REPRESENTATION OF FACTUAL INFORMATION BY EQUATIONS AND THEIR EVALUATION

Peter Lucas and Tore Risch *

IBM Research Laboratory, San Jose, California 95193

ABSTRACT

This paper describes a methodology for application software development, the objective being the reduction of volume of code and ease of maintenance. It is shown that constants as well as rules and regulations typically found in business applications should be factored out and stored separately from the application programs in a data base. Definitional equations are proposed as a method for specifying such rules and regulations. The equations can be used as parameters to various types of interpreters to be used by application programs.

As an illustration of the methodology, one such interpreter has been implemented. This paper shows its application to a screen handling program; other uses are discussed. The interpreter and its implementation are outlined.

Key Words and Phrases: Interpreter, Application Development, Knowledge Representation, Applicative Language.

1. INTRODUCTION

There is widespread consensus that progress in programmers' productivity has not kept pace with the rapid technological advances in computer hardware. D.Mc.Cracken [4] estimates a factor of two to three in performance improvement for programmers over the past 25 years; in contrast, the significant characteristics of hardware, such as storage capacity, speed, and price performance, have been improved by orders of magnitude. Many applications that could well be justified on the basis of hardware cost remain uneconomic as a consequence, much to the regret of users and manufacturers alike.

A related phenomenon is the fact that the percentage of programming personnel devoted to the maintenance of installed and operational applications has steadily increased, leaving a decreasing percentage of programmers and other resources for the development of new applications.

The underlying hypothesis of the methodology proposed in this paper is that the volume of code typical for business applications is unnecessarily large. More precisely, it is claimed that the overall application code as it exists in an installation or an enterprise contains considerable useless redundancy. Avoiding this useless redundancy should reduce the volume of code for the benefit of production and maintenance cost.

Redundancy has different sources:

a. repetitive programming of the same function,

b. failure to generalize,

c. encoding of essentially the same information in different forms for different purposes.

The first item represents a management problem to be solved by global management of libraries of programs and specifications together with the necessary administrative procedures to ensure that commonly useful functions are not re-programmed. The above issue is not further addressed in this paper which is solely concerned with items b. and c.

The latter two issues are more subtle; their satisfactory resolution requires a programming style hitherto not widely used for business applications. For illustration of the programming style necessary addressing items b. and c. it is useful to consider a well known example: syntax driven parsers.

A special parser takes a character string as argument and produces a parse tree; the parser's procedural body encodes the underlying grammar. Conceptually, a generalized parser takes two arguments: a syntax and a character string; it produces a parse tree according to the specified syntax. BNF or one of its variants is typically used as the specification language.

The step of generalization leading from special parsers to general parsers is analogous to the generalization proposed for business applications.

The following observations characterize the analogy:

a. Generalized syntax parsers separate specific methods of parsing from the definition of the syntax of specific languages. Having accom-

* This work was done while Tore Risch was visiting scientist at IBM from UPMAIL, Uppsala University, Computer Science Dept., Box 2059, 750 02 Uppsala, Sweden.

plished this separation one may freely combine methods of parsing with any language whose syntax is specified. For example, three implemented parsing methods and four language specifications are in effect equivalent to 12 different special parsers. Collectively these parsers would contain redundancy in the form of repeated specifications of syntax as well as methods of parsing.

b. BNF is a simple equational form of specification which is not biased towards particular uses. The same specification can not only be supplied to generalized parsers but can also be used in conjunction with other algorithms, for example for analysing BNF specifications for interesting properties.

c. Changing a BNF syntax is easier than changing one or more special parsers; the maintenance characteristics have therefore been improved.

d. The ratio of volumes of code between a set of special purpose processing programs and the same functions implemented with generalized programs and separate syntax specifications depends on the number and complexity of the specified languages and the processes to be applied. Dramatic effects on the volume of code can be expected as the number of languages and processing programs increases.

There are many opportunities to carry the analogy into the area of business applications. Examples are: the separation of generalized dialogue patterns from particular dialogue content, e.g. a general fill-in-the-form-and-store-in-the-data-base dialog applicable to some large class of forms, the separation of generalized formatting procedures from specific format specifications for classes of documents, etc.

The questions as to which procedures are in need of generalization and what information is usefully factored are not easily answered. In our opinion this is the central technical question in application development. Program generators, such as IBM's ADF[7] and DMS[8], are contributions in this direction and so are knowledge based systems.

The specific problem addressed here is the factoring of rules and regulations typically found in business applications. Examples are: tax rules (income tax, sales tax, etc.), and company regulations pertaining to overtime compensations and travel expenses.

Definitional equations are proposed as a method for specifying rules. The external specifications and some implementation details of an equation interpreter are presented. The interpreter supports one particular use of definitional equations (i.e. is analogous to a generalized syntax parser in the previous discussion). The interpreter would be useful as part of an interactive program implementing the completion of forms that has been mentioned before. It has been developed as a part of the IBA project, a research project at the IBM Research Center, San Jose, California, focusing on application development methodology.

The interpreter is a variation and extension of the interpreter described in [10].

The example presented in the following section illustrates the factoring and specification of rules. The detailed syntax and semantics of rules are defined in section 3. The interpreter is described in section 4. The last section discusses the relation of the proposed methodology with related published efforts.

2. AN EXAMPLE APPLICATION

2.1 The Display Frame

Consider a dialog fragment starting with a form being displayed by the machine. It is assumed that the form consists of a collection of labelled scalar fields, some for input by the user, some for output by the machine. To be more specific, let ARTICLE, QUANTITY, and STATE be the names of input fields, and let TOTAL-DUE be the name of the only output field of the assumed form. Further, let the format of the blank form be as shown in Fig. 1 below.

```
ARTICLE: ____
QUANTITY: ______
STATE: __

TOTAL-DUE: ______
```

Fig. 1: Blank Form

In practice there would usually be various other components contained by the display frame, e.g. a command field, message fields, a surface area visualizing the current assignments of commands to program function keys, etc. However, as the discussion will focus entirely on the data fields any further detail may safely be suppressed.

2.2 The Dialog

The contemplated dialog consists of the the following steps:

(1) The machine displays a blank form;

(2) the user responds by filling in some or all input fields;

(3) If the user's input is acceptable, i.e. all required inputs are present and syntactically correct, the machine calculates the outputs and displays the result using the appropriate output fields of the displayed form; the dialog fragment under consideration then ends; otherwise, appropriate messages urge the user to fill in missing inputs and/or correct erroneous ones;

(4) the user modifies some or all input fields and the dialog proceeds with step (3);.

Similar to the specification of the display frame, all detail considered unimportant to the further discussion is suppressed, such as the nature of the imbedding dialog that leads into step (1), the continuation of the fragment, and various convenient options one might expect from a well engineered interactive program, e.g. a help function, the possibility to leave the dialog unfinished and quit, etc.

2.3 A Straightforward Implementation

We proceed to sketch a program that implements the specified dialog fragment.

The calculation of the total-due for a given article code, quantity and state of delivery is based on the unit price of the specified article and the rules in effect for computing the sales tax depending on the state of delivery. For the sake of this example it is further assumed that certain articles are exempt from sales tax by federal law; i.e. the exempt status of an article is independent from the state of delivery. As a consequence, the user need not to be prompted for the state in case the article is exempt.

The Business Factors

Business factors describe either characteristics of the environment in which a business operates, such as federal and state laws, or characteristics of the business itself, such as its products and organization 1).

Since business factors are of potential use to many applications they should be kept external to application programs. To avoid duplication business factors should be managed centrally.

Their rate of change, albeit slow, makes it desirable to keep them in easily updatable form.

Unit prices of articles, their exempt status, and the tax rate for each state are business factors pertaining to the present example. The information on unit prices and the exempt status of the various articles may be viewed as functions (f1, f2) from valid article codes to dollar amounts and truth values respectively. The sales tax is given by a function (f3) from states to percentage figures. All three functions are assumed to be represented by tables as indicated in Fig. 2.

f1:

article	unit-price
A001	23.50
A002	180.05
...	...

f2:

article	exempt
A001	yes
A002	no
...	...

f3:

state	tax
CA	6.00
...	...

Fig. 2: Business Factors

The Dialog Procedure

The following is only a sketch of a procedure implementing the example dialog.

```
 proc DIALOG;
   ...
   dcl article, quantity, state, total-due,
       total, taxrate, tax ... ;
   ...
   display blank form;
   wait for response;
   ...
   until all required inputs are specified
     prompt for missing input and error correction;
     ...
L: taxrate := if f2(article) then 0 else
                                  f3(state)/100;
   total := f1(article)*quantity;
   tax := total*taxrate;
   total-due := total+tax;
   display form;
   wait for response;
 ...
```

Fig. 3: Dialog Procedure

The content of the program variables: article, quantity, state, and total-due, reflect the content of the correspondingly labelled fields in the display frame, converted, if necessary to the internal representation required by the operations to be applied to the variables.

At program point 'L:' it is assumed that all required input variables have suitable values, such that the computation can be completed as programmed without further intervention from the user; this implies that the code fragment indicated by 'prompt for missing input and error correction' has to check that the user inputs are valid; more precisely, that the article code occurs in table f1 and that the state is listed in table f3 etc. Furthermore, if the user is not to be prompted unnecessarily for the state, this code fragment must check the exempt status of the article when specified by the user.

2.4 Implementation using Generalized Procedures

Two observations guide this second design. First, the program encodes the rules in effect for computing the sales tax, a piece of information presumably useful in context other than this particular program. It would therefore appear desirable to represent this information separately. Second, the general dialog pattern seems useful for collections of data elements other than the specific sales related data elements of the example. All references to the specifics of the data elements must be parametrized to achieve this generality. The consequences of both observations will be elaborated in turn.

1) The concept and term is due to Dennis Burk (Motorola)

Some of the tax related information has already been separated in the form of business factors. These business factors can be used by multiple applications and changed without having to adapt the application code.

However, this separation has not gone far enough. Changes in the computational rules for the sales tax, rather than the percentages or exempt status, induce modifications to the application code; if these computational rules are used by several applications, all copies of the rules have to be found and changed (redundancy). It would therefore be practical to separate the rules together with the business factors.

Next, a method of representation for the tax rules has to be found. The obvious factoring of the tax rules in form of a subroutine, upon closer examination, is inappropriate; the reason is analogous to that cited in the introduction in support of BNF for syntax specifications, i.e. a representation of the essence of the information without bias towards a particular use. Definitional equations, discussed next, are the proposed solution.

Definitional Equations

The functional dependencies among the quantities involved can be conveniently defined by the set of definitional equations below.

```
(1) exempt = f2(article)
(2) taxrate = if exempt then 0 else f3(state)/100
(3) total = f1(article)*quantity
(4) tax = total*taxrate
(5) total-due = total+tax
```

Fig. 4: Definitional Equations for Sales Tax

With f1, f2 and f3 bound to specific tables (see Fig. 2), the collection of equations defines a function from value assignments to the input variables: article, state, and total, to the output variables: exempt, taxrate, tax, and total-due. The collection of equations is not a program, i.e. an executable unit defining a computation on a real or hypothetical machine, but is merely a statement of dependencies.

The equations are used as parameters to interpreters of one kind or another, which may define any desirable dynamic behavior. One example is an interpreter supporting the example dialog to be discussed later.

The remaining problem to be discussed is the generalization of the dialog. As will be argued, the dialog and equation interpreter are best implemented as separate tasks. The ADA rendezvous mechanism [1] will be used to present the example; the mechanism is briefly reviewed below.

ADA-like Process Communication Discipline

The mechanism is only sketched as far as necessary for the purpose of the present example.

The linguistic representation of an independent process is called 'task'.

A task consists of a specification, defining the external interface, and a body. The specification of a task defines its names, parameters, and entry points, each with its own parameter list.

Below are specifications of two tasks P, and Q.

```
task P(x);                task Q(x);
   entry P1(a);              entry Q1(a);
   entry P2(b);
```

Fig. 5: Task Specifications

The outline of bodies for tasks P and Q in Fig. 6 below show the constructs used for initiation and synchronization of tasks.

```
body of P:                body of Q:
 ...                       ...
initiate Q(A1);           P1(A3);
 ...                           ...
accept P1(x) .. end;      accept  Q1(a) ...  end;
 ...                       ...
Q1(A2);
 ...
select
      accept P1(a) ... end;
or
      accept P2(b) ... end;
 ...
```

Fig. 6: Task Initiation and Communication Constructs

Assuming that task P is initiated and running, task Q is started with the initiate statement passing the argument A1. If task Q reaches the call of the entry P1 before the entry is activated through an accept P1 statement, task Q waits for this event to occur.

If, on the other hand, task P reaches an accept P1 statement before a corresponding call is issued, it will wait until this event occurs.

In case the entry has been activated by an accept statement and a call has been issued, the called task proceeds by executing the clause associated with the accept statement, delimited by a matching end. This sequence of events is called a 'rendezvous'.

As soon as the clause is executed, the calling task is released and can proceed independently.

The select construct activates several entries simultaneously, e.g. the select construct contained in the body of P activates the entries P1 and P2.

Generalization of the Dialog

In the straightforward implementation of the example dialog, the prompting for missing input is complicated by the fact that the state of purchase need not be specified in case the article is exempt. Thus, in general, the prompting mechanism needs to take into account dependencies between variables. One way to accomplish the effect in the general case is to start the computation with whatever input the user provides initially and prompt for further information as the need arises. This evaluation mechanism is contained in a task called 'compute' whose specification is detailed below.

```
task compute(equations);
   entry update(in);
   entry request(out);
   entry cancel;
   entry retvalue(val);


in: ID->VAL     mappings from identifiers to values
out: ID-> null  mappings from identifiers to null
val: VAL        one value
```

Fig. 7: General Compute Task

The interface provides a late binding for the collection of equations to be used; The collection of equations is passed as argument when the compute task is initiated. Which specific form the argument may take depends on dictionary and data base functions for storing and retrieving sets of equations.

After the compute task is initialized, the update(in) entry is used to pass a value assignment to some or all input variables of the equations previously bound. The entry request(out) is used to request computation of some or all output variables of the equations.

As indicated, value assignments to variables are represented as maps from identifiers to values. The set of requested output variables is represented as a value assignment: a map from the identifiers to null values.

The compute task assumes that the task requesting its service provides two entries: an entry called 'prompt(var)' to be used when further input is needed and an entry 'result(out)' to be used for delivery of the result.

The entry prompt(var) is called when the compute task needs the value of some input variable, var, which was not initialized by update(in). The value, val, of the prompted variable can be returned to the compute task with the entry retvalue(val) continuing the interpretation where it was interrupted.

The entry cancel permits the cancelation of a pending request, and resets all variables so far defined or computed to null.

Finally the example shall be completed with a sketch of the generalized dialog program. The external interface as far as relevant to the discussion is shown in Fig. 8 below.

```
task dialog(form, equations)
 ...
   entry prompt(var);
   entry result(out);
    ...

var: ID        one identifier
out: ID->VAL   mappings from identifiers to values
```

Fig. 8: Specification of the Dialog Task

The argument 'form' provides all information about the data elements involved in the dialog necessary to display the corresponding fields, check for syntactic correctness of the inputs, and conversions from external to internal representation of values. The equations define the dependencies among those elements.

The two arguments are therefore not independent and should be derived from a consistent database representations of the information.

The two entries specified are those expected by the compute task.

Fig. 9 shows the internal structure of the dialog task.

```
task dialog(form, equations)
   display blank form and obtain initial input;
   initiate compute(equations);
   update(in);
   request(out);
    ...
   while request uncompleted
     select
       accept prompt(var); var1 := var end;
         prompt user for a value assignment to the
         variable named var.  if successful return
         with    retvalue(val),  otherwise   cancel
         (e.g.).
       or
       accept result(out); out1 := out end;
         request completed;
The task may continue with further requests for
additional output variables to be computed, or can-
cel (reset) the compute task and start again with a
blank form;
 ...

The types of referenced variables are:
      in: ID -> VAL
      out, out1: ID -> VAL
      var, var1: ID
      val: VAL
```

Fig. 9: Outline of Internal Dialog Structure

The dialog task starts by displaying a blank form with the input and output fields available to the user, these correspond, of course to some input and output variables of the equations on which the dialog is to be based.

The dialog obtains some initial set of inputs from the user, initiates the compute task, passes the available inputs to compute using the update entry, and requests that the output variables of the form be computed, using the request entry.

The dialog task then enables the prompt and result entries, thus waiting for the compute task to either come back with the result, or prompt for further input.

In both cases, the critical section is used to copy the argument passed by the compute task to some program local variable.

The generalized program now separates the definition of the data elements and their functional dependencies, a particular evaluation algorithm, and the code accomplishing the communication with the end user.

2.5 Discussion

The generalized version of the example application, lately discussed, was decomposed into three major parts:

a. the definition of a set of variables and their functional dependencies, pertaining to sales tax regulations, and represented in the form of definitional equations,

b. an evaluation algorithm capable of computing the values of some arbitrary subset of dependent variables, prompting for needed inputs,

c. a dialog task which takes responsibility to communicate with the end user, calling the evaluation algorithm when appropriate.

Any one of these three parts is potentially useful in more than one context. The equations of part a. can be used in any application involving the particular encoded rules, but not necessarily with the specific evaluation algorithm of part b.

On the other hand, the evaluation algorithm can be applied to any set of equations as long as these conform to the syntactic and semantic restrictions specified in more detail in the next section.

Finally, part c. embodies the detailed flow of a user-machine interaction but is independent of the subject of the dialog, i.e. the number and kind of variables involved. The dialog part has received a sketchy treatment but would deserve a more detailed study including the specifications of display formats, type information to be associated with variables, etc.

However, here the latter subject is not further pursued. The following sections detail the general form and meaning of equations exemplified in part a. and the associated evaluation algorithm of part b.

3. SYNTAX AND SEMANTICS OF RULES

The language defined below is somewhat richer than was indicated by the examples of the previous section. In particular auxiliary functions may be defined as part of a package; large expressions may be avoided using variables local to an expression.

No particular value domain or type system is introduced at this point. It is, however assumed that the boolean values are contained in the domain.

3.1 Names

```
<fnsymb>            ::= <identifier>
<variable>          ::= <identifier>
```

Names are drawn from the syntactic category of <identifiers>. Although identifiers do not have any inherent meaning, two classes of names are distinguishable by context: function names and variable names.

3.2 Expressions

```
<expression> ::= <constant> |
                 <variable> |
                 <term> |
                 <select expression> |
                 <where-expression>
<term>   ::= <fnsymb>(<argument list>) |
             <expression><fnsymb><expression>
<argument list> ::=
             <expression> |
             <expression> , <argument list> |
             <empty>
<select expression>::= select; <select body> end
<select body>::= <select clause> ; |
                 <select clause> ; <select body> |
                 otherwise <expression> ;
<select clause> ::=
             when(<expression>)<expression>
<where-expression> ::=
             <expression> where <declarations>
             <declarations> end
<declarations> ::=
             <variable definition> ; |
             <variable definition> ; <declarations>
<variable definition>::= <variable> = <expression>
```

Each expression contains a set of free variable names and a set of free function names; these sets are determined by the rules specified below. Given an interpretation for these free names, i.e. a value assignment for the variable names and an assignment of partial functions to the function names, the expression denotes a value or undefined.

Since the value domain has not been elaborated at this point, constants are left undefined.

Terms are constructed from function names and argument lists. The interpretation of the function name is restricted to functions whose number of formal parameters agrees with the number of arguments. Infix-notation may be used for binary function names. The set of free names is the union of the free names of the argument expressions and the function name of the construction.

Select expressions are used as a generalized form of if-then-else expressions. Given an interpretation, select expressions are evaluated by examining each select clause in turn. The expression following the keyword when must denote a truth value. The set of free names of a select expression is the union of the free names of the component expressions.

The where-expression permits the definition of local variable names. All variable names defined in the declaration part of the construct are bound, their scope is the where-expression. The set of free names is the set of free names of the expression components minus the bound variable names defined in the declaration part.

3.3 Rules

```
<rule package>    ::= /* a set of rules */
<rule>            ::= <variable definition> ; |
                      <function definition> ;
<variable definition> ::= <variable> = <expression>
<function definition> ::= <fnsymb>(<parms>)=
                          <expression>
<parms>       ::= <variable> | <variable>,<parms>
```

Function definitions have the usual meaning. Parameters are variable names bound in the scope of the function definition, i.e. the free names of the construction are the free names of the expression minus the names of the parameters. The scope of the function name defined by the construct is the rule package in which the construct is embedded.

In contrast, variables names defined by a variable definition are considered free in the context of a rule package. The set of free names of a variable definition are the free names of the expression and the variable name being defined.

A rule package is an unordered set of function definitions and variable definitions. The set of free names of a rule package is the union of the free names of variable definitions and the union of the free names occurring in function definitions but excluding the function names being defined.

Some important constraints on rule packages need to be mentioned.

a. The names defined by any two rules of a given package must be distinct.

b. Let R' be a relation defined among the free variable names of a rule package as follows: a R' b holds if there is a rule defining a and if b is a free variable of the right hand side of the definition. Let R be the transitive closure of R'. The constraint is that R must be antisymmetric, i.e. variable definitions must not be circular.

Note that this constraint need not hold for function definitions which may be recursive.

One may take two quite different views of the semantics of a rule package. We assume that all free function names are bound to suitable specific functions. A rule package together with this binding can then be viewed as an open sentence: given a value assignment to the free variables, the rule package denotes a truth value, i.e. the value assignment may or may not satisfy the equations. Due to the partiality of the functions there is of course also the possibility that neither is the case.

Due to the restricted forms of the equations, with a single variable on the left hand side and no two equations defining the same variable one may view the semantics of rule packages differently. The variables occurring on the left hand side of the equations can be viewed as dependent variables defined in terms of the other free variables, the independent variables.

More precisely, let V be the set of free variables of a given rule package. This set of free variables can be partitioned into two sets, the set of variables occurring on the left hand side of variable definitions called the set of output variables O, and the set of all other variables called input variables I. Given interpretations for the free function names, a rule package denotes a function from value assignments of the input variables I to value assignments of the output variables O.

More formally, the meaning of a rule package p is a function M(p) of the following kind:

$$M(p): (F \to Funct) \to ((I \to Val) \to (O \to Val))$$

where: F ... set of free function names in p
Funct ... set of functions on Val
I ... input variables of p
Val ... value domain
O ... output variables of p

The interpreter described in the following section is based on the latter semantic view of rule packages.

4. A Rule Interpreter

4.1 Problem Statement

The rule interpreter implemented in the context of the IBA project, and described in this section supports but one particular use of rule packages. A complete description of the interpreter is found in [12]. Given a rule package, the interpreter computes the values of a given subset of output variables based on a given value assignment to a subset of the input variables. The interpreter prompts for further input as needed to satisfy the specified request.

4.2 External Specifications

Value Domain

The interpreter handles data values of several different data types. For example, a variable need not be bound only to scalars, but it can be bound to, e.g., arrays and records as well. The interpreter supports an abstract data type system, where a few basic data types (e.g. integers, real numbers, and records) are built-in and where the programmer can define new data types in terms of other data types. All data types, basic as well as user defined, are treated in a uniform way by the interpreter. The type system is stored in a System R [2] database.

The rule interpreter internally uses four data types, named CHARS (text strings), INTEGER, REAL, and BOOL (logical values).

Literals. A literal in the rule language is regarded as a typed constant. Therefore every literal has an associated data type, indicated by the format of the literal, e.g., 123 is an integer, 1.23 is a real number, and 'Ape' is a string. The literals are encoded into type tagged data values when a rule package is accessed.

Literals may be defined for data values of any data type, provided that the programmer has specified a conversion routine from text strings to data values of the data type, as will be described later. Once that conversion routine is defined, literals of the data type may be written as a qualified literal:

    '<text string>'<type>

where <text string> is the string representation of a data value of type <type>. For example, the literal '1'INTEGER is equivalent to 1.

Overloaded functions. There is a mechanism in the system for calling external PL/I subroutines as functions from rules, by using overloaded functions An overloaded function accepts actual parameters of several different data types, and it has several different definitions depending on the data types of its actual arguments. Each overloaded function is associated with one or several specific subroutines Each specific subroutine implements the overloaded function for a particular number of arguments and particular types of arguments. The specific subroutines are defined as separately compiled PL/I subroutines with standardized formal parameters. When the interpreter encounters a call to an overloaded function a generic selection mechanism is invoked to get the name of the actual specific subroutine. When the specific subroutine is found the subroutine will be dynamically loaded and executed by the interpreter.

When an overloaded function is defined, the programmer must specify the data type combinations which are allowed for its actual parameters. The name of a specific subroutine must always be associated with each combination of data types.

The conversion routines mentioned for qualifyed literals are implemented as overloaded functions.

See [12] for a complete description of how overloaded functions are handled by the system.

System Environment

The interpreter can access a number of rule packages stored in a data base. In the present implementation the rule packages are stored in flat files as sequences of variable and function definitions in source form.

The union of two rule packages, P1 U P2, is defined by adding the variable and local function definitions of P2 to those of P1. If a variable or local function is defined in both P1 and P2, only the one defined in P2 will be defined in P1 U P2.

A global rule package, INIT, is available. When the interpreter is initialized for a rule package, P, the interpreter will perform INIT U P, and the interpretation will be done with the package INIT U P.

The global rule package contains some variable and function definitions which are commonly used, namely
- The variables TRUE and FALSE for boolean values.

- The logical functions AND, OR, and NOT are defined with their normal meanings.

- select expressions are internally represented as function calls to a system function, *SELECT.

Internal State

The internal state of the interpreter is defined by the following data:
1. The interpreter must have access to a rule package or a union of several rule packages which are used for the interpretation.

2. The interpreter must also be able to describe bindings between variable names and their values.

3. Finally, the bindings between free function names and their definitions must be known.

Multiple activations of the interpreter may exist simultaneously. With each new activation a new internal state is created and identified with an activation identifier. The internal state can be successively updated by successive calls to rule interpreter entries referring to the same activation identifier.

Operations

The interpreter can be called from a PL/I program through some external entry points. A number of PL/I macros are defined for simplifying the interface. The interface is somewhat modified compared to the rendezvous interface of section 2.6 so that it can be implemented with PL/I macros and a coroutine extension to PL/I.

4.3 The Evaluation Algorithm

We begin this section with an informal description of the evaluation algorithm, and its motivation.

The interpreter tries to calculate the output variables using input variables, and the the functions and variables defined in a rule package. Eventually the interpreter may find that some variables are undefined. When the interpreter finds an undefined variable, it will call the entry PROMPT in the routine which called the interpreter. This entry may either prompt the user for the value or get the value from some other source, e.g. the screen.

Unnecessary prompting should be avoided during the evaluation. Therefore the interpreter tries to minimize the number of prompt variables by minimizing the number of expressions evaluated. The number of expressions that need to be evaluated to solve a problem using our method is greater than or equal to the minimal number of expressions.

The minimization is achieved in our implementation in two ways:

- Once a variable is prompted during an evaluation, it will not be prompted again. In general, once the definition of any variable has been evaluated, the definition will be replaced with that value. The interpreter has a feature for unbinding variables when, e.g., an interpretation is cancelled.

- The commonly used methods to pass arguments to functions will cause too much evaluation. 'Call by value' will evaluate the arguments before the function definition is evaluated whether or not the value of every argument is needed for the calculation. 'Call by name' may possibly evaluate the arguments several times. Therefore a method, 'call by need', is used in which the arguments are evaluated only when needed and only once.

An example of a case where 'call by need' is useful is when defining the function sales_tax as in figure 11. If f2(article) is true then the value of the function is independent on the parameter 'state'. Our call by need mechanism ensures that the parameter 'state' is evaluated only when f2(article) is false.

```
sales_tax(article,state,total)=
    total*select;
          when(f2(article)) 0;
          otherwise f3(state)/100;
        end;
```

Fig. 11: Illustration to call by need

Internal representation

When the rule interpreter is initialized, it will access the specified rule package and sequentially read the variable and function definitions in the package. After a variable or function definition is read, a parser is called which transforms a statement in the external rule language into an abstract syntax tree. One rule package will thus result in several abstract syntax trees, which are stored in the work space of the interpreter. The abstract syntax trees are finally interpreted.

The abstract syntax trees are represented as strictly binary trees, using a representation similar to Lisp's S-expressions.

Figure 12 shows examples of S-expressions representing a variable and a function definition respectively. We use the notation <e1 e2 etc..> to denote a list with the elements e1,e2,etc.. The examples should illustrate clearly the rules for the transformation from the external to the abstract syntax. Conventional LAMBDA expressions are used for representing function definitions. Variables are represented similarly, except that their definition is tagged VLAMBDA instead of LAMBDA.

The definition of

```
taxrate = select;
            when(exempt) 0;
            otherwise f3(state)/100;
          end;
```

is internally represented as the structure

```
<VLAMBDA
    <SELECT
       <EXEMPT 0>
       <TRUE </ <F3 STATE> 100> >> >
```

The definition of

```
dif(x,y)=x-y+1;
```

is internally represented as

```
<LAMBDA <X Y>
        <- X <+ Y 1>> >
```

Fig. 12: the internal representation of an equation and a function

The abstract syntax trees are represented using three data structures:

1. List cells are used for representing nodes of binary trees. List cells have the two fields HEAD and TAIL which contain pointers to any of the other two data structures.

2. Symbols (atoms) have unique representations by using hash technique. Each symbol have one field, the definition cell (DEFCELL), which is used when a variable or a function definition is associated with the symbol. When the interpreter parses a variable or a function definition it will store a LAMBDA or a VLAMBDA expression in the definition cell.

3. Finally the abstract syntax trees may contain references to data values. The parser will transform literals into such references. In the definition of the variable 'taxrate' in figure 12, '0', and '100' are examples of data values.

The definition cell

The definition cell of a symbol is used for associating rule definitions with the symbol. The definition cell contains a list structure describing the definition. If the symbol is defined as a function or a variable it will contain LAMBDA or VLAMBDA expressions respectively. The HEAD of the list structure in the function cell (LAMBDA or VLAMBDA) is a tag, indicating the type of the definition of the symbol. The tag informs the interpreter how to evaluate a reference to the symbol.

The function cell is also used for associating a data value with a variable. If the function cell of a variable contains a pair, (VALUE . <obj>), it means that the variable is assigned to the data value <obj>, which is a pointer to a data value.

There are also other tags allowed in the definition cell, used for defining internal functions and variables.

5. DISCUSSION

Lazy evaluation is an evaluation and optimization method introduced by [6]. The principle of that method is that calls to the constructor of LISP (CONS) does not create new objects, but rather it returns descriptions of what to evaluate in order to get the values of each field of the objects. The selectors (CAR,CDR) know how to use these descriptions for getting the field values. Thus all evaluation is performed by the selectors.

A similar technique is used by [5] for APL objects. By using this technique they avoid constructions of intermediate objects (e.g. arrays) during the execution of an APL program.

In contrast to [6] and [5], our evaluation optimization method tries not to avoid evaluations when constructing objects, but rather it optimizes evaluations of function calls. We do so by evaluating variables only once, and by using the 'call by need' mechanism when calling parametrized functions. A similar method was proposed by [15].

The interpreter is related to Lisp in that we use the Lisp's S-expressions for internal representation of expressions, and a similar classification of function types as in Interlisp [14]. However, our evaluation algorithm is different. For example, Lisp does not use 'call by need' or global variable substitution.

Other techniques which can be used for interpreting definitional equations include production systems [11], logic programming [9], and programming with constraints [3,13]. These types of techniques make it possible in some cases to use the equations in different directions, i.e. a given set of equations does not restrict the possible input, and output variables respectively.

REFERENCES

[1] 'The Reference Manual for the Ada Programming Language', Superintendent of Documents, U.S.. Government Printing Office, Washington, D.C. 20402, Order No. L008-000-00354-8, 1980.

[2] M.M.Astrahan, et. al.: 'System R: Relational Approach to Database Management', ACM Transactions on Database Systems, June 1976.

[3] A.Borning: 'The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory', ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, Oct 1981.

[4] D.McCracken: 'The Changing Face of Application Programming', Datamation, Nov. 15, 1978.

[5] L.J.Guibas, D.K.Wyott: 'Compilation and Delayed Evaluation in APL', Proc. 5th ACM Symposium on Principles of Programming Languages, 1978.

[6] P.Henderson, J.Morris: 'A Lazy Evaluator', Proc. 3rd ACM Symposium on the Principles of Programming Languages, 1976.

[7] IBM Corp.: 'IMS Application Development Facility Program Description/Operations Manual', SH20-1931-1, 1977.

[8] IBM Corp.: 'Development Management System/ Distributed Processing Programming Executive: Program Reference and Operations Manual', SH20-2420-0, 1980.

[9] R.Kowalski: 'Algorithm = Logic + Control', CACM, Vol. 22, No. 7, July 1979.

[10] P.Lucas: 'On the Structure of Application Programs', in: D.Bjorner (ed.): 'Abstract Software Specifications', Lecture Notes in Computer Science No. 86, Springer-Verlag, Berlin-Heidelberg-New York, 1979.

[11] N.J.Nilsson: 'Principles of Artificial Intelligence', Tioga Publishing Co., Palo Alto, California, 1980.

[12] T.Risch: 'An Interpreter for Functional Rules', IBM Research Report RJ3360, 1982.

[13] G.L.Steele Jr., G.J.Sussman: 'Constraints.', APL '79: Conf. Proc., APL Quote Quad (ACM SIGPLAN/STAPL) Vol.9, No. 4, June 1979.

[14] W.Teitelman et. al.: 'Interlisp Reference Manual', Xerox Palo Alto Res. Cen., 1978.

[15] J.Vuillemin: 'Correct and Optimal Implementations of Recursion in a Simple Programming Language', Journal of Computer and System Sciences, vol. 9, No. 3, Dec 1974.