Variational Attribute Grammars for Computer Aided Design (Release 3.0)

David McAllester
ADAGE-MIT-94-02

Abstract: This document describes a variational attribute grammar (VAG) design language and release 3.0 of the VAG implementation. VAG is a functional programming language specifically designed for the rapid implementation of domain specific CAD systems. VAG is intended to support the implementations of CAD systems in such diverse areas as mechanical, electrical, and software design. The main feature of the VAG language is a constraint based reasoning mechanisms to support the CAD user in analyzing partial designs in a wide variety of domains. Release 3.0 provides a justification mechanism — no such mechanisms were provided in earlier releases.

1 Introduction

This document, ADAGE-MIT-94-02, is largely identical to the earlier document MIT-ADAGE-94-01. The differences involve the justification mechasisms provided in release 3.0 of the software. These mechanisms are not provided in eariler releases. The new mechanisms are described in section 4.

VAG is a functional programming language designed for the development of domain specific constraint based CAD systems. There are two kinds of users of the VAG system — the VAG programmer and the CAD system user. The VAG programmer defines a set of VAG programs which are then used as design primitives in a CAD system. CAD systems built in this way will be called VAGCAD systems. This document is intended to both describe the VAG language at a somewhat theoretical level and to provide a user's manual for a certain implementation of this language. This document is intended for CAD systems researchers, VAG programmers, and to lesser extent, VAGCAD system users.

Although the VAG language can support the design of CAD systems in a variety of domains, the primary emphasis is on CAD systems for software. The original motivation for the VAG language involved component based design of avionics software. The construction of a CAD system for software presents a variety of challenges. Component based software design is traditionally organized around procedure and object libraries. The VAG language is designed to support components which are in some sense more general than traditional or object oriented procedures. In particular we are interested in parameterized components where the parameters of the components are determined at system design time and tailoring a component to particular parameter values is nontrivial. For example, in the avionics domain a natural component is a high pass filter. Natural parameters of a high pass filter are the frequency at which the filter starts to pass the signal and the "sharpness" of the step in the frequency response. Different values of the sharpness parameter can lead to quite different filter designs so it is important that the parameters be specified, and the filters be constructed, at system design time rather than software execution time. In general, we view a component as a design time procedure that takes a set of parameters and produces a part of a design. Since they are called at design time rather than system run time, the procedures for mapping parameter values to design fragments do not have to be particularly efficient. Furthermore, the constraint based methodology of VAG does not involve calling these design time procedures until the design is completed. The constraints involved in partially automating the design process can be used without actually invoking the procedures that convert parameter values to design fragments.

The term "VAG" is an acronym for Variational Attribute Grammar. Traditional attribute grammars play an important role in software development tools such as compilers and editors. A computer program is an expression satisfying the syntactic requirements embodied in a grammar which describes the language in which the program is written. Traditional attribute grammars augment traditional context free grammars with equations that define "attributes" of programs and program fragments [Reps and Teitelbaum, 1985], [Deransart *et al.*, 1988]. A set of VAG programs can be viewed as an attribute grammar and a VAG programmer can be viewed as a grammar designer. Actually, it is the type information associated with defined functions that plays the role of a grammar. Type information specifies a set of well typed expressions in much the same way that a traditional grammar specifies a set of grammatical expressions. The relationship between types and grammars has been widely studied in the context of natural language syntax [Adjuciewicz, 1935], [Bach, 1988]. A design can be viewed as an attributed grammatical (or well typed) expression. However, the analogy with attribute grammars hides the fact that the VAG language is also a general purpose functional programming language. Both views of VAG — the grammar view and the functional programming view — are important and useful. The functional programming view is perhaps the more natural of the two and is the primary view taken in section 2.

Traditional attribute grammars make a distinction between inherited attributes whose values are derived from context and synthesized attributes whose values are computed from subexpressions. In a VAG expression attributes are related by constraints which allow multidirectional flow of information. Variational attribute grammars are similar to, but formally simpler than, relational attribute grammars as described in [Courcelle and Deransart, 1988]. Although the value of a VAG attribute can be inferred using multidirectional constraint flow, at a semantic level the VAG language has only synthesized attributes. An attribute `foo` of an object x is represented by the term (`foo` x) which is semantically no different from any other term in a functional programming language. The value of the term (`foo` x) is derived by applying the function `foo` to the object x. However, if x is a variable, or contains free variables, then the value of `(foo` x`)` can not be determined in this way. It might be determined, however, by global constraints on the design from which one can infer that the term `(foo` x`)` must have value 3. In this way multidirectional constraints can be used to derive the value of the attribute `(foo` x`)` of the object x.

Although VAG is formally a functional programming language, VAG programming is quite different from programming in traditional functional languages. VAG programs are intended to be used as parts of partial designs. The VAG interpreter is designed to operate on partial designs, i.e., expressions in which many subexpressions are not yet specified. In a VAGCAD system a partial design is a VAG expression in which some parts are not given. These yet to be designed parts are represented by variables. In order to create useful VAG primitives the VAG programmer must understand the constraint based reasoning mechanisms that will be available to the VAGCAD system user. The VAG functions created by the VAG programmer should be such that the VAGCAD reasoning mechanisms can reason about them effectively.

The use of variables to represent yet to be designed components is formally similar to the use of logic variables in constraint logic programming. The main difference is the intended degree of interaction between the system and a human designer. In a VAGCAD system the logic variables range over yet to be designed aspects of a system. The VAGCAD system performs as much constraint based reasoning as is feasible but the system relies on a human designer to make major decisions in filling in a design.

Section 2 specifies the syntax and semantics of the VAG language. Section 3 describes the constraint based inference mechanisms that are used to support the analysis of partial designs. Instructions for running the current implementation of the VAG language are given in section 4.

2 The VAG Language

The VAG language derives much of its syntax (or lack thereof) from Lisp. Function applications are represented with parenthesis. For example the application of F to X is represented by `(`F X`)`. VAG is a purely functional language — function application is the only method of building large expressions and programs. A VAG program is itself simply a function which can be applied to build other expressions. The VAG language is typed. The well-typed expressions correspond to the "grammatical" expressions under a classical attribute grammar. In addition to types, the VAG language associates each function symbol with constraints. The constraints associated with a function give conditions, in addition to the typing constraints, which must be satisfied in order for an application of that function to be well formed. These constraints provide the foundation for run time constraint processing in a VAGCAD system. Most of this section concerns aspects of the language other than constraints. The mechanisms for associating functions (and types) with constraints are described in section 2.7. Constraint based reasoning is described in section 3.

The VAG language is implemented as a macro package in Common Lisp. Many of the features of the language, such as list processing primitives and numerical primitives, are inherited from Common Lisp [Steele, 1990]. The following subsections describe both the primitives of the VAG language and some of their intended uses. As noted in the introduction the intended use of VAG programs almost always involves constraint based reasoning about partially specified arguments.

2.1 Types and Data Structures

The VAG language is typed and the system prevents either the VAG programmer or the VAGCAD system user from constructing ill-typed expressions. A type is either a type symbol or a type of the form `(list-of` s`)` where s is a type. For example, `number`, `(list-of number)`, and `(list-of (list-of number))` are all types. There are six primitive symbolic types — boolean, fixnum, float, number, symbol and anything. In any given domain there will also be many other type symbols defined by the VAG programmer. The VAG language does not support function types. In fact higher order procedures are not first class objects in VAG. Procedural arguments are only allowed as the first argument to the special procedure `map`. The primitive `map` is described in more detail below.

The VAG programmer can introduce new types with `defstruct` declarations. These declarations have syntax and semantics similar to the corresponding declarations in Common Lisp. As a simple example of a `defstruct` declaration we consider the type "drawer" which might be defined by a VAG programmer as a type in a VAGCAD system for furniture design.

```
(defstruct drawer
  (drawer-height number)
  (drawer-width number)
  (drawer-depth number))
```

The above declaration introduces `drawer` as a type symbol. It also introduces four functions — `make-drawer`, `drawer-height`, `drawer-width` and `drawer-depth`. The function `make-drawer` takes three arguments — one for each field of the drawer data structure. For example, the value of `(make-drawer 10 15 25)` is a drawer data structure with a height of 10 a width of 15 and depth of 25. The functions `drawer-height`, `drawer-width` and `drawer-depth` each take a drawer data structure as an argument and return a number representing the height, width, or depth of that drawer respectively.

In the above declaration each field of the data structure is associated with a type. In this case all three fields are associated with the type `number`. All VAG functions, other than the polymorphic primitives described in section 2.2, must be associated with input types and output types. In addition to introducing the type `drawer`, the above declaration associates each of the four newly introduced functions with input and output types. We write $f : \sigma_1 \times \sigma_2 \times \cdots \times \sigma_n \rightarrow \tau$ to indicate that f is a function whose inputs must be of type σ_1, σ_2, …, σ_n respectively and whose output is of type τ. The above declaration introduces the following type declarations.

```
make-drawer:  number × number × number → drawer
drawer-height:  drawer → number
drawer-width:  drawer → number
drawer-depth:  drawer → number
```

Note that if we think of the type `drawer` as a nonterminal of a grammar then `(make-drawer 10 15 25)` is a well formed expression "generated by" this nonterminal. As we shall see below, the VAG programmer can define other functions with output type `drawer` and in so doing enrich the class of expressions of type `drawer` available to the VAGCAD system user. The functions `drawer-height`, `drawer-width`, and `drawer-depth` define attributes of drawers. In any completed expression of type `drawer` these attributes will have well defined values. In a partial design the VAG system may infer the value of these attributes from global constraints on the design.

The general form of `defstruct` declarations include features described in sections 2.4 and 2.7. The general form of a `defstruct` declaration is given in section 2.7.

2.2 The Primitives of VAG

The VAG language includes a variety of "polymorphic" primitives. A function is polymorphic if it can be used with different types at different times. For example the primitive `cons` can be applied to two arguments where the first is of type σ and the second is of type (`list-of` σ) and when applied to these arguments it returns an object of type (`list-of` σ). The polymorphic primitives of VAG use a form of polymorphism known as *parametric polymorphism* [Milner, 1978], [Cardelli and Wegner, 1985], [Milner *et al.*, 1990]. This means that the type declaration for the primitives can include *type variables* which can be filled in by particular types at the time the primitive is used. For example, we have the following type declaration for `cons`

`cons:` $\sigma \times$ (`list-of` σ) $\rightarrow \sigma$

In the above type declaration σ is a type variable which must be replaced by a particular type in any application of the primitive `cons`. All primitives of the VAG language are listed in table 2.2. The meaning of these primitives is essentially identical to the meaning of the corresponding primitives in Common Lisp. Each primitive is associated with a type declaration. Many of these type declarations are polymorphic. The declaration `true:`$\rightarrow$ `boolean` means that `true` is a function of no arguments which, when applied to no arguments, returns a value of type `boolean`. Hence the expression `(true)` has type `boolean`. Similar comments apply to the primitives `false` and `nil`.

All VAG expressions have a well defined type, i.e., a type not involving type variables. For example, the expression `(cons 'a (nil))` has the type `(list-of symbol)`. The type of an application of a polymorphic primitive can always be computed from the types of the arguments. The VAG types are organized into a subtype tree as described in section 2.4 and subtypes must be taken into account in type checking and type inference.

Primitive	Type Declaration
+, *, -, /	number $\times$ number $\rightarrow$ number
<, <=, >, >=	number $\times$ number $\rightarrow$ boolean
and, or	boolean $\times$ boolean $\rightarrow$ boolean
not	boolean $\rightarrow$ boolean
true, false	$\rightarrow$ boolean
=	σ $\times$ σ $\rightarrow$ boolean
if	boolean $\times$ σ $\times$ σ $\rightarrow$ σ
cons	σ $\times$ (list-of σ) $\rightarrow$ σ
car	(list-of σ) $\rightarrow$ σ
cdr	(list-of σ) $\rightarrow$ (list-of σ)
nil	$\rightarrow$ (list-of anything)
null?	(list-of σ) $\rightarrow$ boolean
member?	σ $\times$ (list-of σ) $\rightarrow$ boolean
append	(list-of σ) $\times$ (list-of σ) $\rightarrow$ (list-of σ)
map	$(\sigma \rightarrow \tau)$ $\times$ (list-of σ) $\rightarrow$ (list-of τ)

Figure 1: The Primitives of VAG

2.3 Function Definitions

The VAG language allows for recursive function definitions. For example the following function takes a width, a depth, and a list of heights and constructs a list of drawers with the given width, depth, and heights.

```
(define (make-drawers (width number)
                      (depth number)
                      (heights (list-of number)))
    (if (null?  heights)
        (nil)
        (cons (make-drawer (car heights) width depth)
              (drawers-from-dimensions width depth (cdr heights)))))
```

The above definition allows the VAG system to automatically construct the following type declaration for the function `drawers-from-dimensions`.

```
make-drawers:number×number×(list-of number)→(list-of drawer)
```

In general a definition has the following form.

```
(define (<fun> (<arg1> <type1>) ...  (<argn> <typen>))
  <body>)
```

Note that a function definition must specify the input types. In almost all natural definitions the VAG system will automatically infer the output type of the defined function. In those cases where it is unable to do this an error is generated. For example, consider the following two definitions.

```
(define (foo (n fixnum))
  (if (not (= n n))
      n
      (foo n)))


(define (foo (n fixnum))
  (foo n))
```

Both of these definitions define `foo` to be a divergent function. However, from the first definition the VAG system infers the type declaration `foo:number→number`. The first definition is accepted by the VAG system. The second definition, although equivalent to the first, is not accepted because no output type for `foo` can be infered.

2.4 Subtypes

A VAG type is either a type symbol or a type expression of the form `(list-of <type>)` where `<type>` is a VAG type. The VAG language allows subtyping both of primitive and of programmer defined types. There are three primitive numerical types in VAG — `fixnum`, `float`, and `number` with the first two being subtypes of the third. All subtype structure on list types is inherited from the subtype structure of type symbols and from the fact that all types are subtypes of the primitive universal type `anything`. For example, `(list-of fixnum)` is a subtype of `(list-of number)` because `fixnum` is a

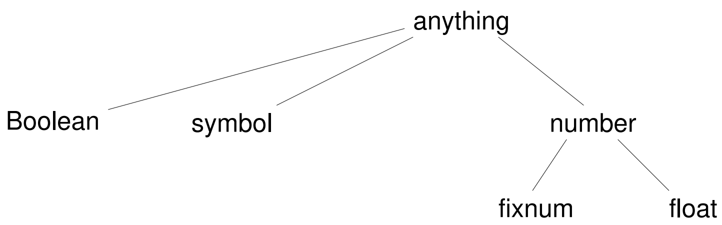

Figure 2: The primitive VAG types

subtype of `number` and all these types are subtypes of the type `anything`.

The subtype structure on type symbols must form a tree with the most general type at the root — no two types may share a common subtype (multiple inheritance is not allowed). The primitive types and their subtype structure is shown in figure 2.

As discussed in section 2.1, the VAG programmer can introduce new type symbols using the `defstruct` declaration. An option in the the defstruct declaration can be used to introduce a new type symbol as a subtype of an existing type. For example, the following declaration introduces the type `drawer-with-trim` as a subtype of `drawer`.

```
(defstruct (drawer-with-trim (subtype-of drawer))
  (drawer-trim-width number))
```

Each type introduced by `defstruct` has an associated set of attribute functions which intuitively correspond to the fields in the data structure defined by the defstruct. When the `subtype-of` option is used in a `defstruct` the attribute functions associated with the defined type include both the attributes explicitly declared in the `defstruct` plus all those attributes associated with the declared supertype. Each `defstruct` declaration also defines a "maker function" which takes the values for the fields as arguments and which returns an object of the specified type with those attribute values. When the `subtype-of` option is used the arguments to the maker function include arguments for all of the attributes associated with the declared supertype attributes with the supertype attribute arguments coming before the arguments for the explicitly declared attributes. For example, under the above declaration the function `make-drawer-with-trim` takes four arguments the first three of which specify the height, width, and depth respectively and the last of which specifies the trim width.

2.5 VAG Expressions

The expressions of the VAG language are defined in terms of the primitives of section 2.2 and a set of type declarations for nonprimitive functions. We assume that each type is associated with an infinite set of variables of that type. In practice variables are just symbols whose type in infered from context. Variables play an important role in partial designs and are discussed in more detail in a later section. It is important to note that all VAG expressions are "first order" — we do not allow VAG expressions which denote functions. The use of strictly first order expressions seems to simplify the analogy between type declarations and grammars. This analogy is fundamental to the use of VAG as a tool for generating CAD systems. A VAG expression is any one of the following.

1. A numeral or a quoted symbol. Each numeral is either of type `fixnum` or `float`. A quoted symbol is of type `symbol`.

2. A typed variable.

3. An application $(f\ e_1\ \ldots\ e_n)$ where f is one of the primitives listed in the previous section, other than `map`, and $e_1\ \ldots\ e_n$ are (recursively) expressions such that the types of $e_1\ \ldots\ e_n$ match (modulo subtyping) the polymorphic type of f. The type of an application of this form is derived from the match of the polymorphic type of f to the types of the arguments. The type of the application is always closed, i.e., it does not contain free type variables.

4. An application of the form $(f\ e_1 \ldots e_n)$ where f is a function name that has been given a type declaration and where $e_1, \ldots, e_n$ are recursively expressions whose types are subtypes of the corresponding argument types of f. The type of the application is the output type of f.

5. An expression of the form

   $$\begin{array}{l}\texttt{(let ((}v_1\ e_1\texttt{)} \\ \qquad\quad \vdots \\ \qquad\quad \texttt{(}v_n\ e_n\texttt{))} \\ \quad \texttt{<body>)}\end{array}$$

   where `body` and each e_i are recursively VAG expressions and each v_i is a typed variable whose type is the type of the corresponding expression e_i. The type of this expression is the type of `<body>`. The bindings in VAG's `let` expressions are parallel, e.g., an occurance of v_1 in the expression e_2 is taken ot be free in the `let` expression.

6. An expression of the form `(map (lambda (`v`) <body>) <list>)` where `<body>` and `<list>` are recursively expressions and v is a variable of type σ where σ is a type such that `<list>` has type `(list-of `σ`)`. The type of this expression is `(list-of `τ`)` where τ is the type of `<body>`.

It is important to note that case 3 is defined relative to a set of type declarations for function names. In practice these type declarations are derived from function definitions. A VAGCAD system user constructs a design by constructing an expression of the desired type. The VAGCAD system user, as such, does not define new VAG functions. The VAG programmer specifies the range of possible designs that can be constructed by the VAGCAD system user by defining a given set of functions to be used in designs. Of course in practice the VAGCAD system user and the VAG programmer may be the same person. The type declarations for the functions defined by the VAG programmer provide a way of associating each type with a language — the set of possible expressions of that type.

2.6 Type Declarations and Grammars

Consider the following type declarations.

```
cons:  drawer × (list-of drawer) → (list-of drawer)
nil:  → (list-of drawer)
```

In the above declarations the polymorphic type of `cons` has been replaced by a particular monomorphic instance. This allows us to view the type `(list-of drawer)` as a nonterminal of a grammar where this the nonterminal generates expressions of the form

```
(cons <drawer> (cons <drawer> ... (cons <drawer> (nil)) ... )).
```

More generally, a declaration of the form $f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \tau$ can be viewed as representation of the grammar production $\tau \Rightarrow (f\ \sigma_1\ \ldots\ \sigma_n)$. In other words, the declaration $f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \tau$ can be viewed as stating that the application $(f\ x_1\ \ldots\ x_n)$ is in the language generated by the nonterminal τ provided that each x_i is in the language generated by σ_i. If we write the type `(list-of drawer)` as `drawer-list` then the above type declarations for `cons` and `nil` can be written as grammar productions as follows.

```
drawer-list ⇒ (cons drawer drawer-list)
drawer-list ⇒ (nil)
```

The type declaration

```
make-drawer:  number × number × number → drawer
```

can be viewed as the grammar production

```
drawer ⇒ (make-drawer number number number).
```

In the VAG language we work with type declarations rather than grammar productions. Type declarations "lexicalize" the grammatical information. That is to say that the information which determines the expressions of a given type (or the language "generated by" that type) is contained in a dictionary which associates function names (words) with type declarations. The VAG programmer constructs the lexicon (set of function names) that are used by the VAGCAD user to construct particular designs.

2.7 Constraints

A VAGCAD system user designs an object by constructing an expression whose value is the desired object. The VAGCAD system user, as such, does not define new VAG functions. Instead, the VAGCAD system user builds a design from previously defined functions. The functions can be viewed as "components" or "primitives" from which a design is constructed.

A partial design is an expression in which some parts, i.e., some subexpressions, are yet to be designed and are represented by variables. The VAGCAD system uses the definitions of the defined functions to reason, to the greatest extent feasible, about the consistency of a partial design. The VAGCAD system also, to the greatest extent feasible, fills in variable values when those values are forced. It is possible to make a VAGCAD system capable of completing a design with default values or in a way that heuristically maximizes some objective function, although the currently implemented system does not have this capability.

If type constraints where the only constraints on a design, then the consistency of a partial design would be relatively easy to check. In fact, the VAGCAD system ensures at all times that the partial design under development is well typed. But there are often constraints on a design that can not be incorporated into the type system. In the VAG language constraints are represented by VAG expressions of type `boolean`. For example, `(>= x (+ y z))` is a VAG expression of type `boolean` and expresses the constraint that `x` must be at least the sum of `y` and `z`. The VAG programmer can associate constraints with both defined functions and defined types. The constraints associated with a defined function specify conditions that the arguments to that function must meet in any "consistent" application of the function. The constraints on a type specify conditions that an instance of that type must satisfy in order to be a consistent instance. A design is consistent if all primitive applications are consistent and all values of all subexpressions are consistent instances of the associated type.

The general form for a defstruct declaration is the following.

```
(defstruct (<type1> (subtype-of <type2>)
                    (with-constraint <constraint>))
  (<field> <type>)
  ...
  (<field> <type>))
```

Either, or both, of the `subtype-of` or `with-constraint` option may be omitted. If both options are omitted then the parentheses around `<type1>` can also be omitted. As mentioned above, the constraint must be a VAG expression of type Boolean. The constraint associated with a `defstruct` type declaration usually involves the special variable `self` which refers to an object of the type being introduced. An object x of the introduced type is consistent if the value of the constraint is true when `self` is interpreted as x. As an example consider the following declaration of the type `dresser`.

```
(defstruct (dresser
             (with-constraint
               (and (= (sum (map (lambda (x) (drawer-height x))
                                 (dresser-drawers self)))
                       (- (dresser-height self) 10))
                    (every (map (lambda (drawer)
                                  (= (drawer-width drawer)
                                     (- (dresser-width self) 5)))
                                (dresser-drawers self))))))
  (dresser-width number)
  (dresser-height number)
  (dresser-drawers (list-of drawer)))
```

The above constraint states that the sum of the heights of the drawers must be 10 less than the height of the dresser and the width of every drawer must be 5 less than the width of the dresser. Note that the constraint makes use of the attribute functions introduced by the `defstruct`. The functions `sum` and `every` can be defined in VAG as follows.

```
(define (sum (l (list-of number)))
  (if (null? l)
      0
      (+ (car l) (sum (cdr l)))))

(define (every (l (list-of boolean)))
  (if (null? l)
      (true)
      (and (car l) (every (cdr l)))))
```

The general form for a function definition is as follows.

```
(define (<fun> (<arg1> <type1>) ... (<argn> <typen>))
  (with-constraint <constraint>)
  <body>)
```

The `with-constraint` declaration is optional. The constraint is an expression of type `boolean` involving the variables `<arg1>` ... `<argn>`. As an example we consider the following function which takes a power source, such as a motor, and a gear train, and returns a power source consisting of the given power source followed by the given gear train.

```
(define (add-gears (motor power-source) (train gear-train))
  (with-constraint (and (>= (train-max-power train)
                            (source-max-power motor))
                        (>= (source-max-power motor)
                            (train-required-power train))
                        (gear-match (output-gear motor)
                                    (input-gear train))))
  ...)
```

Although we have not defined the types `power-source` or `gear-train`, and have not define various functions used in the above constraint, the general nature of the constraint should be clear. The constraint states that the gear train is big enough to transmit the power generated by the power source, that the power source is big enough to drive the gear train, and that the output gear of the power source matches the input gear of the power train.

It should also be noted that the functions used in the constraint need not be attribute functions, i.e., they can be defined functions as well as attribute functions introduced by a `defstruct` declaration. In general, any expression of type `Boolean` can be used as a constraint. The function `train-max-power` used in the above definition might be a defined function rather than an attribute.

To emphasize the lack of any clear distinction between attributes introduced by `defstruct` and other properties of objects consider the following definition of the volume of a drawer.

```
(define (drawer-volume (d drawer))
  (* (drawer-width d)
     (* (drawer-height d)
        (drawer-depth d))))
```

A constraint on a function taking a drawer as an argument can mention both attributes such as `drawer-width` and defined properties such as `drawer-volume`.

3 Reasoning about Partial Designs

Reasoning about partial designs is a fundamental part of the VAGCAD system. The system must be able to make inferences about partial designs to both check consistency and to infer aspects of the design. To provide helpful user support the VAGCAD system must be able to reason effectively from the constraints associated with types and defined functions. It is important that the VAG programmer understand what support is being provided to the VAGCAD system user. The constraint based reasoning process is not able to draw all possible valid conclusions about a given partial design. To understand exactly what conclusion the VAGCAD is able to draw one must have an understanding of the inference mechanisms involved.

Partial designs do not contain sufficient information to allow the computation of a specific value for all of the subexpressions appearing in the design. The constraint based reasoning done by the VAGCAD system is not in general able to run the defined functions to compute values. Rather than compute values, the VAGCAD system maintains a data base of facts about a current partial design. This database of facts includes values for some, but usually not all, of the subexpressions of the design. For example, if `d` is a variable ranging over drawers, then the database maintained by the VAGCAD system might include the assertion `(= (drawer-volume d) 200)`. This fact assigns a value to the expression `(drawer-volume d)`. It is possible that the system has assigned a value to `(drawer-volume d)` even though it has not assigned a value to the height, width, or depth of `d`. However, if the system has assigned a value to the volume, the height, and the width, then it will automatically assign a value to the depth. The definition of `drawer-volume` establishes a constraint between the volume, height, width, and depth which the system can use in four different directions — any one of the volume, width, height, or depth can be derived from the constraint given the other values.

As mentioned above, the VAGCAD system maintains a database of facts about the current partial design. In describing the inference process I will assume that the data base is a set of assertions where each assertion is a VAG expression of type `boolean`. Note that one can add an assertion to a data base without being able to "compute" the truth value of that assertion. For example, we can prove

```
(= (drawer-volume d)
   (* (drawer-height d)
      (* (drawer-width d)
         (drawer-depth d))))
```

even though the height, width, and depth of `d` are unknown.

The input to the VAGCAD reasoning system is a partial design, i.e., a VAG expression containing typed variables which represent yet to be designed components. Given a partial design the system constructs a data base of assertions about that design. Initially this data base simply contains an assertion of the form (`consider-expression` s) where s is the partial design (recall that a partial design is just an expression).

The reasoning processes employed by the VAGCAD system can all be characterized by inference rules operating on a data base of assertions where the initial assertion is of the form (`consider-expression` s). Each inference rule states that whenever facts of a certain form are in the data base, other facts should be added. This is equivalent to stating that the constraint based reasoning mechanisms used in the VAGCAD system can be defined by forward chaining inference rules. Forward chaining inference rules have been widely studied as a foundation for expert systems and data base inference mechanisms [Ullman, 1988], [Naughton and Ramakrishnan, 1991], [Givan and McAllester, 1992].

The inference process described in sections 3.2 through 3.8 is a "batch" process, i.e., it starts from the single assertion (`consider-expression` s) and builds a data base by applying a large collection of forward chaining rules. This description of the inference process is a convenient fiction. The actual inference process is incremental — it allows for the incremental refinement of a design and the database is updated incrementally as additional design decisions are made. The method for making the reasoning process incremental is discussed in section 3.9.

3.1 Variables

A partial design is an expression containing free variables. These variables should be viewed as placeholders for parts of the design that are to be filled in at some later point. In VAG the free variables appearing in a partial design must be declared. This is done with a `defvar` declaration of the following form.

```
(defvar <varname> <type>)
```

This `defvar` construct should not be confused with the defvar caonstruct of Common Lisp. The VAG version of defvar is a seperate symbol in the VAG package. An example of the use of `defvar` is given in section 3.8.

3.2 Considering Expressions and Installing Constraints

For simplicity of presentation we assume that the inference process starts with the single assertion (`consider-expression` s) for some partial design s. Assertions of the form (`consider-expression` w) "propagate down" to subexpressions of the design. We say that an expression w is "considered" by the reasoning process if the assertion (`consider-expression` w) is in the data base. The reasoning process contains an inference rule which states that if an application $(f\ w_1\ \ldots\ w_n)$ is considered then each w_i becomes considered. An exception to this rule is when the function is the primitive `if` — if the system is considering the expression (`if` $s\ w_1\ w_2$) then it considers s but does not consider w_1 or w_2 unless the value of s can be determined in which case w_1 us considered if s is true and w_2 is considered if s is false. Consideration also does not propagate into the λ-expression that is the first argument to an application of `map`, although consideration does propagate into the second argument of `map`. In all other applications of either primitives or defined functions consideration propagates to all arguments. Additional mechanisms for propagating consideration in `map` and `let` expressions are discussed in section 3.6.

There are two inference rules for installing constraints. The first rule states that if the system is considering an expression s of type σ and the type σ is associated with a constraint Φ[`self`] as described in section 2.7, then the system adds both $\Phi[s]$ and (`consider-expression` $\Phi[s]$) to the data base. For example, if the type `drawer` is associated with the constraint `(> (drawer-height self) 0)`, and the system is considering an expression d of type `drawer`, then the system adds the assertion `(>= (drawer-height` d`) 0)` to the data base.

The second rule for installing constraints states that if the system is considering an application of the form (f w_1 ... w_n) where f is a defined function, and f is associated with a constraint of the form $\Phi[x_1, \ldots x_n]$ as described in section 2.7, then the system adds both $\Phi[w_1, \ldots, w_n]$ and (`consider-expression` $\Phi[w_1, \ldots w_n]$) to the data base. For example, consider a function `attach-threaded-pipes` which takes two pipe specifications and constructs a specification for the result of threading the first pipe into the second pipe. By associating a general constraint with the defined function `attach-threaded-pipes` the VAG programmer can ensure that whenever the system considers an application of the form (`attach-threaded-pipes` p_1 p_2) it adds the assertion (`threads-match` p_1 p_2) to the data base. It also considers the expression (`threads-match` p_1 p_2) which will cause, among other things, an expansion of the definition of the function `threads-match`.

Note that the assertions added by the constraint installation rules are all VAG expressions of type `boolean`. Many of the inference rules discussed in the following sections operate on Boolean VAG expressions.

3.3 Values

A partial design may have subexpressions which are "completed" in the sense that they do not contain free variables, i.e., there are no "yet to be designed" subparts. Whenever the system considers such a completed expression it computes the value of that expression. There are also methods for deriving values that do not involve direct computation, such as the propagation methods described in section 3.4.

We say that the system has determined a value for an expression s if the data base contains an equation of the form (`=` s v) where v is a special kind of VAG expression called a value. Numerals are VAG values as well as are the expressions `(true)`, `(false)`, `(nil)` and `(cons 1 (nil))`. More generally, a value is either a numeral, one of the expressions `(true)`, `(false)`, or `(nil)`, an expression of the form `(cons` v_1 v_2`)` where v_1 and v_2 are values or an expression of the form `(make-foo` v_1 ... v_n`)` where each v_i is a value and `make-foo` is a data structure constructor function introduced by a `defstruct` declaration.

3.4 Boolean Inference

Boolean inference is an NP hard problem — determining whether one Boolean formula follows from another can take exponential time in the size of the two formulas. The inference mechanisms employed in the VAGCAD system run in linear time but are not complete for Boolean inference. They correspond to the Boolean constraint propagation (BCP) rules that have been used by a variety of constraint processing systems [McAllester, 1990]. The BCP rules are used to derive values for expressions of type Boolean.

It is important that Boolean expressions have values in the same sense that other expressions, such as numerical expressions, have values. The system has inference rules stating that an assertion Φ is in the data base then the equation `(=` Φ `(true))` is also in the data base, i.e., if the system can prove Φ then it can assign Φ the value `(true)`. The converse also holds. If the system determines that Φ has value `true` then it adds Φ to the data base. I will use the phrase "Φ is true" as an abbreviation for either "the data base contains Φ" or "the data base contains `(=` Φ `(true))`". Similar statements hold for the value `(false)`. The system ensures that `(not` Φ`)` is in the data base if and only if Φ has been assigned the value `(false)` and I will use the phrase "Φ is false" to abbreviate this condition.

The BCP rules for deriving Boolean values are local in the sense that they relate a formula to its immediate subformulas. For example, if the system has determined that Φ is true, and the system is considering the expression `(not` Φ`)` (in the sense of section 3.2), then the system will derive that `(not` Φ`)` is false. Similarly, if the system has derived that both Φ and Ψ are true and is considering the expression `(and` Φ Ψ`)` then it will derive that the expression `(and` Φ Ψ`)` is true. Note that it is important that this inference rule is restricted to expressions under consideration. Otherwise the system would derive an infinite set of formulas of the form `(and` Φ `(and` Φ ... `(and` Φ Φ`)` ...`))` whenever it derived an assertion Φ. Another rule states if the system has determined that `(and` Φ Ψ`)` is true then it will infer that Φ is true and that Ψ is true. In general, BCP includes all logically valid inference rules relating a Boolean operation to the arguments of that operation. The reader is referred to [McAllester, 1990] for more details.

As stated above, BCP does not derive all assertion derivable by Boolean reasoning. For example the rules will not derive Φ from `(or` Φ Ψ`)` and `(or` Φ `(not` Ψ`))`. This inference, while sound, is not local — it can not be derived from the local propagation of values through a single Boolean expression.

3.5 Equality Reasoning

Equality reasoning is perhaps the most subtle aspect of the VAGCAD reasoning system. In classical first order logic there are four fundamental inference rules for equality which express the properties of reflexivity, transitivity, symmetry, and the substitution of equals for equals. These four rules are implicitly present in the VAGCAD reasoning system. However, it is worth noting that if the equality rules are implemented in a direct manner then the system would be forced to derive an infinite set of conclusions. For example if the system derived `(=` s `(f` s`))` for some expression s then it would be forced to derive `(=` s `(f (f` s`)))`, `(=` s `(f (f (f` s`))))`, and so on. The VAGCAD reasoning system uses congruence grammars to get the full power of the four inference rules for equality with a terminating inference process [McAllester, 1992].

Congruence grammars are too complex to be fully described here. However, a few comments are worth mentioning. Congruence grammars are formally equivalent to finite state tree automaton [Thomas, 1990]. However, the grammar view of this formalism allows each equivalence class to be viewed as the language generated by a nonterminal of a grammar. The data structure used to represent the equivalence relation is quite similar to a union-find data structure [Kozen, 1992] where there is a table called `find` such that two terms s and t are taken to be equal exactly when `find`(s) is the same as `find`(t). In the tree automaton formalism the value of `find`(s) is the grammar nonterminal representing the equivalence class of s. The algorithms underlying the construction and maintenance of congruence grammars are quite similar to congruence closure algorithms [Kozen, 1977], [Downey *et al.*, 1980], [Shostak, 1978]. The details can be found in [McAllester, 1992]. Although for efficiency reasons the actual equality data structures used in the implementation of the VAGCAD system are somewhat subtle, the VAG programmer need not understand these mechanisms. To understand the content of the equality reasoning done by the system the VAG programmer can assume that the data base is closed under all uses of the four basic inference rules for equality. These rules imply that if s and w have been proven to be equal and the database contains an assertion of the form $\Phi[s]$, then the data base will behave just as if it also contains the assertion $\Phi[t]$.

One particularly important consequence of the inference rules for equality is that if the system is considering an expression s and the system has proved that s equals another expression w, then the system behaves just as if it is also considering the expression w.

3.6 Partial Evaluation

Partial evaluation is a process of "partially" computing the value of an expression. For example, the expression `(cdr (cons` x `(cons` y `(nil))))` can be "evaluated" to `(cons` y `(nil))` even though the value of x and y have not yet been specified. The most fundamental partial evaluation inference rule is that if the system is considering an application of the form `(`f w_1 ... w_n`)` and f is a defined function with definition

```
(define (f (x1 σ1) ... (xn σn))
  b[x1, ... xn])
```

then the system infers the equation `(= (`f w_1 ... w_n`)` $b[w_1, \ldots, w_n]$`)`. This is the well known inference rule of β reduction in the λ-calculus [Barendregt, 1984].

Note that β-reduction is only performed on expressions which are being considered by the system. Also recall that consideration does not propagate into the branches of a conditional. This prevents infinite β-reduction provided that the defined functions terminate when applied to values. For example, consider the following standard definition of the factorial function.

```
(define (fact (n fixnum))
  (if (<= n 0)
      1
```

```
      (* n (fact (- n 1)))))
```

Suppose that the system is considering the expression `(fact` n`)` where n is a variable of type `fixnum`. The β-reduction inference rule will generate the equation

```
(= (fact n)
   (if (<= n 0)
       1
       (* n (fact (- n 1)))))
```

This equation will cause the system to behave as if it were also considering the `if` expression that is the β-reduction of `(fact` n`)`. If consideration propagated into the branches of a conditional then this would cause the system to consider the expression `(fact (-` n `1))`. This would cause another β-reduction and the system would also end up considering `(fact (- (- n 1) 1))`. So if consideration propagated into the branches of a conditional then the β-reduction would cause the system to generate an infinite number of terms. This kind of divergent computation is a well known problem for partial evaluation. Preventing propagation of consideration into the branches of conditionals prevents divergent computation in most cases but results in less inference than is performed by more aggressive forms of partial evaluation.

The VAGCAD reasoning system also has a partial evaluation inference rule for `let` expressions. If the system is considering a `let` expression of the form

```
(let ((x1 w1)
       ⋮
      (xn wn))
  b[x1, ..., xn])
```

then the system automatically generates the equation

```
(= (let ((x1 w1)
           ⋮
          (xn wn))
```

    $b[x_1, \ldots, x_n]$)
  $b[w_1, \ldots, w_n]$).

This rule is very similar to the β-reduction rule for function applications. In fact `let` expressions can be viewed as abbreviations for applications of λ-expressions and this rule is really just a variant of β-reduction.

In addition to the fundamental β-reduction rule the VAGCAD system has a variety of partial evaluation rules specific to the primitives of the language. Most significantly, if the system is considering an `if` expression (`if` Φ s w) and the system determines that Φ is true, then it generates the equation (= (`if` Φ s w) s). This causes the system to consider the expression s. If the system determines that Φ is false then it generates the equation (= (`if` Φ s w) w) which cases the system to consider the expression w. In this way consideration can propagate through `if` expressions in those cases where the test can be assigned a truth value.

Another set of partial evaluation rules are specific to the list manipulation functions `cons`, `car` and `cdr`. If the system is considering the an expression of the form (`car` (`cons` s w)) then it generates the equation (= (`car` (`cons` s w)) s). A similar rule holds for `cdr`. The partial evaluation rules treat the functions `member?` and `append` as defined functions — partial evaluation of these functions is performed using the β-reduction rule on standard recursive definitions of these functions. Since procedures are not first class objects in the VAG language the β-reduction rule for `map` must be treated as a special case. If the system is considering an expression of the form (`map` (`lambda` (x) $b[x]$) (`cons` s w)) then it automatically generates the equation

(= (`map` (`lambda` (x) $b[x]$) (`cons` s w))
  (`cons` $b[s]$ (`map` (`lambda` (x) $b[x]$) (`cons` s w)))).

If the system is considering an expression of the form (`map` f (`nil`)) then it automatically generates the equation (= (`map` f (`nil`)) (`nil`)).

There are also special partial evaluation rules for the functions introduced by `defstruct` declarations. For example, if `drawer-height` is the first attribute function associated with the type `drawer` and the system is considering an expression of the form (`drawer-height` (`make-drawer` s w

u)) then the system generates the equation

(= (drawer-height (make-drawer s w u)) s)

Note that the rules for the maker and attribute functions associated with `defstruct` declarations are analogous to the rules associated with `cons`, `car`, and `cdr`.

3.7 Numerical Reasoning

The VAGCAD system maintains and propagates bounds information on numerical expressions. The bounds propagation rules are the obvious rules for bounds on arithmetic expressions. For example if the system has derived the assertions (>= s 3) and (>= w 5), and the the system is considering the expression (+ s w), then the system will derive the expression (>= (+ s w) 8). Similarly, if the system has derived (<= (+ s w) 5) and (>= s 2) then the system will derive (<= w 3). In general, the system performs all sound inferences which relate bounds on an application of an arithmetic operation to bounds on the arguments in that application. The numerical constraint propagations mechanisms used here are quite similar to the those that have been used in a variety of systems, e.g., [Sussman and Steele, 1980].

3.8 An Example

Consider the definition of the type dresser given in section 2.7 and repeated below.

```
(defstruct (dresser
             (with-constraint
               (and (= (sum (map (lambda (x) (drawer-height x))
                                 (dresser-drawers self)))
                       (- (dresser-height self) 10))
                    (every (map (lambda (drawer)
                                  (= (drawer-width drawer)
                                     (- (dresser-width self) 5)))
                                (dresser-drawers self))))))
  (dresser-width number)
```

```
  (dresser-height number)
  (dresser-drawers (list-of drawer)))
```

The constraint associated with this declaration involves the defined functions `sum` and `every` whose definitions are given in section 2.7. Now suppose that we ask the system to consider the following partial design.

```
(defvar w number)
(defvar h1 number)
(defvar w1 number)
(defvar d number)
(defvar w2 number)

(make-dresser w 50
  (list (make-drawer h1 w1 d)
        (make-drawer 15 w2 d)
        (make-drawer 15 30 d)))
```

The constraints generated by this expression will be partially evaluated to yield the following.

```
(and (= (+ h1 (+ 15 15)) (- 50 10))
     (and (= w1 (- w 5))
          (and (= w2 (- w 5))
               (and (= 30 (- w 5))
                    (true)))))
```

Given these formulas VAGCAD's Boolean and numerical reasoning will derive that `h1` is 10, `w` is 35, and `w1` and `w2` are both 30.

3.9 Incremental Reasoning

The VAGCAD reasoning system operates by applying forward chaining inference rules to an assertional data base. This reasoning process can easily be made incremental with respect to refinements in the design. Consider a partial design $s[x_1, \ldots, x_n]$ where $x_1, \ldots, x_n$ are yet to be designed components represented by variables. We can refine the design by adding an assertion to the data base of the form (= x_i w) where w is an expression which may have additional yet to be designed components. The VAGCAD inference system is incremental with respect to any additional premises added to the assertional data base.

The VAGCAD inference process also maintains and "undo stack" so that the state of the data base can be restored to any earlier state. In this way additional premises can be incrementally retracted in the reverse of the order in which they were asserted. This architecture is well suited for "case analysis" in which assumptions can be incrementally made and retracted. For example, the system can "try out" each of a list of components c_1, … c_n as values for a variable x by incrementally adding and retracting assertions of the form (= x c_i).

3.10 External Functions

The VAGCAD evaluation mechanism is far less efficient than the Lisp evaluator which is in turn far less efficient than compiled Lisp code. So functions defined in the VAG language run quite slowly compared with functions defined in traditional programming languages. The VAG language is optimized for reasoning about partial designs, i.e., expressions which contain free variables. If the VAGCAD system needs to reason about the definition of a function in cases where the arguments are not fully specified, and if the function is relatively simple, then the function should be defined by a VAG definition as described in sections 2.3 and 2.7. However, if the function is complex and requires significant computational resources then it is better to implement the function directly in Lisp. The VAG system allows for functions defined directly in common lisp to be added as "external" VAG functions. It is possible to associate these external functions with constraints. The associated constraints allow reasoning about applications of external functions even when the values of the arguments can not be determined.

The general form of a definition of an external function is the following.

```
(defprim (<fun-name> (<arg1> <type1>) ... (<argn> <typen>))
  (output-type <type>)
  (with-constraint <constraint>)
```

```
  <body>)
```

Note that the syntactic structure of definitions of external functions is very similar to the syntactic structure of ordinary definitions in the VAG language. The only difference is that the body of the definition is written in Common Lisp rather than the VAG language and that in the output type must be explicitly declared. The VAGCAD reasoning system uses the constraints associated with external functions in the same way that it uses the constraints associated with functions defined in the VAG language. However, the VAGCAD system does not perform partial evaluation β-reductions on applications of external functions. Such β-reductions are not possible because the VAGCAD system does not "understand" the syntax and semantics of Common Lisp — the inference rules for VAG expression are not sound when applied to Common Lisp expressions.

When the VAGCAD reasoning system is considering an application of the form `(`f w_1 ... w_n`)` where f is an external function, and each w_i has been assigned a value, the VAGCAD system translates the VAG values assigned to each w_i into Common Lisp data structures and applies a compiled version of the external function f to these data structures to get a Common Lisp value for the application. This Common Lisp value is then translated back into a VAG value which is used as the value of the application expression. The mapping between VAG values and Common Lisp values is straightforward. The VAG expressions `(true)`, `(false)` translate to the Common Lisp symbols `TRUE` and `FALSE` respectively (in the vag package). The VAG expression `(nil)` translates to the Common Lisp value `NIL`. The VAG function `cons` corresponds to the Common Lisp function `cons`. The VAG expression `(make-foo` v_1 ... v_n`)` corresponds to the Common Lisp expression `(LIST 'IMP::FOO` v_1 ... v_n`)` where `IMP` is a special "internal" package used for machine generated symbols.

As an example we could define a cubic-drawer primitive as follows.

```
(defprim (cubic-drawer (n number))
  (output-type drawer)
  (with-constraint (and (= (drawer-height (cubic-drawer n)) n)
                        (= (drawer-width (cubic-drawer n)) n)
                        (= (drawer-depth (cubic-drawer n)) n)))
```

```
  (list 'imp::drawer n n n))
```

Of course the above function could have been defined directly in the VAG language. However, in the above definition the body of the function is written in Common Lisp rather than VAG. The VAGCAD reasoning system does not "understand" the body of the above definition. However, when considering an application (cubic-drawer n), where n is a variable, it can use the constraints associated with cubic-drawer to prove the following equation.

```
(= (drawer-height (cubic-drawer n))
   (drawer-width (cubic-drawer n)))
```

Of course the external function feature is intended to be used with functions which require much more computation than cubic-drawer. A more realistic example of would be a function compile-expression which took an algebraic expression as input and produced a sequence of RISC machine instructions as an output. Such a function could be associated with constraints relating the number of instruction in the output to the number of algebraic operators in the input. This would allow the VAGCAD reasoning system to reason about an expression such as (compile-expression (simplification e)) even in cases where the value of e is not yet fixed.

4 Using the System

This section describes installation and use of the VAG system. The VAG system is implemented in Lucid Common Lisp. The code can be retrieved by anonymous ftp from the directory ftp.ai.mit.edu:/pub/users/dam/vags3. This directory contains a readme file, a postscript version of this document, and the seven lisp files vag-system.lisp, util.lisp, undo.lisp, defnoticer.lisp, vag.lisp, vag2.lisp, and vagtest.lisp. These seven lisp files should be copied to a local directory. Then from the read-eval-print loop of Lucid Common Lisp one should run the following.

```
> (user::set-working-directory <working directory>>)
...
> (load "vag-system")
```

```
...
> (load-vag)
...
> (in-package :vag)
...
```

This will cause the Lisp files to be compiled (if they are not already) and loaded. The `vag` package corresponds to the VAG language. Once in this package `defstruct`, `define`, and `defprim` expressions can be executed as Common Lisp expressions. The version of `defstruct` in the `vag` package is separate from Common Lisp's `defstruct`. It is possible to redefine any VAG function or type by simply executing another `define`, `defprim`, or `defstruct` form.

The file vagtest.lisp contains a set of test cases and examples and should be loaded seperately.

The VAGCAD reasoning system can be invoked using the following functions:

goto-context. This function takes a list of premise assertions and sets the VAGCAD data base to be the result of applying the VAGCAD inference rules to the premises in the given list. Each premise in the context must have the form `(consider-expression` e`)` where e is a vag expression or `(assert` e`)` where e is a vag expression of type Boolean. For example

```
(goto-context '((consider-expression (make-drawer h w d))
                (assert (= d 3))))
```

will set the data base to be the result of applying the VAGCAD inference rules to the premises `(consider-expression (make-drawer h w d))` and `(assert (= d 3))`. In these assertions `h`, `w` and `d` are variables. The function `goto-context` is incremental in the sense that the system will do a minimum of backtracking and incremental premise addition to move from the current premise set to the requested premise set.

If a function or struture is redefined then the meaning of expressions in the current context may change. For this reason, whenever a function or structure is redefined the system goes to the null context.

contradiction? This is a function of no arguments and returns `T` if the current data base contains a contradiction such as two different values for the same expression. If there is no contradiction in the current data base then this function returns `nil`. Recall that the inference rules are incomplete. This means that it is possible that a given premise set is in fact impossible (there is no situation in which all of the given premises are true) and yet the reasoning system is unable to find the inconsistency.

value-of. This function takes a VAG expression and returns a list of two values. If the given VAG expression has a value under the current assertional data base, this function returns a list containing the Common Lisp data structure representing the value and the flag T. If the given VAG expression does not have a value under the current assertional data base then this function returns `(nil nil)`. The second element of the list is used to distinguish between having the value `nil` and not having a value.

why. This function is the standard way in which justifications are examined. It takes an expression and first determines if that expression has a value in the current context. If the expression is of type Boolean, then this is equivalent to determining if the expression has a truth value. If no value has been determined thshen a message to that effect is printed. If a value has been determined then a justification for that value is printed. The function `why` maintains a stack of justifications. The justification on the top of this stack is called the current justification and the justification just below the current justification is called the parent justification. When called on an expression the function `why` clears the justification stack then makes the justification it prints (if any) the current justification. If the function `why` is then given an integer n it will return the nth "subjustification" of the current justification. This subjustification becomes the current justification and the original justification becomes the parent. If the function `why` is given the symbol `up` it will pop the justification stack so that the old parent justification becomes the current justification. It then prints the old parent justification (which is now current). If the function `why` is passed the symbol `current` it will print the current justification without changing the state of the justification stack. These features allow one to easily "walk" the justification tree underlying any given value.

justification-for. This function takes one agrument which must be an expression (unlike `why`, the function `justification-for` doe not handle integers and the symbols `up` and `current` as tree walking commands). If the given expression has a value and that value has a justification in the current context then this function returns the justification data structure. This justification can be used in various ways by the functions described below.

print-justification. This function will print the data in a justification on the default output stream. It prints in a way that allows one to walk the justification tree easily.

just-assertion. This function takes a justification and returns the "assertion" which is justified by that justification. In many cases this assertion is one that is used in the internals of the implementation. However, the intuitive meaning of the assertion should be clear.

just-rule-name. This takes a justification and reuturns the name of the rule of inference, or inference principle, used to derive the justified assertion. The special rule name `context-premise` means that the assertion is an assumption in the current context. The name is usually a symbol but it can also be an expression of the form `(vag-definition` s`)` where s is a symbol, either a function name or a type name. Rule names not of one of these two forms should probably be ignored since they represent details of the internal implementation (they correspond to procedure names in the internal implementation).

just-subjustifications. This takes a justification and returns a list of justifications. The justifications returned are justifications for the assertions used to derive the assertion justified by input justification. This function allows one to write programs which walk down the justification tree. For example, it is not difficult to write a function which finds all of the context premises used in deriving a given assertion.

`*record-justifications*`. This is a variable which is set to `T` by default. The efficiency of the system can be improved somewhat by setting this variable to `nil`. However, once this is done the system is no longer able to generate justifications for derived facts.

Ideally, the VAGCAD user would work in terms of a graphical user interface which allowed a design to be incrementally developed with options presented on menus. However, the software described above does not provide such an interface. The best interface for the software described here is the ilisp package for running lisp as an inferior process to emacs. The VAG programmer can develop VAG functions in much the same manner as a

Lisp programmer develops lisp functions. Presumably an appropriate emacs based interface for the VAGCAD system user can also be developed although the current software does not support such an interface.

Justifications have two purposes — debugging the VAG implementation itself and providing explanations to VAG programmers and VAGCAD system users. The above functions are perhaps bes suited to the problem of debugging the VAG implementation itself. For a VAG programmer, or VAGCAD system user, the justifications should be presented at a higher level. This can be done by writing procedures which summarize justification trees. To write such procedures it is necessary to know the precise structure of the assertions appearing in justifications. First, one must know that the assertions contain "internal expressions". An internal expression is a VAG expression in which each variable foo has been replaced by `(:the-variable (quote foo))` and each numberal n has been replaced with `(quote` n`)`. For example the internal version of `(+ x (+ 1 2))` is

```
(+ (:the-variable (quote x)) (+ (quote 1) (quote 2))).
```

In general quotation is used to flag constants (numbers or symbols). The constant `(quote x)` is used as the "name" of the variable `(:the-variable (quote x))`. The expression returned by `just-assertion` is always of one of the following forms.

- `(has-value` e v`)` where e is an internal assertion and v is a Lisp value, i.e., a number symbol or cons cell containing two values. In may cases e is an expression of type Boolean and v is either the symbol `true` or the symbol `false`.

- `(=` e_1 e_2`)` where e_1 and e_2 are internal expressions. This has the obvious meaning, i.e., the two expressions have the same value even though that value may not be known.

- `(has-type` e t`)` where e is an internal expression and t is a symbol. This asserts that the value of e is an instance of type t.

- `(upper-bound` e n`)` where e is an internal expression and n is a number. This states that the value of e is a number and that n is an upper bound on that value (the value may equal n).

- `(lower-bound` e n`)`. This is analogous to `(upper-bound` e n`)`.

- `(consider!` e`)` where e is an internal expression. This is a "control assertion", i.e., it controls the inference process but not make any claim about e. These assertions should probably be hidden from the user.

In writing procedures which summarize justifications it is probalby best to ignore rule names other than `context-premise` and rule names of the form `(vag-definition` s`)` where s is a symbol.

5 Conclusion

A variational attribute grammar is formally simply a set of function definitions in the VAG language. The type declarations of the defined functions can be viewed as a grammar — it defines a set of well typed expressions. The attributes of an expression w are given by other expressions such as `(drawer-width` w`)` and attribute values can be derived using a multidirectional constraint based reasoning process.

The VAG language provides the VAG programmer with a method of rapidly constructing a variational attribute grammar with powerful inference mechanisms to support the VAGCAD system user. The development of an appropriate VAGCAD system user interface will be an important next step in determining the empirical utility of the VAG approach to the construction of CAD systems.

References

[Adjuciewicz, 1935] Kasimierz Adjuciewicz. Die syntaktische konnexitat. *Studia Philophica*, 1:1–27, 1935. Translated as "Sytactic Connection" in Strolls McCall (ed), *Polish Logic: 1920-1939* (Oxford University Press, 1967).

[Bach, 1988] Emmon Bach. Categorial grammars as theories of language. In Richard Oehrle and Edmond Bach, editors, *Categorial Grammars and Natural Language Structures*, pages 17–34. D. Reidel, 1988.

[Barendregt, 1984] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics, Revised Edition*, volume 103 of *Studies in Logic*. Elsevier Science Publishers B.V., 1984.

[Cardelli and Wegner, 1985] Luca Cardelli and Peter Wegner. On understanding data abstraction and polymorphism. *Computing Surveys*, 17(4), 1985.

[Courcelle and Deransart, 1988] B. Courcelle and P. Deransart. Proofs of partial correctness for attribute grammars with applications to recursive procedures and logic programming. *Information and Computation*, 78:1–55, 1988.

[Deransart *et al.*, 1988] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Defintions, Systems and Bibliography*. Springer-Verlag, 1988. Lecture Notes in Computer Science Vol. 323.

[Downey *et al.*, 1980] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpression problem. *JACM*, 27(4):758–771, October 1980.

[Givan and McAllester, 1992] Robert Givan and David McAllester. New results on local inference relations. In *Principles of Knolwedge Representation and Reasoning: Proceedings of the Third International Conference*, pages 403–412. Morgan Kaufman Press, October 1992. internet file ftp.ai.mit.edu:/pub/dam/kr92.ps.

[Kozen, 1977] Dexter C. Kozen. Complexity of finitely presented algebras. In *Proceedings of the Ninth Annual ACM Symposium on the Theory of Compututation*, pages 164–177, 1977.

[Kozen, 1992] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.

[McAllester, 1990] David McAllester. Truth maintenance. In *Proceedings AAAI90*, pages 1109–1116. Morgan Kaufmann Publishers, 1990. internet file ftp.ai.mit.edu:/pub/dam/aaai90.ps.

[McAllester, 1992] David McAllester. Grammar rewriting. In *CADE-11*, pages 124–138. Springer-Verlag, June 1992. internet file ftp.ai.mit.edu:/pub/dam/cade92.ps.

[Milner *et al.*, 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[Naughton and Ramakrishnan, 1991] Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. MIT Press, 1991.

[Reps and Teitelbaum, 1985] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Manual*. Cornell University, Ithaca, NY, August 1985.

[Shostak, 1978] R. Shostak. An algorithm for reasoning about equality. *Comm. ACM.*, 21(2):583–585, July 1978.

[Steele, 1990] G. L. Steele. *Common Lisp: The Language (Second Edition)*. Digital Press, 1990.

[Sussman and Steele, 1980] Gerald J. Sussman and Guy Lewis Steele. Constraits — a language for expression almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[Thomas, 1990] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics*, pages 133–164. MIT Press, 1990.

[Ullman, 1988] J. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.