Invocation_Exprs> }

<Definitions> ::= <Type_Defns> <Function_Defns>

The type definitions precede function definitions only to enhance the speed of the type checking capabilities of the interpreter.

Once all the definitions have been completed, the invocation expressions follow, enclosed within braces.

The next section describes the definition of types.

4.2.2. Type Definitions

Context free data types as well as sub-types, which may be non-context free, are accomodated by Kaviar.

<Type_Defns> ::= <Context_Free_Types> <Sub_Data_Types>

The context free types are defined by a context free grammar, while sub-types are defined by predicates.

Since sub-types are, essentially, restrictions on context free types, they follow the definition of these types to simplify ensuring the predicate operates over some previously defined context free type.

Considered first will be the definition of context free types.

4.2.2.1. Context Free Data Types

Each context free type definition can be viewed as defining only one of the nonterminals N_i, $1 \leq i \leq k$, $N = \{N_1, N_2, \ldots, N_k\}$ of a grammar, G. Recall that there is no restriction on the number of alternatives a single nonterminal may have. That is, a nonterminal N_i may appear on the left hand side of arbitrarily many production rules.

In Kaviar, each alternative is further named by a *parser*. This facilitates, when given a terminal string ω, derived from a nonterminal $\{ \omega \mid N_i \rightarrow^* \omega \}$ finding which alternative of N_i ω was derived from. The parser names for each alternative of N_i must be distinct so that the alternative used in deriving ω can be uniquely determined.

As an example, consider the familiar LISP symbolic expression, or S-expression. Its recursive definition is:

Basis: An atom is an S-expression.
Recursive Construction Rule: If s_1 and s_2 are S-expressions, so is $(s_1.s_2)$.
Closure: Only those values that result from finitely many applications
of the recursive construction rule to the basis values are S-expressions.

An equivalent context free grammar over the atoms x and y is:

s-expr → atom
s-expr → (s-expr . s-expr)
atom → x
atom → y

As just described, each alternative of a given nonterminal, N_i, is further named by a parser. A nonterminal symbol will begin with <, followed by the nonterminal name, N_i, a colon, the parser name and then the closing >. Terminal symbols are strings and so will be enclosed within double quotes. Then, the grammar for S-expressions becomes:

<s-expr:atom> → <atom:a>
<s-expr:list> → "(" <s-expr:car> "." <s-expr:cdr> ")"
<atom:x> → "x"
<atom:y> → "y"

Since this form is lengthy, it is abbreviated by

```
type s-expr
{
     <atom> → <atom:a>;
     <list> → "(" <sexpr:car> "." <s-expr:cdr> ")"
}

type atom
{
     <x> → "x";
     <y> → "y"
}
```

and this is exactly the form the BNF for context free types in Kaviar generates. The interpreter converts this abbreviated form to the lengthier form.

The BNF for a context free type, then, is:

<Context_Free_Types> ::= <Context_Free_Type> |
<Context_Free_Type> <Context_Free_Types>

<Context_Free_Type> ::= type <Type_Name> { <Alternative> }

<Type_Name> ::= <Identifier>

<Alternative> ::= <Production> |

<Production> ; <Alternative>

<Production> ::= < <Parser> > → <Rewrite_Rule>

<Rewrite_Rule> ::= <Null_Rule> |
<Non_Null_Rule>

<Null_Rule> ::= λ

<Non_Null_Rule> ::= <Terminal> |
<Nonterminal> |
<Terminal> <Non_Null_Rule> |
<Nonterminal> <Non_Null_Rule>

<Terminal> ::= <String>

<Nonterminal> ::= < <Type_Name> : <Selector> >

<Selector> ::= <Identifier>

This allows for an arbitrary number of context free data types.

Notice that nonterminal symbols on the right hand side of the production rule are composed of two parts: a type name and a selector. The type name must be the name of a context free type, already defined or to be defined by <Context_Free_Type>, or one of the two predefined types, *Nat* or *Bool*. The selector is an identifier. Its use will become clear when recursive functions are discussed. Selector names for all the nonterminals appearing on the right hand side of a given alternative must be distinct.

Sub-type definitions follow context free type definitions. The next section describes these types.

4.2.2.2. Sub-Data Types

Suppose N_i is a context free type and

$$P(x : N_i) : \text{Bool}$$

is a defined total predicate. By

$$R = P(N_i)$$

a decidable subset

$$\{ x \mid x \in L(N_i) \cap P(x) = T \}$$

of $L(N_i)$ can be specified. Then R is called a *sub-type* of N_i. That is, a sub-type restricts a context free type N_i to only those values which satisfy the predicate P. These sub-types can be non-context free.

In Kaviar, a sub-type is specified as:

```
<Sub_Data_Types> ::= <Sub_Data_Type> |
                     <Sub_Data_Type> <Sub_Data_Types>

<Sub_Data_Type> ::= sub-type <Type_Name>
                    ( <Context_Free_Type_Name> )
                    = <Predicate> ;

<Context_Free_Type_Name> ::= <Identifier>
```

This allows for any number of sub-types the user wishes to define.

As mentioned already, the sub-type is a decidable subset of a context free type hence <Context_Free_Type_Name> must be the name of one of context free types. The BNF of the predicate that must be satisfied is given by:

```
<Predicate> ::= ( <Predicate> ) |
                ¬ <Predicate> |
                <Function_Name> |
                ( <Predicate> ) ∩ <Predicate> |
                ( <Predicate> ) ∪ <Predicate> |
                ¬ <Predicate> ∩ <Predicate> |
                ¬ <Predicate> ∪ <Predicate> |
                <Function_Name> ∩ <Predicate> |
                <Function_Name> ∪ <Predicate>

<Function_Name> ::= <Identifier>
```

That is, the predicate is an arbitrary logical formula. The functions referenced in the formula are restricted to be unary over a context free data type, being the same type the sub-type is defined over. In addition, these functions must have as their result type the predefined type *Bool*. This is so that the predicate itself can evaluate to true or false.

Consider the following context free type *number* defined as being a natural number:

```
type number
{
     <n> → <Nat:n>
```

}

and suppose the subset to be specified is

$$\{ x \mid x \in L(\text{number}) \cap x<5 = T\}$$

that is, that is *x* is a *number,* but is also less than five. This sub-type may be specified as:

sub-type lt-five(number) = less5;

So the functionality of *less5* must be:

less5(x : number) : Bool

Now that the construction of types has been dealt with, a discussion of how to define functions that operate over these types appears in the next sections.

4.2.3. Function Definitions

Function definitions in Kaviar have the form:

<Function_Defns> ::= <Fcn_Defn> |
<Fcn_Defn> <Function_Defns>

<Fcn_Defn> ::= <By_Recursion> |
<By_Unification>

Functions may be defined by one of two methods: by recursion, or by unification. Each method will be discussed in turn, accompanied by an example.

4.2.3.1. Recursive Functions

The BNF of recursive functions is:

<By_Recursion> ::= <Function_Name> (<Typed_Variable_List>)
: <Result_Type> { <Case_Stmts> }

<Typed_Variable_List> ::= <Identifier> : <Type_Name> |
<Identifier> : <Type_Name> , <Typed_Variable_List>

<Result_Type> ::= <Identifier>

<Case_Stmts> ::= <Case_Stmt> |
<Case_Stmt> ; <Case_Stmts>

```
<Case_Stmt> ::= is- <Parser> ( <Identifier> )
                → <Recursive_Exprs>

<Parser> ::= <Identifier>

<Recursive_Exprs> ::= <Recursive_Expr> |
                      <Recursive_Expr> <Recursive_Exprs>

<Recursive_Expr> ::= <Number> |
                     <String> |
                     <Identifier> |
                     <Truth_Value> |
                     <Selector> < <Identifier> > |
                     <Function_Name> ( <Rec_Par_List> )

<Rec_Par_List> ::= <Recursive_Exprs> |
                   <Recursive_Exprs> , <Rec_Par_List>
```

Suppose a recursive function was to be defined which takes an arbitrary S-expression and reverses it. The functionality of the function would be:

```
reverse( s : s-expr ) : s-expr
```

Ideally, a <Case_Stmt> should be given for each alternative of the type of the identifier in the parameter list to take care of any derivation of that type. The type *s-expr* was defined as:

```
type s-expr
{
      <atom> → <atom:a>;
      <list> → "(" <s-expr:car> "." <s-expr:cdr> ")"
}
```

which has two alternatives. If the value of *s* is an atom, reversing an atom yields itself so the <Case_Stmt> would be:

```
is-atom( s ) → s
```

If the value of *s*, however, is a list, reversing a list consists of concatenating the reverse of the tail end of the list, given by the derivation of the nonterminal <s-expr:cdr>, with the reverse of the head of the list, given by the derivation of the nonterminal <s-expr:car>. Hence, the <Case_Stmt> would look like:

```
is-list( s ) → "(" reverse( cdr<s> ) "." reverse( car<s> ) ")"
```

where the selectors *car* and *cdr* select the values corresponding to the derivations of their nonterminals. The operation of selectors will be discussed in more detail in a following section.

So the complete function is:

```
reverse( s : s-expr ) : s-expr
{
        is-atom(s) → s;
        is-list(s) → "(" reverse( cdr<s> ) "." reverse( car<s> ) ")"
}
```

If the function defined has more than one parameter a <Case_Stmt> which operates over any of the identifiers could be given, but only the <Recursive_Exprs> of the first applicable parser found will be evaluated. The notion of what an applicable parser is will be discussed in a later section.

Of course, each function defined must be given a unique name. Kaviar permits no operator, or function, overloading. The identifiers present in the <Typed_Variable_List> are the only ones that may appear within the function. The type of a given identifier in this list may be a context free type name, a sub-type name or one of the predefined type names. Likewise for the result type of the function. Before entry to the function each parameter is checked to ensure it is of the correct type. If it is not an error will report the value at fault.

If the type of an identifier is a context free type, N_i, the interpreter can determine whether the value, $\omega \in L(N_i)$ by running Earley's algorithm on it. A digression will now be taken to explain the operation of Earley's algorithm.

4.2.3.2. The Parsing Method of Earley

Earley's algorithm, developed in the early seventies, [Earley,1970], [Aho&Ullman,1972], [Graham&Harrison,1976], provides a means of determining whether a terminal string ω can be derived by some context free grammar. That is, whether $\omega \in L(G)$.

4.2.3.2.1. Conventions

The following conventions will be used to represent various symbols and strings concerned with a grammar while discussing Earley's algorithm.

(1) The lowercase letters a,b,c and d will represent terminal symbols, as will the digits $0,1,...,9$.

(2) The capital letters A,B,C and D represent nonterminal symbols.

(3) $U,V,...,Z$ represent either nonterminal or terminal symbols.

(4) The Greek letters $\alpha,\beta,...$ represent strings of nonterminal and terminal symbols. λ denotes the null string, and Σ represents the start symbol.

(5) $u,v,...,z$ represent strings of terminal symbols only.

Subscripts and superscripts do not change these conventions.

4.2.3.2.2. Terminology

Let $G=(N,T,P,\Sigma)$ be a context free grammar and let $\omega = a_1 a_2 \cdots a_n$, $n \geq 0$, $1 \leq j \leq n$, $a_j \in \Sigma$ be the input string. An **item** of the form: $[A \rightarrow X_1 X_2 \cdots X_k \bullet X_{k+1} \cdots X_m, i]$ represents:

(1) A production $A \rightarrow X_1 \cdots X_m$ in P such that a portion of the input string which is derived from its right side is currently being scanned.

(2) A point in that production called the metasymbol, denoted by the symbol $\bullet$ not in N or Σ, which shows how much of the production's right side has been recognized so far. The integer k can be any number ranging from zero, in which case $\bullet$ is the first symbol, through $m+1$, in which case it is the last. A production of the form $A \rightarrow \alpha \bullet \beta$ will be called a **dotted rule.**

(3) A pointer i, $0 \leq i \leq n$, back to the position in the input at which that instance of the production was first looked for.

For each integer j, $0 \leq j \leq n$, a set of items I_j will be constructed such that [A→$\alpha \bullet \beta$,i] is in I_j for $0 \leq i \leq j$ if and only if for some γ and δ, $\Sigma \rightarrow^* \gamma A \delta$, $\gamma \rightarrow^* a_1 \cdots a_i$ and $\alpha \rightarrow^* a_{i+1} \cdots a_j$. In other words, if the dotted rule A→$\alpha \bullet \beta$ is in I_j then work is proceeding on a potentially valid parse in that it is known that there is a sentential form $\gamma A \delta$ where $\gamma \rightarrow^* a_1 a_2 \cdots a_i$. Furthermore, A→$\alpha \bullet \beta$ indicates that $\alpha \rightarrow^* a_{i+1} \cdots a_j$. Nothing yet is known about β or δ. That is, A→$\alpha\beta$ could be used in some input sequence that is consistent with ω up to position j.

The number of distinct items in I_j is determined by the grammar and the input string. The sequence of lists $I_0, I_1, \ldots, I_n$ will be called the **parse lists** for the input string ω.

4.2.3.2.3. Informal Explanation of Earley's Algorithm

Earley's algorithm works on any context free grammar. The grammar need not be in any normal form. The function of the algorithm is to form the parse lists.

The operation of the algorithm is as follows. An input string $a_1 \cdots a_n$ is scanned from left to right. As each a_j is scanned, a set of items for I_j is constructed which represents the condition of the recognition process at that point in the scan. In general, parse list I_j is operated on by processing the items, *in order*, performing one of three operations depending on the form of the item.

The **predictor** operation is applicable when the form of the item is [A→$\alpha \bullet B \beta$,i], that is, when there is a nonterminal to the right of the metasymbol. It causes the addition of a new item of the form [B→$\bullet \gamma$,j] to I_j for each alternative of that nonterminal, here B. The metasymbol is placed at the beginning of the production in each new item since none of its symbols has yet been scanned. The pointer is set to j since the item was created in I_j.

Thus the predictor adds to I_j all productions which might generate substrings beginning at a_{j+1}. In other words, it indicates which rules might possibly generate the

next portion of the input since the dotted rules introduced by the predictor represent rules that could be used in the derivation since their left hand sides occur in valid sentential forms.

The **scanner** serves to update those dotted rules (or partial subtrees) whose *expected* next element is the next input symbol. In other words, the scanner is applicable just in the case the item has the form [B→α•aβ,i] that is, when there is a terminal to the right of the metasymbol. It compares that symbol with a_j and if they match the metasymbol is moved over one in the item, as in [B→αa•β,i], to indicate that that terminal symbol has been scanned and then adds the item to I_j. The pointer remains the same because the item is not being created, only updated.

The third operation, the **completer,** has as its role to update those productions whose *expected* next element is a nonterminal which generates a suffix of the input read thus far. That is, the completer is applicable to an item if its metasymbol is at the end of its production, as in [A→α•,i]. It goes to the parse list I_i indicated by the pointer and looks for all items from it that have the nonterminal on the left hand side, here A, to the right of the metasymbol as occurs in [B→α•Aβ,k]. It moves the metasymbol over the nonterminal in these items [B→αA•β,k] and adds them to I_j. I_i can be thought of as the parse list being operated on when A was being looked for. It has now been found, so for all the items in I_i which caused search for an A the metasymbol must be moved over it in them to show that it has been successfully scanned.

Earley's algorithm is, in effect, a top down parser in which all the parse lists generated represent possible parses carried along simultaneously.

4.2.3.2.4. Earley's Parsing Algorithm

Input: A context free grammar G=(N,T,P,Σ) and an input string $\omega = a_1 a_2 \cdots a_n$ in Σ^*.

Output: The parse lists $I_0, I_1, \ldots, I_n$.

Method: First, construct I_0 as follows:[1]

[1] If $\Sigma \rightarrow \alpha$ is a production in P, add $[\Sigma \rightarrow \bullet\alpha, 0]$ to I_0.

Now perform steps [2] and [3] until no new items can be added to I_0.

[2] If $[B \rightarrow \gamma\bullet, 0]$ is in I_0 add $[A \rightarrow \alpha B\bullet\beta, 0]$ for all $[A \rightarrow \alpha\bullet B\beta, 0]$ in I_0.

[3] Suppose that $[A \rightarrow \alpha\bullet B\beta, 0]$ is an item in I_0. Add to I_0, for all productions in P of the form $B \rightarrow \gamma$, the item $[B \rightarrow \bullet\gamma, 0]$.

Now construct I_j, having constructed $I_0, I_1, \ldots, I_{j-1}$.

[4] **Scanner.** For each $[B \rightarrow \alpha\bullet a\beta, i]$ in I_{j-1} such that $a = a_j$, add $[B \rightarrow \alpha a\bullet\beta, i]$ to I_j.

Now perform steps [5] and [6] until no new items can be added.

[5] **Completer.** Let $[A \rightarrow \alpha\bullet, i]$ be an item in I_j. Examine I_i for items of the form $[B \rightarrow \alpha\bullet A\beta, k]$. For each one found, add $[B \rightarrow \alpha A\bullet\beta, k]$ to I_j.

[6] **Predictor.** Let $[A \rightarrow \alpha\bullet B\beta, i]$ be an item in I_j. For all $B \rightarrow \gamma$ in P, add $[B \rightarrow \bullet\gamma, j]$ to I_j.

The algorithm, then, is to construct I_j for $0 \leq j \leq n$.

Since the number of parse lists constructed is of bounded size, it is easily shown that Earley's algorithm terminates.

If the underlying grammar is unambiguous, then this algorithm can be executed in $O(n^2)$ reasonably defined elementary operations when the input is of length n.

In all cases, it can be executed in $O(n^3)$ reasonably defined elementary operations when the input is of length n. For proofs of these and all other complexity

[1] Before adding an item to a parse list it is important to ensure that it is not a duplicate since this could lead to unbounded growth of the list.

figures regarding Earley's algorithm see [Aho&Ullman,1972].

4.2.3.2.5. Example of the Operation of Earley's Algorithm

Consider the following grammar G for generating symbolic expressions with the productions numbered as shown:

1: S → A
2: S → (S . S)
3: A → x
4: A → y

and let $\omega = (x . y)$ be the input string. Since Σ=S, the items

$[S \rightarrow \bullet A, 0]$ and (1)
$[S \rightarrow \bullet(S.S), 0]$ (2)

are the first to be added to I_0. The predictor is applicable to the first of these items. Operating on (1) it produces

$[A \rightarrow \bullet x, 0]$ and
$[A \rightarrow \bullet y, 0]$

I_0 is now complete so move on to construct I_1. The scanner is applicable to (2) since the terminal to the right of the metasymbol matches a_1=(. The item

$[S \rightarrow (\bullet S.S), 0]$

is added to I_1. Notice that the metasymbol has been moved over one in the item to indicate that the symbol has been scanned. The predictor now causes

$[S \rightarrow \bullet A, 1]$ and (3)
$[S \rightarrow \bullet(S.S), 1]$

to be added. The predictor is once again applicable to (3) and adds

$[A \rightarrow \bullet x, 1]$ and
$[A \rightarrow \bullet y, 1]$

to I_1. No further items can be added to I_1.

In constructing I_2 note that a_2=x and so the scanner adds

$[A \rightarrow x \bullet, 1]$

to I_2. The completer is applicable to this state because the metasymbol is at the end of its production. It goes back to the parse list indicated by the pointer, in this case I_1, and looks for all states from I_1 which have A to the right of the metasymbol. The only item meeting these specifications is [S→•A,1] in I_1 and so:

$$[S \rightarrow A \bullet,1] \tag{4}$$

is added to I_2 moving the metasymbol over A in the item. Considering (4) causes the completer to reexamine I_1, this time searching for items with the metasymbol preceding S in them. It can add one more item to I_2

$$[S \rightarrow (S \bullet . S),1]$$

This completes I_2.

A complete run of the algorithm on grammar G is given in Table I:

Table I: Parse Lists for G with ω=(x.y)

I_0	I_1	I_2
[S→•A,0]	[S→(•S.S),0]	[A→x•,1]
[S→•(S.S),0]	[S→•A,1]	[S→A•,1]
[A→•x,0]	[S→•(S.S),1]	[S→(S•.S),0]
[A→•y,0]	[A→•x,1]	
	[A→•y,1]	
I_3	I_4	I_5
[S→(S.•S),0]	[A→y•,3]	[S→(S.S)•,0]
[S→•A,3]	[S→A•,3]	
[S→•(S.S),3]	[S→(S.S•),0]	
[A→•x,3]		
[A→•y,3]		

It is possible to determine membership in the language by inspecting the items of the last parse list. If no item of the form [$\Sigma \rightarrow \alpha \bullet$,0] is in I_n, then ω is not in L(G).

4.2.3.2.6. Modifications Required to Earley's Parsing Algorithm

Recall that in Kaviar all production rules are named by parsers. Their form is:

$$<A{:}p_i> \rightarrow \beta$$

Also, all nonterminals in β have selector names and have the form $<B:s_j>$. Because of this, Earley's algorithm requires some adjustments. The algorithm that works is given below.

Input: A context free grammar G=(N,T,P,Σ) and an input string $\omega = a_1 a_2 \cdots a_n$ in Σ^*.

Output: The parse lists $I_0, I_1, \ldots, I_n$.

Method: First, construct I_0 as follows:

[1] If $<\Sigma:p_i> \rightarrow \alpha$ is a production in P, add $[<\Sigma:p_i> \rightarrow \bullet\alpha,0]$ to I_0.

Now perform steps [2] and [3] until no new items can be added to I_0.

[2] If $[<B:p_i> \rightarrow \gamma\bullet,0]$ is in I_0 add $[<A:p_j> \rightarrow \alpha<B:s_k> \bullet\beta,0]$ for all $[<A:p_j> \rightarrow \alpha\bullet<B:s_k>\beta,0]$ in I_0.

[3] Suppose that $[<A:p_i> \rightarrow \alpha\bullet<B:s_j>\beta,0]$ is an item in I_0. Add to I_0, for all productions in P of the form $<B:p_k> \rightarrow \gamma$, the item $[<B:p_k> \rightarrow \bullet\gamma,0]$.

Now construct I_j, having constructed $I_0, I_1, \ldots, I_{j-1}$.

[4] **Scanner.** For each $[<B:p_i> \rightarrow \alpha\bullet a\beta,i]$ in I_{j-1} such that $a = a_j$, add $[<B:p_i> \rightarrow \alpha a\bullet\beta,i]$ to I_j.

Now perform steps [5] and [6] until no new items can be added.

[5] **Completer.** Let $[<A:p_i> \rightarrow \alpha\bullet,i]$ be an item in I_j. Examine I_i for items of the form $[<B:p_j> \rightarrow \alpha\bullet<A:p_k>\beta,k]$. For each one found, add $[<B:p_j> \rightarrow \alpha<A:p_k> \bullet\beta,k]$ to I_j.

[6] **Predictor.** Let $[<A:p_i> \rightarrow \alpha\bullet<B:p_j>\beta,i]$ be an item in I_j. For all $<B:p_k> \rightarrow \gamma$ in P, add $[<B:p_k> \rightarrow \bullet\gamma,j]$ to I_j.

It is now known how the interpreter determines whether an input string is generated by some context free grammar but what if it is a sub-type? The next section describes this process.

4.2.3.2.7. Type Checking Sub-Data Types

If the type of the identifier in the <Typed_Variable_List> is a sub-type, S_i, the interpreter can determine whether the value is of the correct type by first running Earley's algorithm with Σ being the name of the <Context_Free_Type_Name> the sub-type is defined over. Then, the functions in the <Predicate> are interpreted resulting in some logical formula. If this resulting logical formula evaluates to true the type of the value is indeed S_i.

4.2.3.2.8. Interpreting a Recursive Function

If all the parameters of a function are of the correct type, the interpreter proceeds to determine the <Case_Stmt> to execute. Suppose the function *reverse* is invoked as:

reverse("(" "x" "." "y" ")")

The terminal string "(" "x" "." "y" ")" $\in$ L(s-expr) so for the duration of the function the identifier *s* has this value. The parsers in a given <Alternative> must be valid for the type of the identifier it operates over. For example, the type of *s* is *s-expr* so the only parsers that may operate on *s* are those that appeared in the definition in the type *s-expr*. Looking back at the definition of *s-expr* the available parsers are found to be *atom* and *list*.

If the type of the identifier is a sub-type, the applicable parsers are those of the context free type the sub-type is defined over. For example, if an identifier had type *lt-five* which is defined over the context free type *number*, the only applicable parser would be *n*.

If the type of the identifier is the predefined type Bool, the user has two parsers available for use. They are *true* and *false* just as if the type Bool had been defined as:

```
type Bool
{
```

<true> → T;
<false> → F
}

No parsers are provided for type Nat because there are an infinite number of them.

The interpreter steps through the <Case_Stmts> of the function one by one until the applicable one is found. How is a <Case_Stmt> determined applicable?

First, the <Parser> of the <Case_Stmt> is extracted, call it *parser*. Then the alternative with this parser name is extracted from the type definition. This might be

<type:parser> → β

Then, if an item of the form [<type:parser>→β•,0] is found on the last item list, I_n, for the value of the identifier the parser is operating over, then the alternative <type:parser>→β was the one the value was derived from. This <Case_Stmt> is then termed applicable. The interpreter would then go on to execute the <Recursive_Exprs> of this <Case_Stmt>.

Otherwise, the interpreter would move on to the next <Case_Stmt> to determine if it is applicable. If the interpreter exhausts all of the <Case_Stmts> without finding one to be applicable an error will result.

Supposing an applicable parser is found, the next section describes how its <Recursive_Exprs> are evaluated by the interpreter.

4.2.3.2.9. Interpreting Recursive Expressions

There are six types of recursive expressions. The expressions occurring are evaluated in sequence and the results concatenated together. The final result must be of type <Result_Type>. Numbers of type *Nat*, truth values of type *Bool* and strings require no interpretation. The interpretation of an identifier is the value assigned to it on entry to the function.

Selectors must operate on the same identifier as the parser, and can only be the ones present in the alternative of the parser. For example, the alternative *list* of type

s-expr is:

<list> → "(" <s-expr:car> "." <s-expr:cdr> ")"

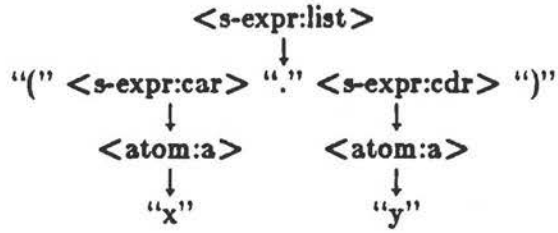
On entry to *reverse*, the value of *s* was set to "(" "x" "." "y" ")". The interpreter can determine that this value was derived from the alternative <list>, so the only selectors that may be active are *car* and *cdr* since they are the ones that appear in the definition of the alternative. Recall, each nonterminal on the right hand side of the production rule has the form:

< <Type_Name> : <Selector> >

The <Selector> may be used as a function that operates over the value of the identifier the parser was applicable to. The effect of a selector, as used in the following extract from the function *reverse*

is-list(s) → "(" reverse(cdr<s>) "." reverse(car<s>) ")"

is to select, from the value of the identifier, the subtree corresponding to the derivation of the nonterminal. The type of the subtree will, of course, be <Type_Name>. For example, the derivation tree of the value of *s* is:



The selector returns the value of the subtree. So *car*<*s*> = *"x"* and *cdr*<*s*> = *"y"*.

If the expression is a function invocation, its parameters are first evaluated and then the function is called. The result of the function is then concatenated to the partial result.

When all of the expressions on the right hand side of the arrow have been evaluated the resulting terminal string is checked to see if it is of type <Result_Type> and if so, it is returned, otherwise an error is reported.

The evaluation of

reverse("(" "x" "." "y" ")")

would then proceed as follows. The parser *is-list(s)* would be found applicable. Its expressions are:

"(" reverse(cdr<s>) "." reverse(car<s>) ")"

The string "(" becomes the partial result. The parameter to *reverse*, *cdr<s>*, is evaluated, yielding "y", so reverse("y") is invoked. It returns "y" as a result. This result is concatenated onto the partial result, the partial result now becoming "(" "y". Continuing in this fashion the final result is the terminal string "(" "y" "." "x" ")", which is of type *s-expr*.

Now that the operation of recursive functions is understood, the next sections describe the second type of function definition - by unification.

4.2.3.3. Unification Functions

Functions may, alternatively, be defined by unification. The BNF for these types of functions is:

```
<By_Unification> ::= <Function_Name> ( <Type_Name> )
                     : <Result_Type> { <Unif_Clauses> }

<Unif_Clauses> ::= <Clause> |
                   default → <Unif_Exprs>

<Clause> ::= <Schema> → <Unif_Exprs> |
             <Schema> → <Unif_Exprs> ; <Clause> |
             <Schema> → <Unif_Exprs> ; default → <Unif_Exprs>

<Unif_Exprs> ::= <Unif_Expr> |
                 <Unif_Expr> <Unif_Exprs>

<Unif_Expr> ::= <Number> |
                <String> |
                <Identifier> |
                <Truth_Value> |
                <Function_Name> ( <Unif_Par_List> )

<Unif_Par_List> ::= <Unif_Exprs> |
                    <Unif_Exprs> , <Unif_Par_List>
```

```
<Schema> ::= <String> |
             <Identifier> |
             <String> <Schema> |
             <Identifier> <Schema>
```

Unification can be intuitively looked at as being pattern matching. In Kaviar, unification is only one way meaning variables may only occur in the <Schema> and not in what it is being matched against.

All unification functions are unary since otherwise it would be impossible to tell what the <Schema> of a given <Clause> should be unified against.

The next section describes how to build a <Schema> so that the interpreter will be able to successfully unify expressions.

4.2.3.4. Finding a Unifier

Finding a unifier consists of finding an expression for each identifier symbol occurring in the <Schema> so that when these expressions are substituted for their respective identifiers the value of the original parameter results. To do this successfully, the <Schema> must mimic the form of the alternatives of the type.

Ideally, there should be a <Clause> for each alternative of the type so that all possible derivations of the type may be handled. The <Schema> is built by specifying a string for every terminal symbol in the <Rewrite_Rule> and an identifier for every nonterminal symbol.

The definition of S-expressions was given as:

```
type s-expr
{
      <atom> → <atom:a>;
      <list> → "(" <s-expr:car> "." <s-expr:cdr> ")"
}
```

A unification function which reverses arbitrary S-expressions would have the functionality:

```
reverse( s-expr ) : s-expr
```

so the schemas of the clauses should mimic the alternatives of *s-expr* since this is the type of the parameter.

The right hand side of the alternative <atom> has only one nonterminal, so the schema would consist of only one identifier. Reversing an atom yields itself so the <Clause> would be:

atom → atom

The right hand side of the second alternative, <list>, contains terminal and nonterminal symbols. Working from left to right, simply copy terminal symbols and assign nonterminal symbols distinct identifiers. So for <list> the <Schema> would be:

"(" s_1 "." s_2 ")"

Reversing a list requires concatenating the reverse of the tail end of the list with the head of the list. Thus the complete function would be:

```
reverse( s-expr ) : s-expr
{
      atom → atom;
      "(" s1 "." s2 ")" → "(" reverse( s2 ) "." reverse( s1 ) ")"
}
```

The interpreter finds a unifier by going through the <Unif_Clauses> of the function one by one trying to find which <Schema> the value of the parameter can be unified against.

The alternative the parameter value was derived from is what the <Schema> is compared to. For example, the alternative the terminal string "(" "x" "." "y" ")" was derived from is:

<list> → "(" <s-expr:car> "." <s-expr:cdr> ")"

Identifiers can be unified against nonterminal symbols and terminal strings must match exactly.

If unification can take place, the interpreter proceeds to evaluate the <Unif_Exprs> to the right of the arrow, binding any identifiers in the <Schema> to

the value of the subtree the nonterminal they stand for derives. It makes sense that the only identifiers that can appear in the <Unif_Exprs> are those that appeared in the <Schema>.

If the reserved word *default* is found instead of a <Schema> the <Unif_Exprs> to the right of the arrow will be evaluated immediately.

If no <Schema> can be unified against the interpreter will indicate failure.

Assuming a unifier is found, the interpreter goes on to evaluate the unification expressions. The next section describes this.

4.2.3.5. Evaluating Unification Expressions

The unification expressions are evaluated in the exact same manner as recursive expressions, except for the fact that the selector functions cannot be used.

Once all the types and functions are defined there must be some way of using all these definitions. The next section describes the invocation expressions.

4.2.4. Invocation Expressions

The BNF of the invocation expressions is:

```
<Invocation_Exprs> ::= <Invoc_Expr> |
                       <Invoc_Expr> ; <Invocation_Exprs>

<Invoc_Expr> ::= Nl |
                 <String> |
                 <Identifier> |
                 read ( <Typed_Variable_List> ) |
                 <Function_Name> ( <Parameter_List> )

<Parameter_List> ::= <Parameters> |
                     <Parameters> , <Parameter_List>

<Parameters> ::= <Parameter> |
                 <Parameter> <Parameters>

<Parameter> ::= <String> |
                <Number> |
                <Identifier> |
```

<Truth_Value> |
<Function_Name> (<Parameter_List>) |

Input may be prepared for a Kaviar program by placing sentential forms in a data file, each one terminated by the character ".". The values may be read from the data file by issuing a *read(* $x_1 : t_1$,..., $x_n : t_n$ *)* invocation expression. The value read will be associated with the identifier x_i if its type is indeed t_i.

For example, the data file for the read statement:

read(s_1 : s-expr, s_2 : s-expr)

might look like:

"(" "x" "." "y" ")" .
"y" .

When a string is given as an invocation expression it will simply be written to standard output. Similarly, when an identifier is given the value associated with it by some read statement will be written to standard output.

When *Nl* is encountered a newline will be written to standard output.

As in interpreting recursive and unification expressions, parameters to a function are first evaluated, if necessary, and then the function is invoked. Upon return, the result of the function will be written to standard output.

4.2.5. Built in Functions

There are several built in functions available to the user. The basic arithmetic functions all have two parameters of type *Nat* with result type *Nat*. They are:

```
add( x, y )       /* x + y   */
subtract( x, y)   /* x - y   */
mutiply( x, y )   /* x * y   */
divide( x, y )
```

Divison is integer division in which the fractional part is truncated. The expression *divide(x,y)* produces the remainder when x is divided by y, and thus is zero when y divides x exactly.

The other built in functions take two arguments of type *Nat* and produce a *Bool* result. They are:

```
eq(x,y)      /* x = y */
ne(x,y)      /* x ≠ y */
le(x,y)      /* x ≤ y */
lt(x,y)      /* x < y */
ge(x,y)      /* x ≥ y */
gt(x,y)      /* x > y */
```

4.3. How to Run a Kaviar Program

A Kaviar source program is expected to come from standard input. All output is directed to standard output. There are three options available to the user.

The *-q* option removes double quotes from values the invocation expressions write to standard output. This makes the output look tidier and would be used when the user is sure the program is working correctly.

The *-s* option stops sub-type checking. In normal circumstances the interpreter verifies that something is a sub-type by evaluating a predicate. When this option is given it will bypass evaluating the predicate. Once a program is fully debugged this option might be used to enhance the speed of the interpreter.

And finally there is the *-d* option. If the Kaviar program has any read statements in it the interpreter expects that the user has prepared a data file. The name of this file is to be given immediately after the *d* as in:

```
-dname
```

4.4. Implementation Restrictions and Details

The following are restrictions which have been arbitrarily set. To change these limits, the constants need only be changed and the interpreter recompiled.

No more than the first SIG_CHARS (12) characters of an identifier is significant, although more may be used.

At most MAX_CF_TYPES (50) and MAX_SUB_TYPES (50), context free and sub-types, respectively, may be defined in any one source program. Also, at most MAX_FCN (50) functions by either method may be defined.

The maximum length of a string is MAX_STR_LEN (256) counting the opening and closing quotes. There is currently no way of including double quotes within a string. Also, the character % should not be used in strings since the Unix routine *printf* is used for output and the % introduces a conversion specification.

The interpreter does no checking for overflow in evaluating the built in functions. Behaviour of the interpreter in these cases is system dependent.

Although the symbol $\rightarrow$ has been used throughout the BNF for an arrow, the implementation recognizes an arrow as the concatenation of the characters = and >. Similarly, the symbol $\cup$ and $\cap$ have been used for logical *or* and *and*, respectively, but the implementation recognizes the characters | and & for these connectives.

4.5. Evaluating this Approach

Having introduced an alternative method for data abstraction, along with a programming language using this method, it is time to evaluate it. Chapter 5 does tthis evaluation and provides a comparison with the initial algebra approach.

5. Evaluation of the Recursive Term Algebra Approach

5. Introduction

As promised, this chapter will evaluate the recursive term algebra approach for specifying data types. It will follow the criteria given in §2.5 and show its superiority over the initial algebra approach by means of examples.

5.1. Formality and Constructibility

The formality of the recursive term algebra approach for specifying data types cannot be disputed.

Constructing specifications is a straightforward task. One need only translate a recursive definition to an unambiguous context free grammar by means described in §3.3.

Once again, consider the integer stack abstraction. In Kaviar, this might be defined as:

```
type stack
{
        <error> → "error";
        <empty> → "empty";
        <not-empty> → <list:data>
}
```

The alternative <error> is introduced for error handling.

A non-empty stack's contents is described by the type *list:*

```
type list
{
        <int> → <Nat:n>;
        <list-of-int> → <Nat:hd> <list:tl>
}
```

One other type is needed to describe the values the data portion a non-empty stack may be. It is:

```
type data
{
        <empty> → "empty";
        <list> → <list:list-of-int>
```

```
}
```

The basic operations of a stack are:

```
CREATE : → STACK
POP : STACK → STACK ∪ {ERROR}
PUSH : STACK × INTEGER → STACK
```

In Kaviar these operations would be written as functions. The CREATE operation creates an empty stack. This function may be written as:

```
create( Bool ) : stack
{
      default → "empty"
}
```

The PUSH operation might be written as:

```
push( s : stack, i : Nat ) : stack
{
      is-error( s ) → s;
      is-empty( s ) → i;
      is-not-empty( s ) → i data<s>
}
```

As usual, the result of an operation applied to ERROR is ERROR. PUSHing a value onto an empty stack yields a stack containing that single value. If the stack is not empty, PUSHing a value onto it corresponds to prepending the value to the stack.

The POP operation could be written as:

```
pop( s : stack ) : stack
{
      is-error( s ) → s;
      is-empty( s ) → "error";
      is-not-empty( s ) → take-tl( data<s> )
}

take-tl( l : list ) : data
{
      is-int( l ) → "empty";
      is-list-of-int( l ) → tl<l>
}
```

If the stack is empty is it not possible to POP a value off it, so an error results. If the stack is not empty POPping off a stack with only one element yields an empty stack, whereas if the stack has more than one element, take the tail end of it.

In §3.1.1.1 is was shown that the addition of the operations

DOWN : STACK → STACK ∪ {ERROR}
READ : STACK → INTEGER ∪ {ERROR}
RETURN : STACK → STACK ∪ {ERROR}

which perform, respectively, movement down the stack by one position, yield the contents of the stack at the current position, and position to the top of the stack, led to the generation of infinite sets of equations.

To define these operations in Kaviar, the stack abstraction must first be modified to include the position information. This might be done as:

```
type stack
{
        <error> → "error";
        <empty> → "empty" "|" <Nat:pos>;
        <not-empty> → <list:data> "|" <Nat:pos>
}
```

where the vertical bar separates the elements of the stack from the position information.

This change affects the definitions of the basic operations. CREATing an empty stack must also set the position information to zero, as in:

```
create( Bool ) : stack
{
        default → "empty" "|" 0
}
```

A PUSH operation places an element on the top of the stack and must now also increment the position information so that it points to this new top element. The modifications required are:

```
push( s : stack, i : Nat ) : stack
{
        is-error( s ) → s;
        is-empty( s ) → i "|" 1;
        is-not-empty( s ) → i data<s> "|" add( pos<s>, 1 )
}
```

Similarly, POP removes the top element from the stack and so must decrement the position information to point to the new top element. Thus POP must be modified as follows:

```
pop( s : stack ) : stack
{
        is-error( s ) → s;
        is-empty( s ) → "error";
        is-not-empty( s ) → take-tl( data<s> ) "|" subtract( pos<s>, 1 )
}
```

Now that the stack abstraction includes the position information, definition of the operations DOWN, READ and RETURN may proceed.

DOWN could be implemented as:

```
down( s : stack ) : stack
{
        is-error( s ) → s;
        is-empty( s ) → "error";
        is-not-empty( s ) → set-ptr( eq( pos<s>, 1 ), data<s>, pos<s> )
}

set-ptr( b : Bool, l : list, n : Nat ) : stack
{
        is-true( b ) → "error";
        is-false( b ) → l "|" subtract( n, 1 )
}
```

If the pointer is at the bottom of the stack already, moving down results in an error. Otherwise decrement the position information.

The introduction of DOWN, however, requires additional changes to PUSH and POP to take care of the fact that

POP(DOWN(s)) = ERROR, and
PUSH(DOWN(s)) = ERROR

So before PUSHing or POPping it is necessary to check that the position information points to the top of the stack. If it does not, an error must result. The changes needed are:

```
push( s : stack, i : Nat ) : stack
{
        is-error( s ) → s;
        is-empty( s ) → i "|" 1;
        is-not-empty( s ) → test-top( eq( count( data<s>, 0 ), pos<s> ),
                i data<s> "|" add( pos<s>, 1 ) )
}

pop( s : stack ) : stack
{
```

```
        is-error( s ) → s;
        is-empty( s ) → "error";
        is-not-empty( s ) → test-top( eq( count( data<s>, 0 ), pos<s> ),
              take-tl( data<s> ) "|" subtract( pos<s>, 1 ) )
}

test-top( b : Bool, s : stack ) : stack
{
        is-true( b ) → s;
        is-false( b ) → "error"
}
```

RETURNing the pointer to the top of the stack simply requires setting the position information to point to the top of the stack, or equivalently to the number of elements in the stack. A function which implements RETURN might be written as:

```
return( s : stack ) : stack
{
        is-error( s ) → s;
        is-empty( s ) → "error";
        is-not-empty( s ) → data<s> "|" count( data<s>, 0 )
}

count( l : list, n : Nat ) : Nat
{
        is-int( l ) → add( n, 1 );
        is-list-of-int( l ) → count( tl<l>, add( n, 1 ) )
}
```

The function READ could be written in a similar fashion.

Thus using the recursive term algebra approach it is possible to compute invocation expressions such as:

```
{
        down( down( push( create( T ), 4 ) ) );
        return( down( push( push( create( T ), 5 ), 7 ) ) )
}
```

In fact, where in the initial algebra approach infinitely many equations such as

$$(DOWN)^{k+1}(PUSH)^{k}(i_1, \ldots, i_n) = ERROR,$$

for k=0,1,2,... and

$$RETURN(DOWN)^{m}(PUSH)^{n}(i_1, \ldots, i_n) = (PUSH)^{n}(i_1, \ldots, i_n)$$

for all $m \geq 1$, $m < n$ often arise but are not permitted, using the recursive term algebra approach they are computable.

From this example, it is evident the recursive term algebra approach provides a simpler and effective means of defining an algebra.

5.2. Comprehensibility and Minimality

In the initial algebra approach, adding a new operation requires the equational axioms to be reworked so that different terms are forced to denote the same object, forming the equivalence class. Hence the addition of an operation causes the specification to grow reducing both the comprehensibility and minimality of the specification as defined in §2.5.

In contrast, adding a new operation to operate on a term algebra specification does not disturb the specification at all. It may, however, affect some of the other functions defined, as DOWN affected PUSH and POP. This, however, will be true in any specification technique. Because adding new operations does not affect the specification, it remains the same size and stays comprehensible and minimal.

5.3. Range of Applicability

Applications most natural to this specification technique are those involving symbolic manipulation. Examples might include symbolic integration or differentiation, transformation of grammars and so on.

5.4. Extensibility

Suppose a specification had to be changed. If the changes caused new basis values to be added to the recursive definition, this would correspond to the addition more terminal production rules in the grammar. Likewise, if the recursive construction rule was extended the result would be the addition of a new nonterminal production rule in the grammar.

Also, changing an existing specification to carry more information, as the position information was added to the stack abstraction, can be done without much effort.

Thus, extending the specification is a simple task. In the initial algebra approach this involves rewriting the equational axioms which can involve considerable work.

5.5. Possible Research

Given that is has been shown the recursive term algebra approach is a viable method for the specification of data types, Chapter 6 proposes some extensions and an interesting application of the method.

6. Further Research and Conclusions

6. Introduction

This chapter proposes some extensions to the recursive term algebra approach for the specification of data types. Also, an interesting application of this method is presented. Some conclusions then follow these proposals.

6.1. Possible Extensions to Kaviar

It might be interesting to investigate and implement the following ideas.

6.1.1. Parameterized Types

By introducing variables over types, that is, nonterminals, it is possible to define parameterized types. For example, if S is a type variable then:

```
type s-expr( S )
{
        <atom> → S;
        <list> → "(" <s-expr( S ):car> "." <s-expr( S ):cdr> ")"
}
```

defines a type of parameterized symbolic expressions. It might be suitable to define such types using two level grammars.

Definition of parameterized types would allow for the introduction of polymorphic functions. A function such as reversal of an S-expression could now be defined as:

```
reverse( s : s-expr( S ) ) : s-expr( S )
{
        is-atom( s ) → s;
        is-list( s ) → "(" reverse( cdr<s> ) "." reverse( car<s> ) ")"
}
```

making it applicable to all S-expressions not just, say, to S-expressions whose atoms are numeric.

6.1.2. N-ary Unification Functions

The current implementation of Kaviar requires functions defined by unification to be unary only. By introducing additional syntax to accomodate n-ary functions the usefulness of these functions might be increased. Semantic issues become more complicated, however.

6.2. A Possible Application of this Method

Given that data types are equivalent to context free grammars, logic programming provides, in the logic grammar notation, a simple and convenient means of specifying and, because of the procedural interpretation of Horn clause logic, implementing data types.

Logic grammars were first introduced by Colmerauer [Colmerauer,1978]. His *metamorphosis grammar,* strictly more powerful than a context free grammar, is a collection of rewriting rules which can mechanically be translated into Horn clauses, hence, to a logic program. Subsequent work has added several different logic grammar formalisms, and a modification of the *definite clause grammars* [Abramson,1984] may be used to specify context free data types.

A logic program can translate grammatical notation into Horn clauses, generating the selector functions *car* and *cdr,* and the parsers *atom* and *list* for S-expressions. These functions and predicates may be combined by the user to write other functions over S-expressions.

The logic grammar formalism may also be used to define sub-types.

Using these ideas, a typed logic programming language may be imposed over a typeless one to aid the reliability of large scale logic programs. A type free logic programming language ideally suited to this is Prolog.

6.3. Conclusions

The initial algebra approach has been proposed as a method for the specification of data types. It was found, however, that it is often quite difficult to construct specifications using this technique.

This spawned the idea that recursive term algebras should be used instead. The algebras generated by recursive definition were found to be expressible as unambiguous context free grammars.

A functional programming language was then developed around the idea.

Evaluation of the recursive term algebra technique showed it to be superior to the initial algebra technique.

Bibliography

[Abramson,1984].

Abramson, H., "Typing Definite Clause Translation Grammars and the Logical Specification of Data Types are Unambiguous Context Free Grammars," *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 6-9, 1984.

[Aho&Ullman,1972].

Aho, A.V. and Ullman, J.D., *The Theory of Parsing, Translation and Compiling, Volume 1: Parsing*, pp. 320-330, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

[Burstall&Goguen,1982].

Burstall, R.M. and Goguen, J.A., "Algebras, Theories and Freeness: An Introduction for Computer Scientists," *University of Edinburgh, Department of Computer Science, Internal Report CSR-101-82*, February 1982.

[Colmerauer,1978].

Colmerauer, A., "Metamorphosis Grammars," in *Natural Language Communication with Computers*, ed. Bolc, L., Springer-Verlag, 1978.

[Earley,1970].

Earley, J., "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94-102, February 1970.

[Goguen,1977].

Goguen, J.A., "Abstract Errors for Abstract Data Types," *Proceedings of the IFIP Working Conference on the Formal Description of Programming Concepts*, pp. 21.1-21.32, August, 1977.

[Goguen&Tardo,1979].

Goguen, J.A. and Tardo, J.J., "An Introduction to OBJ: A Language for Writing and Testing Algebraic Program Specifications," *Specifications of Reliable Software Conference Proceedings*, Cambridge, Massachusetts, April 1979.

[Goguen,Thatcher&Wagner,1978].

Goguen, J.A., Thatcher, J.W., and Wagner, E.G., "Initial Algebra Approach to Specification, Correctness and Implementation of Abstract Data Types," in *Current Trends in Programming Methodology*, ed. Yeh, R., vol. 4, pp. 80-140, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Goguen,Thatcher,Wagner&Wright,1975].

Goguen, J.A., Thatcher, J.W., Wagner, E.G., and Wright, J.B., "Abstract Data Types as Initial Algebras and the Correctness of Data Representations," *Proceedings of the Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pp. 89-93, sponsored by UCLA Extension in participation with the IEEE Computer Society and in cooperation with the ACM Special Interest Group on Computer Graphics, Los Angeles, California, May 14-16, 1975.

[Graham&Harrison,1976].

Graham, S.L. and Harrison, M.A., "Parsing of General Context-Free Languages," in *Advances in Computers*, ed. Yovits, M.C., vol. 14, pp. 77-139, Academic Press, New York, New York, 1976.

[Guttag,1975].

Guttag, J.V., "The Specification and Application of Programming of Abstract Data Types," Computer Systems Research Group report CSRG-59, University of Toronto, 1975.

[Guttag&Horning,1978].

Guttag, J.V. and Horning, J.J., "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, pp. 27-52, 1978.

[Kanda&Abrahamson,1983].

Kanda, A. and Abrahamson, K., "Data Types as Term Algebras," *University of British Columbia, Department of Computer Science, Technical Report 83-2*, March 1983.

[Klaeren,1980].

Klaeren, H.A., "An Abstract Software Specification Technique based on Structural

Recursion," *ACM SIGPLAN Notice*, vol. 15, no. 3, pp. 28-34, March 1980.

[Liskov&Zilles,1978].

Liskov, B.H. and Zilles, S., "An Introduction to Formal Specifications of Data Abstractions," in *Current Trends in Programming Methodology*, ed. Yeh, R., vol. 1, pp. 1-32, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Majster,1977].

Majster, M.E., "Limits of the Algebraic Specification of Abstract Data Types," *ACM SIGPLAN Notices*, vol. 12, no. 10, pp. 37-42, October 1977.

[Meseguer&Goguen,1983].

Meseguer, J. and Goguen, J.A., "Initiality, Induction and Computability," *Computer Science Laboratory, Computer Science and Technology Division, SRI International, CSL Technical Report 140*, Menlo Park, California, December, 1983.