Pattern Matching for Program Generation: A User Manual

Ted J. Biggerstaff

December, 1998

Technical Report
MSR-TR-98-55

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Pattern Matching for Program Generation: A User Manual

Ted J. Biggerstaff

Abstract

The Anticipatory Optimization Generation (AOG) system is built to generate custom programs using large-scale, programmatic transformations (versus pure pattern-based transformations) that operate on Abstract Syntax Trees (ASTs). Because of the scale of the AOG transformations and the often-complex program reorganizations that they enable, the transformations must operate on the low-level physical formats of the ASTs and simultaneously accommodate high degrees of variation in the ASTs. However, both the physical details of and variations within an AST are irrelevant to the AOG system in the same sense that both the physical details of and variations within a database format are irrelevant to a database management system. In both cases, only the logical structure is relevant. Database management systems introduce logical schemas to hide the details and variations in the format of their databases. Similarly, the AOG pattern matcher introduces a pattern notation that performs the analogous service for ASTs. It hides the details of and variations within an AST from the large-scale transformations that must operate on the AST and thereby reduces the complexity of the transformation code.

The pattern notation is embedded in the Lisp language and provides a set of Prolog-like operators for performing complex matching and inference logic. The operators include analogs to the Prolog operators and, or, not, mark, cut, bind, is, succeed, fail, prove, and rule definition (i.e., <-). They also include a number of more specialize operators such as operators that span sections of an AST, bind the remainder of a list, perform top-down and bottom-up tree searches, rename local variables, use dynamically computed variable values as patterns, invoke and use rule sets, navigate object structures, perform recursive matching operations, and execute arbitrary Lisp expressions. The pattern matcher is fully backtracking and is built in a “continuation passing style” of programming.

This user manual describes the pattern notation and the operation of the pattern matching system that evaluates that notation. It also includes a number of extended examples drawn from the AOG system that illustrate the use of the pattern notation in the context of transformation-based program generation.

1 Introduction

The Anticipatory Optimization Generator system is built on large scale transformations that accommodate much variation in the patterns of the program segments that they can deal with. (See Biggerstaff, 1998a-c.) As a consequence, AOG requires the ability to express patterns in a highly abstract and compact form to accommodate the large scale while at the same time, it requires high pattern variability to accommodate the variability of the programs that it deals with. This document describes the pattern matcher that was built to satisfy these requirements.

2 The Pattern Matcher

2.1 The Design Philosophy

For generation systems, there is a trade off between generality of specification and performance of the generator. This trade off depends on the granularity and generality of the specification mechanisms. Small highly general specification mechanisms (e.g., as found in generators based on theorem proving engines or highly general, small grain transformational rewrite systems) induce very large search spaces. Thus, while such generators are highly general and apply to virtually any domain, their performance may prevent them from scaling beyond toy problems. On the other hand, large grained, highly specific specification mechanisms (e.g., as found in domain specific, programmatic generators and large grained transformation systems) run fast even in the face of large-scale problems but are often restricted to a few narrow domains and resist easy transfer between domains.

I believe that this is not an unsolvable dilemma. I believe that some representational choices can achieve substantial amounts of both generality and performance. The strategy that I take is to provide a general specification notation (i.e., the AOG pattern notation) that can be used in the context of large grained and otherwise, highly specific specification mechanisms (i.e., programmatic transformations). Because the notational realization is really **data**, it plays a parametric role for the transformations that use it and thereby, allows them go be more general. That is, the large grain transformations are specific to the **conceptual structures** of the programs upon which they operate but are independent of the concrete details and variations of the programs. Additionally, the pattern notation attempts to be highly declarative thereby, avoiding the obscurity of low level prescriptive code for AST operations. Finally, the pattern notation is much more compact than prescriptive code and that too enhances its usefulness.

The pattern notation and the associated pattern matching software is used as the central engine of the AOG program generation system.

2.2 The Implementation Philosophy

The AOG pattern matcher is embedded in Lisp and views the Lisp system, its constructs, and its libraries as a rich source of reusable components without which the matcher would not have been feasible within the time and resources available. The current version of the matcher is implemented in roughly 2000 lines of Common Lisp using the FranzLisp CL development environment. It runs on Windows 98^{TM} and NT^{TM}.

2.3 Patterns

2.3.1 Literal Data

The AOG pattern matcher is a unification-based system for matching data structures against patterns. The patterns are expressed in an *inverse quoting* representation. That is, list or data literals in a pattern are expressed directly without any syntactic adornment but executable or operational elements of a pattern require syntactic adornment. For example, a pattern representing a list of three data literals a, b and c would be expressed directly as the list

```
(a b c)
```

However, if we want to match other than literal data, which one almost always does, then we need operational elements in the pattern. These include pattern variables and pattern operators.

2.3.2 Pattern Variables

If the example in the previous section is used as a pattern, it would match only the list “(a b c)” and nothing else. However, if we wanted the second and third items of the list to be variable and we needed to use the value of those variable elements elsewhere, we would write the pattern as

```
(a ?x ?y)
```

This pattern would recognize any three element list (or longer) whose first element is “a”. What is more, matching this pattern against the three element list “(a m c)” would not only produce a successful match but would also return a *binding list* of the form “((?x m) (?y c))” representing the fact that the *pattern variables* ?x and ?y are bound to the data values “m” and “c”, respectively.

2.3.3 Pattern Operators

2.3.3.1 The Or Operator

However, patterns need more that just literals and variables to be useful. Among the most important needs are specifications of alternative ways to match a particular data item or sequence of items. For example, suppose that the third element of the previous example can be either “c” or “d”. Now that pattern would be expressed as

```
(a ?x $(por c d))
```

where the $(por …) syntax expresses the pattern alternative via the pattern "or" operator *por*. This pattern will match any three (or longer) element list with "a" in the first position, anything in the second position, and "c" or "d" in the third. The dollar sign syntax identifies the following parenthesized expression as a computable (i.e., non-literal) pattern operator.

2.3.3.2 Bindvar

Now, after such a match, the program would not know whether the third element matched "c" or "d". So, we need a way to bind whichever data value was matched to a pattern variable such as ?y. To achieve this, we could express the pattern as

```
(a ?x $(bindvar ?y $(por c d)))
```

which would match c or d in the third position and then bind the matched value to the pattern variable ?y. Any pattern or subpattern regardless of its size or complexity can be wrapped with a $(bindvar ?variable …) expression and the structure matched by that pattern or subpattern will be bound to the pattern variable. The bindvar operator can be embedded in expressions that are themselves wrapped with a bindvar operation. It is common practice to bind a structure matching an overall pattern to a variable and at the same time bind smaller but important pieces of the overall structure to other variables.

2.3.3.3 Testing Properties

Now, matching often depends more on the **properties** of the data (e.g., type) than it does on the literal value of the data. Therefore, one might like to match any data item of some type (e.g., number) in the third position and then bind the value of the number to some pattern variable such as ?y. If the third position does not contain a number, then the match should fail. The *ptest* operator provides a way to test the properties of data items by allowing the pattern writer to supply the name of a Lisp function that when applied to the data item will succeed if the property is true and fail otherwise. So, now the pattern example would become

```
(a ?x $(bindvar ?y $(ptest numberp)))
```

where "numberp" is the Lisp function that tests a data value to see if it is some form of a number. If this pattern was matched against the list "(a b 47)", the match would succeed and return the binding list "((?x b) (?y 47))".

2.3.3.4 The And Operator and Lisp Escapes

But, what if we need two or more properties to be simultaneously true (say numberp and a value less than 50)? We can use the pattern "and" operator *pand* but there is a catch. When the expression $(ptest numberp) matches the third element, it advances the data pointer so that the pattern matcher is now looking at the data after the third element. That is, once matched, the data has been consumed and subsequent matches will be looking at subsequent data. So, we need the ability to perform an arbitrary function that does no direct matching of the target data string and only serves to determine success or failure of the matching process using previously bound pattern variables. This is accomplished via a very general capability. The pattern language allows evaluation of an arbitrary Lisp expression that will cause the match to fail if it returns nil and to succeed if it returns non-nil. This is the *plisp* operator. So, the new pattern could be expressed as

```
(a ?x $(pand $(bindvar ?y $(ptest numberp))
             $(plisp (<= ?y 50))))
```

However, it could also be expressed in other ways to achieve the same result. Common practice is to match anything and then use plisp or other operators to test for multiple properties all at once. So, an alternative form matching the same form of the data is

```
(a ?x $(pand  ?y  $(plisp (and (numberp ?y) (<= ?y 50)))))
```

The matcher provides a large number of pattern operations (see the Appendix for a complete list) but if the desired operator is not in that list, the matcher is user extensible. The end user can extend the pattern matcher by writing new operators that obey a few well-defined rules of behavior and will be invoked by using the dollar sign ($) notation.

In the following sections, we will introduce a number of other operators in the context of the discussing some of the basic ideas of the pattern matcher.

2.3.4 Invoking the matcher

So far, we have not illustrated how to apply the pattern matcher. The matcher supplies a Lisp macro that will establish a structure for performing a pattern matching operation. This macro has the following form.

(with-matching *pattern data bindinglist body*)

where *pattern* is a pattern like the previous examples, *data* is a Lisp list structure that is to be matched, *bindinglist* is a list of (?variable value) pairs that will be the initial bindings for variables mentioned in the pattern, and *body* is a block of Lisp code that will be executed with the final bindings, which could be nil if the match fails. *Body* is wrapped with a Lisp let scope that includes one Lisp let variable “x” for each pattern variable “?x” in the pattern. In addition, with-matching binds three other Lisp let variables for the scope of *body*. These are **success** which is true if the match succeeded and nil if not, **newbindings** which is the post matching binding list, and **result**, which is a two element list of the form `(,**success** ,**newbindings**).

So, the code that would actually perform a pattern match for the last example against the list ‘(a b 47), and print out the final binding of ?y would look like

```
(with-matching ‘(a ?x $(pand  ?y  $(plisp (and (numberp ?y) (<= ?y 50))))) ‘(a b 47) nil
                    (format t “~%success =  ~A, y = ~A” success  y))
```

This would produce

```
success = t, y = 47
```

on the user’s console window.

2.3.5 Backtracking

Earlier we introduced the concept of pattern alternatives via the por operator. This raises the question of what happens if the first alternative of por is a complex pattern most of which (but not all of which) matches the given data. Perhaps in the course of that partial match, a number of pattern variables are bound to values and then the pattern fails. What happens? The system backtracks to the last choice point (i.e., that point where there are alternative, untested subpatterns) and the matcher restarts the match with the next available alternative. All bindings arising out of the first alternative pattern that partially matched are undone and the matcher restarts with the next choice as if the previous choice never existed. This is called *backtracking*.

2.3.5.1 The Spanto and Pfail Operators

Of course, por is not the only operator that implies alternative choices and therefore, can cause backtracking. For example, the *spanto* operator skips over a number of non-matching items until it finds one that matches its pattern. So, spanto will cause failures until it matches its pattern. There are several other such operators that we will discuss later.

Not only can a pattern fail because a pattern element does not match. It can also fail purposely by use of the *pfail* operator. Pfail is quite useful for constructing searches and we will see some examples of such searches later in the document but for now, we will use pfail to illustrate backtracking. While the pfail operator is used to cause programmer-directed failure, the pattern matcher behaves in exactly the same way when pattern matching fails at some point in the middle of a pattern. It simply backtracks to the last choice point (e.g., a spanto expression), picks the next choice, and proceeds forward with the pattern matching right from where it was when it started matching the failed choice.

In the following example, spanto will skip over elements of the list until it gets to one that matches the literal "g" at which point it binds the intervening items on the list to the variable ?x and allows the matching to proceed to the plisp expression. It prints ?x's value followed by a newline and then returns t (true) thereby indicating to the pattern matcher that the plisp expression succeeded. However, pfail will then cause a failure and the pattern matcher will backtrack to the spanto expression, which will start looking for the next "g".

```
(with-matching   '( $(spanto ?x g) $(plisp (print ?x) (terpri) t) $(pfail))
                 '(a b c g d e f g h i j g g)
                 nil
                 (print newbindings)
                 (print success))
```

The output from this pattern matching (performed at the Lisp top level) is:

```
(A B C)
(A B C G D E F)
(A B C G D E F G H I J)
(A B C G D E F G H I J G)
NIL
NIL
```

The last two NILs are the results of printing the value of newbindings and printing the value of success (i.e., the pattern matching ultimately failed overall, as expected).

Backtracking is vitally important to the pattern representation because it hides much control structure that would obscure the intent of the pattern match. It allows the pattern to be expressed in a declarative form that is more understandable than if the same search were expressed procedurally.

2.3.5.2 Cuts

One does not always want to search exhaustively. Sometimes one wants to search until a particular value is found and then stop the search. Cuts (implemented via the *pcut* operator) are the mechanism whereby this is accomplished.

The pcut operator cuts off any alternative choices thereby accepting the matching results up to the invocation of pcut. This forecloses any opportunities to explore the remaining choices. Using the same example as in the previous section with the pfail operator replaced by a pcut operator will produce the example:

```
(with-matching   '( $(spanto ?x g) $(plisp (print ?x) (terpri) t) $(pcut))
                 '(a b c g d e f g h i j g g)
                 nil
                 (print newbindings)
                 (print success))
```

The output produced by this example is:

```
(A B C)
((?X (A B C)))
T
```

where the first “(A B C)” is value of ?x produced by the print inside of the plisp expression, “((?X (A B C)))” is the result of printing the value of newbindings, and the “T” is the result of printing the value of success.

2.3.5.3 Marking for Cuts

The pcut operator will cut all choices off of the choice stack unless there is an indicated stopping point for a cut. The *pmark* operator supplies that stopping point indication. Let us consider the behavior of the following example and then see how pmark can be used to change that behavior.

```
(with-matching   '(  $(spanto ?x g) ?g $(spanto ?y g) ?g2
                       $(plisp (format t "~%?x,?y=~A,~A" ?x ?y)
                                 (terpri) t) $(pfail) )
                 '(a b c g d e f g h i j g g)
                 nil
                 (print newbindings)
                 (print success))
```

This example will produce the following output:

```
?x,?y=(A B C),(D E F)
?x,?y=(A B C),(D E F G H I J)
?x,?y=(A B C),(D E F G H I J G)
?x,?y=(A B C G D E F),(H I J)
?x,?y=(A B C G D E F),(H I J G)
?x,?y=(A B C G D E F G H I J),NIL
NIL
NIL
```

That is, the first spanto operator finds a “g” which is then matched by and bound to the variable ?g. After this, the second spanto searches for the next “g”, which if found will be bound to ?g2. In the course of the match, the intervening spans will be bound to ?x and ?y, respectively. The pfail operator then causes a series of failures that cause the second spanto to try all of possibilities. When those possibilities are exhausted it causes the pattern to fail back to the first spanto which searches for the next g. In other words, this pattern finds all combinations of the two spans, each terminated by a g.

However, suppose that we want to find all candidates for the first span but only the first candidate for the second span? A cut will not do it because the pattern will find only the first case of both spans, which is not what we want. We want to cut back to the ?g variable but no further. To accomplish this, we can put a mark at that point in the pattern to stop the next cut. After the cut, the next pfail will cause the first spanto to search for the next candidate. So, now the pattern becomes

```
(with-matching   '(  $(spanto ?x g) ?g $(pmark) $(spanto ?y g) ?g2
                       $(plisp (format t "~%?x,?y=~A,~A" ?x ?y)
                                 (terpri) t) $(pcut) $(pfail) )
                 '(a b c g d e f g h i j g g)
                 nil
                 (print newbindings)
                 (print success))
```

which will produce the following output.

```
?x,?y=(A B C),(D E F)
?x,?y=(A B C G D E F),(H I J)
?x,?y=(A B C G D E F G H I J),NIL
```

```
NIL
NIL
```

2.3.6 Other Operators

2.3.6.1 Pnot and None

Sometimes one wants to succeed with a pattern match only if the data does not contain a particular pattern. The *pnot* operator serves this purpose. For example, if we are looking for a two element (or longer) list that does not contain a number in the first position and contains anything in the second position, we could express that as

```
(with-matching '($(pnot $(ptest numberp)) ?any) '(a c) nil (pprint newbindings))
```

which would print out the binding list "((?ANY C))." Of course, the expression

```
(with-matching '($(pnot $(ptest numberp)) ?any) '(47 c) nil (pprint newbindings))
```

would fail and print only NIL.

Of course, there might be a list of things that we wanted to exclude (say a, b, c, and any number) and we could express that as

```
(with-matching   '($(pnot $(por a b c $(ptest numberp))) ?any) '(d c) nil
                 (print success)
                 (pprint newbindings))
```

which would print

```
T
((?ANY C))
```

There is also a shorthand way of expressing the same pattern via the *none* operator

```
(with-matching   '($(none a b c $(ptest numberp)) ?any)  '(d c) nil
                 (print success)
                 (pprint newbindings))
```

which would produce exactly the same result:

```
T
((?ANY C))
```

Of course, if this pattern is used with data such as '(a c) or '(47 c), it will fail just as we would expect.

2.3.6.2 Remain

In addition to matching individual elements of a list, it is often quite useful to match all of the remaining data in the list and bind that list segment to a variable, which can be done with the *remain* operator. For example,

```
(with-matching   '($(spanto ?mainlist (tags $(remain ?taglist))))
                 '(+ a b (tags (foo 10) (bar 2)))
                 nil
                 (pprint newbindings))
```

will print the binding list

```
((?MAINLIST (+ A B)) (?TAGLIST ((FOO 10) (BAR 2))))
```

2.3.6.3 Pmatch

Often one will want to match the same data from several different perspectives or to perform a match that first picks out the broad structures and then uses further matching to look for the details within each of the broad structures. Doing all matching with all subpatterns inlined in one big pattern often leads to a confusing overall pattern. Therefore, the *pmatch* operator provides a way to express submatches. For example, one might want to look for patterns within the remainder of the list bound earlier in the match by Remain. Maybe one wants to extract several different subpatterns within the remainder but does not know which order they occur in on the remainder list. It would be very messy to have to account for the different orderings in one pattern. Therefore, if we want to get the values associated with "foo" and "bar" in a taglist from the previous example, we could write the pattern as:

```
(with-matching '$(pand ($(spanto ?mainlist (tags $(remain ?taglist))))
                        $(pmatch $(spanto ?x (foo ?fooval)) ?taglist)
                        $(pmatch $(spanto ?y (bar ?barval)) ?taglist))
               '(+ a b (tags (foo 10) (bar 2))) nil
               (pprint newbindings)
               (print (+ fooval barval)))
```

This pattern will succeed in finding foo and bar (if they are there) regardless of their order on taglist. This example would print out the binding list

```
((?Y ((FOO 10))) (?BARVAL 2) (?X NIL) (?FOOVAL 10)
(?MAINLIST (+ A B)) (?TAGLIST ((FOO 10) (BAR 2))))
```

followed by the value 12, which is the value of the expression "(+ fooval barval)."

2.3.6.4 Within and Preorderwithin

Spanto allows one to search at the top level of a list but sometimes one wants to look for a structure that may be deeper within a list structure. The matcher provides two operators for this: *within*, which will search for the pattern from the leaves of the structure up, and *preorderwithin*, which will search for the pattern from the top of the tree down. For example,

```
(with-matching '$(within (+ ?x ?y)) '(sqrt (+ (square a) (square b))) nil (pprint newbindings))
```

would print the binding list ((?Y (SQUARE B)) (?X (SQUARE A))).

Now, there could be several possible matches within the data structure. Consider the example,

```
(with-matching '$(pand $(within (+ ?x ?y))
                        $(plisp (format t "~%x = ~A, y = ~A~%" ?x ?y) t)
                        $(pfail))
               '(sqrt (+ (square (+ a c)) (square (+ b d)))) nil
              (pprint newbindings))
```

This would produce the following printout.

```
x = B, y = D
x = A, y = C
x = (SQUARE (+ A C)), y = (SQUARE (+ B D))
```

```
NIL
```

where the final NIL is the value of the final binding list because, in the end, the matcher runs out of answers and returns failure.

Now, we may want to find the answers at the top of the tree first. In that case, we use the preorderwithin operator:

```
(with-matching '$(pand $(preorderwithin (+ ?x ?y))
                       $(plisp (format t "~%x = ~A, y = ~A~%" ?x ?y) t)
                       $(pfail))
               '(sqrt (+ (square (+ a c)) (square (+ b d)))) nil
               (pprint newbindings))
```

which returns the answers in a different order.

```
x = (SQUARE (+ A C)), y = (SQUARE (+ B D))
x = B, y = D
x = A, y = C
NIL
```

2.3.6.5 Psuch and Bindconst

Figure 1 is an example of a typical generator data structure that is built out of CLOS (Common Lisp Object System) objects. Each object captures information about one design artifact. The overall structure and field values that describe the design are built up over time as the generator operates. By code generation time, all of the necessary design information should be computed and assembled into these structures from which the resultant code is generated.

This figure shows an array reference to a two dimensional image array object and its two iterators – Iterator1 and Iterator2. These iterator objects have a dimension field that points to a structure describing the range of the the particular iterator, within which are target program expressions for computing the low and high values of each range. The individual CLOS objects are pointed to by the Lisp variables A, I, and J.

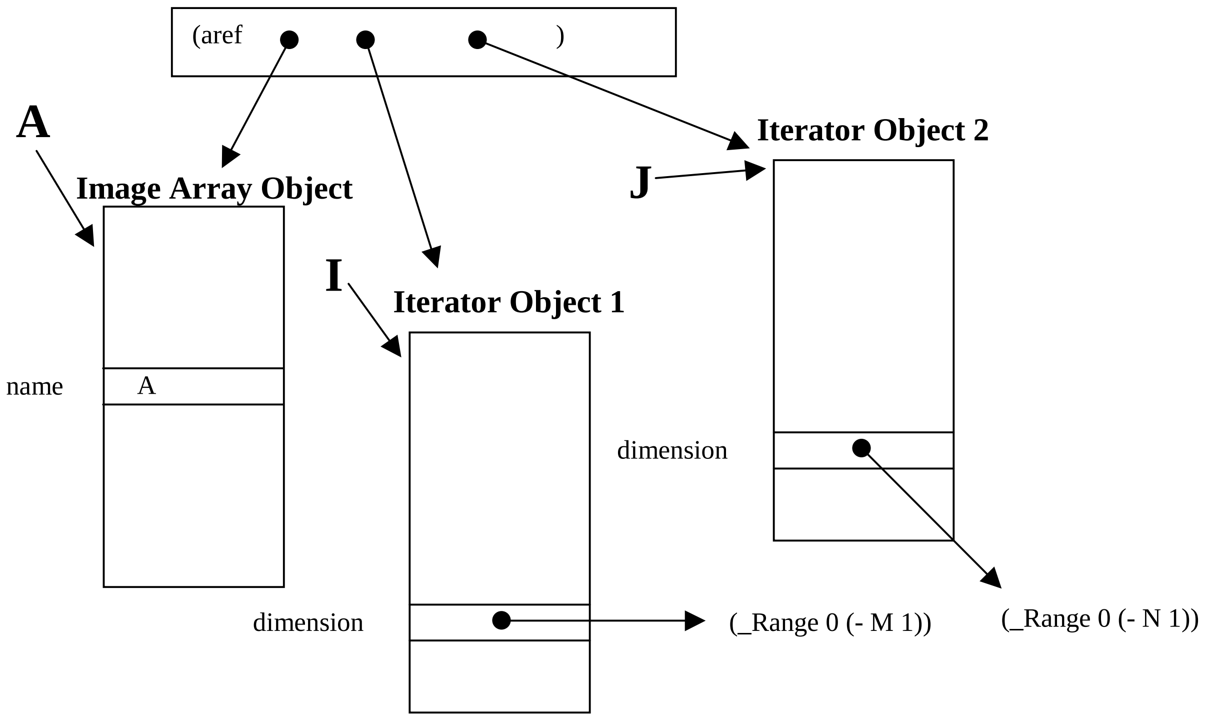

Figure 1
Generator Data Structure

When the generator recognizes an array reference structure, it will typically need to get the high and low values of the iterator or iterators. The *psuch* operator provides a pmatch like construct that is tailored to matching structures within the fields of CLOS objects.

Such patterns often incorporate various optional cases. In this case, for example, the pattern will recognize either one or two dimensional array references, i.e., either of the forms (aref *name iterator1*) or (aref *name iterator1 iterator2*). The transforms that use the bindings produced by this pattern need to know which case was found. The *bindconst* operator serves this need. It provides a mechanism to bind a constant to a variable, which can be used to identify which case was recognized. In the following patterns, the ?acase variable is used to identify which case was recognized.

A typical pattern for matching an array reference is the following:

```
(defconstant[1] ArrayReference '$(por $(pand (aref ?aname #.Iter1AndRange #.Iter2AndRange)
                                                $(bindconst ?acase 2D))
                                        $(pand (aref ?aname #.Iter1AndRange)
                                                $(bindconst ?acase 1D))))
```

and it depends further upon the following subpatterns.

```
(defconstant Iter1AndRange '$(pand ?iter1
                                    $(psuch dimensions ?iter1
```

[1] (defconstant x xvalue) is an assignment of the constant value xvalue to the constant variable x. The *#.Lispvariable* notation allows the value of the Lisp variable *Lispvariable* to be included in the pattern (as shared data) at variable definition time.

```
                                        (_Range ?i1low ?i1high))))
    (defconstant Iter2AndRange `$(pand ?iter2
                                    $(psuch dimensions ?iter2
                                        (_Range ?i2low ?i2high))))
```

Matching the ArrayReference pattern against an array reference structure like that of Figure 1:

```
    (with-matching ArrayReference ‘(aref ,a ,i ,j) nil (pprint newbindings))
```

where A, I, and J are Lisp variables pointing to the CLOS objects, will print out a set of bindings something like

```
    ((?ACASE 2D) (?I2HIGH (- N 1)) (?I2LOW 0)
    (?ITER2 #<ITERATOR #xFF0260>) (?I1HIGH (- M 1)) (?I1LOW 0)
    (?ITER1 #<ITERATOR #xFF0164>) (?ANAME #<IMAGE #xFEFADC>))
```

Values like “#<ITERATOR #xFF0260>” illustrate the way that the Lisp print routine denotes CLOS objects. The same pattern matching result could be accomplished by a combination of plisp and pmatch but it would be much less clean and much more difficult to understand what is going on. The psuch operator is just a simple shorthand for navigating the matcher through CLOS objects and fields.

2.3.6.6 Pat

Often the generator creates patterns dynamically and stores them for later use. As a result, a pattern may be a value of a pattern variable within another pattern. This represents an extra level of indirection and the user needs a way to tell the pattern matcher that the value of a variable is not passive data but rather an active pattern that should be used at its point of occurrence to continue the match. The *pat* operator supplies this facility. The form is

```
    $(pat ?variable)
```

where ?variable is expected to be bound to a pattern. The usage of the pat operator is a bit subtle so the reader is referred to the examples section later in this document for extended examples.

2.3.6.7 Pdeclare

The pat operator introduces a problem. The with-matching macro does not know the names of the pattern variables in the pattern (because they are not available until runtime) and therefore, cannot generate local Lisp let variables and bind the resulting pattern variable values to the local let variables. To overcome this problem, the *pdeclare* operator allows the programmer to mention the names of pattern variables that are anticipated to be in the pattern bound to the pat variable and thereby allow with-matching to set up all of the key local variable scoping and binding structures for the programmer. A typical usage of pdeclare is in a pattern like

```
    `$(pdeclare (?opcase ?fp) $(pat ?fp))
```

which is taken from the extended pat example shown in a later section of this document.

2.3.6.8 Plet

Because of sharing, subpattern elements may be used in contexts where variables in the overall pattern could clash with variables within the subpattern. If these variables are logically local to the subpattern, i.e., they are being used only to communicate information between parts of the subpattern, then they are like local variables in a function and should be invisible outside of the local pattern. The plet operator assures this kind of scoping condition. For example, in the pattern below the variable ?self will be renamed to an anonymous variable by plet and even though there may be several other subpatterns that contain a ?self variable, plet will prevent collisions. Each such ?self will be unique.

```
(defconstant IATemplate-instance
     ‘$(plet    (?self)
               $(pand ?self
                         $(psuch dimensions ?self
                                   ((_Member ?piter (_Range ?plow ?phigh))
                                    (_Member ?qiter (_Range ?qlow ?qhigh)))))))
```

2.3.6.9 Psucceed

Sometimes one wants to cause the last choice point to succeed unconditionally. For example, a structure like the taglist in the following example may be optional. So, one wants the pattern to succeed whether or not a taglist is present. The only difference will be the bindings that one ends up with. The *psucceed* operator will cause the last choice point to succeed. The following example illustrates the psucceed operator.

```
(with-matching    ‘($(por $(spanto ?x (tags $(remain ?taglist))) $(psucceed)))
                  ‘(a b d (tags d e f))
                   nil
                  (pprint newbindings)
                  (print success))
```

The result of the match will be bindings for ?x and ?taglist. ?x will be bound to the list preceding the taglist and ?taglist will be bound to the list of elements just after the atom “tags” on the sublist. This example will print

```
 ((?X (A B D)) (?TAGLIST (D E F)))
T
```

On the other hand, if there is no sublist with a “tags” atom, neither ?x nor ?taglist will get bound but the match will still succeed (i.e., the value of success will be t). The Lisp variables x and taglist in the body of the with-matching macro will have Lisp values of nil.

```
 (with-matching   `($(por $(spanto ?x (tags $(remain ?taglist))) $(psucceed))) `(a b d) nil
                  (pprint newbindings)
                  (print success))
```

This example will print

```
NIL
T
```

2.3.6.10 Rules and Pprove

Sometimes one may need to intersperse some inference with the pattern matching operations. For example, one may want to test for the truth of a condition where the condition may be implied by many combinations of structure alternatives. For example, there is a loop splitting transformation that must do a bit of inference to infer the modified range of a loop iterator from a logical expression specifying the range. For example, the original specification of the range of the loop iterator might have been expressed as

```
(_member ?i (_range 0 (- m 1))
```

and the new intermediate specification produced by the transformation might be

```
(and (_member ?i (_range 0 (- m 1)) (!= ?i 0))
```

or

```
(and (_member ?i (_range 0 (- m 1)) (< ?i 0) )
```

or a variety of other forms. The AOG system must do some inference to reformulate the iterator specification into a form that the code generator can deal with, such as inferring that the specification

```
(and (_member ?i (_range 0 (- m 1)) (!= ?i 0))
```

can be formulated as

```
(_member ?i (_range 1 (- m 1)).
```

That is to say, the transform wants to infer a new range (e.g., 1 to (m – 1)) from the old range (e.g., 0 to (M- 1)) plus some additional constraining clauses such as (!= ?i 0) or (< ?i 0). But of course, one does not want to express all of the possible variations in terms of alternative structures within every different pattern that might need to recognize the variations because we would have to build into each pattern all of the inferable alternatives that one might ever need. This would create hugely complex, highly redundant and highly un-maintainable patterns. A better way would be to state the individual inference rules once each and let the pattern matcher do the combination at inference time. This is how the AOG pattern matcher handles this problem.

One needs the analog of a subroutine, which in the world of logic is a *rule*. A rule in the AOG matcher expresses the logical relationship between the inference goal that the program is trying to achieve (called the *consequent*) and an expression of other goals and matching operations (called the *antecedent*) that describe one way (of possibly many ways) that the consequent can be satisfied or achieved. If there is no antecedent, then the rule will be satisfied if the goal that the program wants to achieve simply matches the consequent of the rule. Rules are defined by the <- macro and have the general form:

(<- *consequent antecedent*)

They are invoked by an expression of the form

$(pprove *goal*)

Rules are roughly analogous to Prolog rules but they are not exactly Prolog rules. In Prolog, all rules and data are conceptually in a single global database. In the AOG matcher, all operations are relative to some local data that is being matched, which is indicated via the data argument of the with-matching macro. As a result, the antecedents of the matcher's rules may be a mixture of pattern matching and inference (i.e., pprove) operations. The pattern matching references the local data and the pprove inference operations reference a separate database of rules.

One can think of the inference activities as being conceptual scaffolding in which to organize the variations in matching. Put another way, the rules are just a way to map from pattern matching on structures into more abstract conceptual facts. For example, if the matcher finds a specific pattern (of possibly combinatorially many pattern variations) in the description of a loop, it may mean that the loop can be restructured in a specific way. In a weak sense, it has inferred the restructuring property of the loop by pattern matching.

The pprove operator is a *pattern directed* operator in the sense that the pattern of the goal determines which rules can be invoked. If the goal matches the pattern of the consequent of a rule, that rule may be invoked. The order of the rules in the rule base determines the order in which rules are tried and that order is determined by their order of definition. The first rule whose consequent matches a pprove's goal will be invoked. If that rule ultimately fails, the next rule with a matching consequent will be invoked and so on.

Consider the two example rules:

```
(<- (append nil ?xs ?xs))
(<- (append (?x . ?xs) ?ys (?x . ?zs)) $(pprove (append ?xs ?ys ?zs)))
```

They define the append function. These rules say that the append relationship among three lists is satisfied when the third list is equal to the second list appended to the first list. These rules allow us to compute the value of any one of the three inputs given the other two. For example, if one wrote[2]

```
(with-matching   '$(with-rules (general)
                         $(pprove (append ?x (30 40) (10 20 30 40)))) nil nil
                 (pprint success)
                 (pprint newbindings)
                 (terpri) ; print newline
                 (format t "~%?x=~A~%" (fullbind* '?x newbindings)))
```

The matcher would match the pattern against the rules and would eventually satisfy the pprove operator when ?x is bound to the list "(10 20)". This example would print something like[3]:

```
T
((?G43697 (30 40)) (?G43685 NIL) (?G43684 (30 40)) (?G43682 20)
 (?G43683 (30 40)) (?G43665 (?G43682 . ?G43685)) (?G43664 (20 30 40))
 (?G43662 10) (?G43663 (30 40)) (?X (?G43662 . ?G43665)))

?x=(10 20)
```

For the moment, we will ignore the role of the with-rules operator and consider that in more detail in the next section.

Notice that the example used the fullbind* routine to express the value of ?x because the particular recursive specification that we chose for append induced extended binding chains made up of partial lists. That is, ?x is immediately bound to the list "(?G43662 . ?G43665)" and the variables in this list are recursively bound to other variables and values. Fullbind* traces down these binding chains to re-express the value of a variable purely in terms of non-variable items. If there is no non-variable item at the end of some chain, it will substitute an uninterned gensym symbol to signal an *undefined* value. Thus, in the end, all variables are eliminated from the value expression, which for this example will produce a value for ?x of "(10 20)". The programmer should be aware that with recursive rule definitions structured like the append definition, some assembly of the resultant values will often be required! Fullbind* does that assembly job.

By contrast, for

```
(with-matching   '$(with-rules (general)
                         $(pprove (append (10 20) ?x  (10 20 30 40)))) nil nil
                 (pprint success)
                 (pprint newbindings)
                 (terpri) ; print newline
                 (format t "~%?x=~A~%" (fullbind* '?x newbindings)))
```

the pprove expression would be satisfied when ?x is bound to (30 40) and no assembly is required. It would print something like:

```
T
((?X (30 40)) (?G43752 ?G43738) (?G43739 (30 40)) (?G43738 ?G43718)
 (?G43740 NIL) (?G43737 20) (?G43719 (20 30 40)) (?G43718 ?X)
 (?G43720 (20)) (?G43717 10))
```

[2] We will define the with-rules operator shortly.
[3] The variables of the form ?G43682 are anonymous (i.e., gensym-ed) variables introduced to avoid variable clashes on recursive iterations of rules and therefore, their names will vary from execution to execution. Hence, I use the description "print something like" to be strictly accurate.

```
?x=(30 40)
```

Notice that in these examples, with-matching's data argument is irrelevant. Pprove statements may or may not cause the invocation of matching operations. In real use, they generally do, eventually. For a more detailed example of the interaction between inference and matching, see the Examples section later in this document.

2.3.6.11 Rulesets and the With-Rules Operator

Some subsets of rules are only relevant to narrow transformation purposes. For example, there is a set of rules -- the *loopcontrol* rules -- that are used to infer simplified forms of loop control specifications. More specifically for example, one rule in the set -- the *fixediterator* rule -- determines that a loop range specification for an iterator i that appears in conjunction with an assertion that i is equal to a constant expression, e.g., (= i 0) or (i= i (n –1)), may be reformulated as the loop body without any iteration structure thereby, converting a loop into a non-loop. This rule (and similar rules dealing with reformulating loops) are only relevant to particular transformations and it would be computationally wasteful if they were considered at other times. Therefore, they are organized into rulesets that can be invoked for specific matching and inference operations.

The general form of a ruleset is

(RuleSet (*rulesetname superset*) … *rules* …)

where *rulesetname* is a Lisp symbol name whose value is a CLOS instance of the Rules class. The *value* slot of the rulesname object contains the list of rules. The *super* slot of rulesname points to another Rules object with rules that are inherited by this Rules object (or nil if this is the root ruleset). Thus, rules are organized into trees with the root object of each tree containing the most general rules (i.e., those potentially applying to all inference processes) and those objects lower in the tree containing rules that apply to more specialized inference processes, e.g., loop control logic.

Example rule sets are:

```
(RuleSet (general nil)

  (<- (member ?x (?x . ?rest)))
  (<- (member ?x (_ . ?rest)) $(pprove (member ?x ?rest)))[4]

  (<- (append nil ?xs ?xs))
  (<- (append (?x . ?xs) ?ys (?x . ?zs)) $(pprove (append ?xs ?ys ?zs)))

  (<- (ordered (?x)))
  (<- (ordered (?x ?y . ?ys)) $(pprove (<= ?x ?y)) $(pprove (ordered (?y . ?ys))))

        …etc…

        )
```

and

```
(RuleSet (loopcontrol nil)

  (<- (fixediterator ?i ?imember ?iequal ?ilow ?ihigh)
     $(pand ($(spanto ?grunge (_suchthat $(remain ?such))))
              $(pmatch ($(spanto ?x $(bindvar ?imember (_member ?i (_range ?ilow ?ihigh)))) )
```

[4] The underscore symbol (_) is the don't care operator. It will match anything.

```
                        ?such)
                $(pmatch ($(spanto ?y $(bindvar ?iequal $(por (= ?i  ?c) (= ?c ?i)))) ) ?such)))
    …etc…

    )
```

To specify the rule set to use for inference, one uses the *with-rules* operator. In the following example, the pprove operator will us the loopcontrol ruleset. The example assumes that the pattern matcher is positioned in a loop structure just above the *_suchthat* clause, which specifies the loop iteration control information. The example

```
    (with-matching   '$(with-rules (loopcontrol)
                            $(pprove (fixediterator ?v ?mem ?eql ?low ?high)))
                     '((_suchthat (_member j (_range 0 (- m 1))) (= j 0))) nil
                     (print success)
                     (pprint newbindings)
                     (terpri) ; print newline
                     (format t "~%?v=~A, ?mem=~A, ?eql=~A, ?low=~A, ?high=~A"
                     v mem eql low high))
```

will produce output like

```
    T
    ((?G45593 ((_MEMBER J (_RANGE 0 (- M 1))))) (?EQL (= J 0))
     (?G45595 0) (?G45600 NIL) (?MEM (_MEMBER J (_RANGE 0 (- M 1))))
     (?HIGH (- M 1)) (?LOW 0) (?V J) (?G45592 NIL)
     (?G45591 ((_MEMBER J (_RANGE 0 (- M 1))) (= J 0))) (?G45596 ?HIGH)
     (?G45597 ?LOW) (?G45594 ?EQL) (?G45599 ?MEM) (?G45598 ?V))

    ?v=J, ?mem=(_MEMBER J (_RANGE 0 (- M 1))), ?eql=(= J 0), ?low=0, ?high=(- M 1)
```

However, this example does not reflect how the loopcontrol rules are used in practice because the transformation would not know which rule is relevant. It would only know which loop variable (e.g., “j”) may engender changes to the loop specification. So, in practice, the transformation provides the loop variable and lets the rules compute which rule applies as in the following example:

```
    (with-matching   '$(with-rules (loopcontrol)
                           $(pprove (?which j ?mem ?eql ?low ?high)))
                     '((_suchthat (_member j (_range 0 (- m 1))) (= j 0))) nil
                     (print success)
                     (pprint newbindings)
                     (terpri) ; print newline
                     (format t "~%?which=~A, ?mem=~A, ?eql=~A, ?low=~A, ?high=~A"
                     which mem eql low high))
```

So, the example invocation supplies the loop control variable “j” and lets the matcher determine that the applicable rule is the “fixediterator” rule. This example produces output of the form:

```
    T
    ((?G19885 ((_MEMBER J (_RANGE 0 (- M 1))))) (?EQL (= J 0))
     (?G19887 0) (?G19892 NIL) (?MEM (_MEMBER J (_RANGE 0 (- M 1))))
     (?HIGH (- M 1)) (?LOW 0) (?G19884 NIL)
     (?G19883 ((_MEMBER J (_RANGE 0 (- M 1))) (= J 0))) (?G19888 ?HIGH)
     (?G19889 ?LOW) (?G19886 ?EQL) (?G19891 ?MEM) (?G19890 J)
     (?WHICH FIXEDITERATOR))
```

```
?which=FIXEDITERATOR, ?mem=(_MEMBER J (_RANGE 0 (- M 1))), ?eql=(= J 0), ?low=0,
?high=(- M 1)
```

Currently, there are five such rules used by the SplitLoopOnCases transform and each one triggers a different loop transformation.

2.3.6.12 Minor operators

There are a number of other minor operators that do not rise to the level of being discussed. The reader is referred to the appendix.

2.4 User extensions

The set of possible operators is open. New operators may be defined via the =defun operator and as long as they follow a few simple rules of operation, they will work within the existing framework. See the implementation section for more details.

2.5 Macro expansions in patterns

One of the reusability objectives of the design of the matcher is to make the Lisp system do as much of the work as possible. As a consequence, a Lisp reader macro is defined to create the pattern operator structures. Thus, "$" is defined as a read macro that will to turn all constructions of the form "$(anyname a b c)" into a two tuple of the form (*pat* *lambdaexpression*) where the lambda expression can be applied to a standard set of arguments and which, in turn, will call the function "=anyname" with the appropriate arguments. This simplifies the pattern matcher's logic. Whenever it sees a *pat* structure, it simply applies the lambda expression, which will do the actual matching work.

For "$(anyname a b c)", the exact form of the resultant structure produced by the $ macro would be

```
(*PAT*
 (LAMBDA (*CONT* DATA BINDINGS)
   (FUNCALL '=ANYNAME *CONT* '(A B C) DATA BINDINGS)))
```

The implementation section will explain the role of the *cont* argument and the relationship between anyname and =anyname.

3 Examples

3.1 Introduction

This section is intended to provide a sense of how patterns are constructed, some tricks and techniques that are useful, and a sense of the most interesting real patterns used by the AOG generator.

3.2 Walking a Tree using Pfail

The *pfail* operator can be used to do useful work. In the following example, the within operator is used to walk over the tree looking for all statements (?stmt) with a tags expression containing the tag "(_CFARG ?cfnum)" and then print the value of each ?cfnum followed by the statement to which it is attached.

```
(with-matching  '$(pand $(within ($(spanto ?stmt (tags))
                                  (tags $(spanto ?y (_CFARG ?cfnum)))))
                        $(PLISP (terpri)
                                (pprint ?cfnum)
                                (format t " is attached to " ) (pprint ?stmt) t)
                        $(pfail))
                '(progn (if (<= x y (tags (_cfarg foo)))
                            (then (print 'hello (tags (_cfarg bar))))
                            (else (print 'goodbye (tags (_cfarg baz))))
```

```
                                        (tags (_cfarg wholeenchilada))))

                                  nil nil)
```

This example will print the report:

```
        BAZ is attached to
         (PRINT 'GOODBYE)

        BAR is attached to
         (PRINT 'HELLO)

        FOO is attached to
        (<= X Y)

        WHOLEENCHILADA is attached to
         (IF (<= X Y (TAGS (_CFARG FOO)))
           (THEN (PRINT 'HELLO (TAGS (_CFARG BAR))))
           (ELSE (PRINT 'GOODBYE (TAGS (_CFARG BAZ)))))
```

This is one way that the generator looks for expressions or statements tagged for some kind of action.

The same effect could be had by omitting the explicit pfail operator and having plisp return nil (i.e., failure) but this would make the behavior more obscure and difficult to understand.

3.3 Lisp Escapes

Practically speaking, not every pattern matching function in the world can or should be defined as a primitive pattern operator. One needs a general escape mechanism from the pattern language that allows arbitrary computations and tests to be performed in the midst of a pattern match. The AOG pattern match provides several such escapes each serving a slightly different set of needs.

Plisp allows an arbitrary set of Lisp code to be executed, which will cause the matching to fail if it returns nil and continue if it returns anything else. However, the plisp operator will NOT advance the data pointer in any case. That is, the plisp escape is used for its (succeed or fail) effect not for moving through the data. For example, the following ImageReference pattern uses the plisp escape to test the value of ?image, to assure that it is an atom and to assure that that the type of that atom is a subtype of "Image."

```
        (defconstant ImageReference '$(pand $(por (leaf ?image) ?image)
                                            $(plisp (and (atom ?image) (IsDSType ?image 'Image)))))
```

Sometimes computing a value for a pattern variable is more easily done in Lisp because a function already exists to perform what may be an extended and complex computation or a computation that is not well suited to a pattern representation. It would be bad design practice to duplicate that computation in the pattern language since that would duplicate the maintenance burden should the underlying design change. For example, one may want to compute a value in a Lisp escape and return that value bound to a pattern variable. The Lisp function "starvarbinding" performs this service. The pattern matcher will add any such binding to the current bindings upon completion of the Lisp escape computation. An example usage is

```
        $(plisp (starvarbinding '?quantifier1 (gettag ?op1 '_Q))
```

In this case, it is easier to use the Lisp gettag function to search the tag list of ?op1 for an attribute value pair of the form "(_Q <value>)" than it would be to perform the same search using pattern matching.

The plisp function is often used to achieve side effects, especially those that save intermediate results of searches for use by later Lisp code.

On the other hand, sometimes one wants the Lisp escape test to advance the data pointer (i.e., move through the data) if the escape test is successful. Ptest is designed to fulfill this purpose. Ptest will call the single parameter Lisp function named in its argument to test the next data element and if it succeeds, it will advance the data pointer. For example,

```
$(ptest numberp)
```

will allow the pattern matching to succeed at that point if the current element being matched is a Lisp number (in any of its forms). The argument of ptest may also be a lambda expression of a single argument.

Finally, the papply operator allows a Lisp function to be applied to several arguments. Like plisp, it does not advance the data pointer.

3.4 Sharing Subpatterns

We have already seen patterns that share common subpatterns, which is common practice in the AOG generator. To avoid having to re-write the subpattern again and again, we use the Lisp #. read macro to allow such sharing. For example, the pattern "$(por (leaf ?image) ?image)" will match an image object as a tree leaf. The "(leaf <item>)" structure is used by the AOG generator to provide a place to hang tags on an atomic item but if there are no tags the item may appear without the leaf structure. The pattern accounts for this variability allowing the variable ?image to be bound to the intended leaf item regardless of which alternative form occurs. The ImageReference pattern uses this leaf pattern to recognize an AST leaf and then adds restrictions to assure that the leaf item is an atom and that it is a subtype of Image.

```
(defconstant ImageReference '$(pand $(por (leaf ?image) ?image)
                                    $(plisp (and (atom ?image) (IsDSType ?image 'Image)))))
```

Image references occur in many contexts, so the ImageReference pattern is a popular component of other patterns. For example, the following code is drawn directly from the program generator.

```
(with-matching  '$(pand #.ImageReference
                        $(psuch dimensions ?image
                                ((_Member ?iiter (_Range ?ilow ?ihigh))
                                 (_Member ?jiter (_Range ?jlow ?jhigh)))))
                        at nil  ….)
```

In this case, the transformation needs to know not only that this is an image reference but it needs to know the dimensions of the image. In some other context, the transformation might need to explore other relationships of the image.

The #. macro operator is used extensively within the AOG generator.

3.5 Patterns in Parameter Lists

Conventional parameter lists are just degenerate patterns. In Lisp, Prolog, ML and other similar languages, parameter lists begin to have a richer structure and more flexibility in their expression. For example, they allow key words, optional parameters, and multi-level list structures. Deconstructors for these parameter lists begin to take on some of the character of a pattern matcher. The AOG pattern matcher carries this idea to its logical conclusion.

Parameter lists for AOG generator components are full-blown pattern expressions that can do arbitrary amounts of computation in the course of parameter binding. What is more, to the degree possible, these parameter patterns are used to check the so-called *enabling conditions* for the components. An enabling condition is an arbitrary condition that must be true for the component (say a transformation or a step within a transformation) to be applicable in a given circumstance. Enabling conditions range from the conventional (e.g., an item must be of a given type) to complex sets of restrictions (e.g., a given field of an object must contain a particular pattern or point to another object meeting some arbitrary set of restrictions).

For example, the second parameter of the Prange and Qrange component definitions seen below, are expected to be an array reference, which is a complex structure whose pattern definitions we have seen earlier but are repeated below.

```
(Defcomponent Prange (s #.ArrayReference ?plow ?phigh ?piter)
   (_Range ?plow ?phigh))

(Defcomponent Qrange (s #.ArrayReference ?qlow ?qhigh ?qiter)
   (_Range ?qlow ?qhigh))
```

The array reference parameter expresses the enabling conditions for these components. Array references may be one dimensional or two dimensional structures (e.g., (aref a i) or (aref a i j)). The index items (e.g., i and j) are expected to be CLOS objects each with a slot named "dimensions" and that slot must contain a range pattern, e.g., (_Range 0 10). The psuch operator allows a pattern match on the value of a named slot of a CLOS object. If any of these conditions are not met, the pattern matching will fail and the component will not apply in the given context.

```
(defconstant Iter1AndRange '$(pand ?iter1
                                   $(psuch dimensions ?iter1
                                           (_Range ?i1low ?i1high))))
(defconstant Iter2AndRange '$(pand ?iter2
                                   $(psuch dimensions ?iter2
                                           (_Range ?i2low ?i2high))))

(defconstant ArrayReference '$(por $(pand (aref ?aname #.Iter1AndRange #.Iter2AndRange)
                                          $(bindconst ?acase 2D))
                                   $(pand (aref ?aname #.Iter1AndRange)
                                          $(bindconst ?acase 1D))))
```

The ArrayReference pattern is a commonly used pattern in the generator.

3.6 Dynamically Created Patterns

A powerful technique often used in the AOG generator is to create patterns dynamically and store them away for later use. This technique is used for objects that represent overloaded operators or operands that have multiple methods. Each such operator or operand object has a "formals" slot that contains a pattern whose job is to discriminate among various choices of operator or method expressions, to perform the appropriate parameter binding, and to assure that any enabling conditions included in the parameter expression are satisfied. Consider the Prange and Qrange components defined for s in the previous section. These can be though of as methods of the object s whose formal parameter lists we will refer to abstractly as *parmpat*. For each such method of s, Defcomponent replaces the formals slot of s with the pattern:

`$(por ,*parmpat* ,(formals s))

In other words, the formals slot pattern just ends up being an "or" tree with each branch being a pattern for one of the possibly many Defcomponent parameter patterns. For example, suppose that we have just used Defcomponent to define Orange shown earlier as:

```
(Defcomponent Orange (s #.ArrayReference ?qlow ?qhigh ?qiter)
        (_Range ?qlow ?qhigh))
```

Immediately after Defcomponent executes, the formals slot of s will contain the following pattern

```
$(POR
    ($(PAND QRANGE ?OP)       ;; Pattern for "QRange" operator

     $(PLET  (?SELF)                 ;; Pattern for CLOS object, e.g., s
            $(PAND  ?SELF
                    $(PSUCH DIMENSIONS ?SELF
                            ((_MEMBER ?PITER (_RANGE ?PLOW ?PHIGH))
                             (_MEMBER ?QITER (_RANGE ?QLOW QHIGH))))))

     $(POR $(PAND                  ;; Pattern for 1 or 2 dimensional array reference (AREF name …)
              (AREF ?ANAME
                    $(PAND ?ITER1
                           $(PSUCH DIMENSIONS ?ITER1 (_RANGE ?I1LOW ?I1HIGH)))
                    $(PAND  ?ITER2
                           $(PSUCH DIMENSIONS ?ITER2 (_RANGE ?I2LOW ?I2HIGH))))
              $(BINDCONST ?ACASE 2D))
             $(PAND
                (AREF ?ANAME
                    $(PAND ?ITER1
                           $(PSUCH DIMENSIONS ?ITER1 (_RANGE ?I1LOW ?I1HIGH))))
              $(BINDCONST ?ACASE 1D)))

     ?QLOW ?QHIGH ?QITER)   ;; Pattern for the last three parameters of QRange
```
… remainder of pattern containing subpatterns for previously defined methods…)

The body of the component in the case of the Qrange component) will be stored in a slot of s whose name is the method name (e.g., the "Qrange" slot in the case of the Qrange component). Hence, (QRange s) will contain the value

```
        (_RANGE ?QLOW ?QHIGH)
```

In contrast to operands, overloaded **operators** do not supply an easily identified method name to help distinguish among their various cases. For example, one might overload the "+" operator (i.e., "+op") to operate on matrices as well as numbers. So, we might have separate definitions for it named +NumberXNumber and +MatrixXMatrix. How does the AOG system map from the patterns that recognize these separate forms to the specific definitions to be used in the translation process? Basically, Defcomponent doctors the pattern to produce the information necessary to distinguish which case was recognized. So, for an operator component defined as

(Defcomponent *nameofbodyslot patternofparameters body)*

defcomponent replaces the fomals slot of the operator object with

\`$(por $(pand ,*patternofparameters* $(bindconst ?opcase ,*nameofbodyslot*)) ,(formals +op))

In other words, it "ands" a subpattern to the parameter pattern to set the pattern variable ?opcase to a constant value if the match gets to this point in the pattern. The value of the variable ?opcase will serve to identify which parameter list matched the actual parameters as well as identify the name of the slot (e.g., +NumberXNumber or +MatrixXMatrix) containing the body of the component. The following section illustrates how such patterns are used in the generator.

3.7 Using Dynamically Created Patterns

The following code illustrates how the AOG generator uses dynamically created components to implement inlining of components. This example assumes that the Lisp variable op points to a CLOS object representing an overloaded operator. First, the example retrieves the pattern from the formals slot of op (see example in previous section) storing it in fparms. Make-change-list analyses the pattern and generates a list of all pattern variables each paired with a unique anonymous variable (i.e., gensym-ed variable) that will replace it. The only variables that are not replaced with anonymous variables are those mentioned in the exception list of the call to make-change-list. In the example, "?opcase" will not be replaced. ?opcase is a global variable that after matching will be bound to the name of the slot containing the body of the correct operator definition. Next, the variables in the pattern (except ?opcase) are replaced with the anonymous variables. Later, when the body is retrieved, it too will have to have its variables replaced by the corresponding anonymous variables. The pattern (in fparm) is then bound to a global variable ?fp which is used in the matching operation that follows. The "pat" pattern operator is designed specifically to provide the necessary level of pattern indirection. The "pdeclare" operator is there to inform the with-matching macro that opcase and fp are global variables that will be needed by the Lisp code in the body of the with-matching. With-matching will create a Lisp let for them and bind the appropriate ?variable values to them before the body of with-matching is executed.

```
(setf fparms (formals op))
(setf renamelist (make-change-list fparms '(opcase)))
(setf fparms (applysubstitution fparms renamelist))
(setf bindingswithformals (append `((?fp ,fparms)) bindings))
(with-matching `$(pdeclare (?opcase ?fp) $(pat ?fp))
               at bindingswithformals
               ….
               (setf body (applysubstitution (slot-value op opcase) renamelist))
               (setf newbody  (applysubstitution body newbindings))
               ….
)
```

After the pattern match is successfully completed, the body is retrieved from the opcase slot of the op object and the appropriate substitutions made in the body. Newbody is now inlinable.

3.8 Dynamically Created Patterns for GenVoca Components

GenVoca is a component based reuse strategy in which the components are layers in a Layer-Of-Abstraction (LOA) model of program construction. Each layer encapsulates a pure abstraction or pure feature (e.g., a doubly linked list property or a synchronization property). Assembly of selected instances of the layers causes the generation of custom large gained components, which are then further assembled by conventional composition methods (e.g., method calls) into the target application.

Each GenVoca component is an aggregation of classes and methods that encapsulate one feature of the target large grained component that is to be generated. The programmer specifies the combination of features for the target by writing a *type equation* that lists which GenVoca components are to be composed and the order of their composition. Fundamentally, this composition is a process of in-lining the method bodies of lower level components into the bodies of higher level components. The type equation introduces a minor complication to the strategy described in the previous section for using dynamically created patterns to produce inline-able component bodies. I will describe how to accommodate that complication.

An example of the right hand side of a type equation (adapted to a Lisp representation) for a deque, might look like

```
(list deqsync deque2c dlist avail heap transient)
```

Conceptually, this type equation represents the LOA model shown in Figure 2.

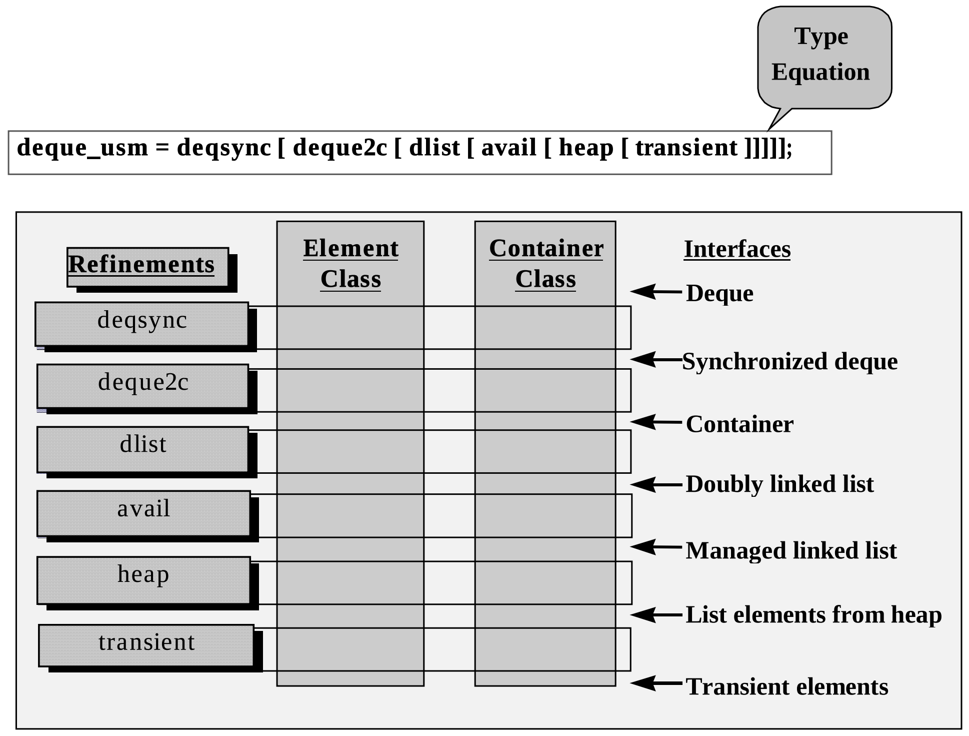

Figure 2
GenVoca definition of a deque

This describes an unbounded queue implemented as a doubly linked list whose data sharing is controlled by semaphores, whose memory management is the responsibility of the programmer (rather than of the system as in a system with garbage collection) and whose memory is transient. The individual components in the type definition build up the instance variables and methods of two classes slice by slice. Each slice may add new instance variables, new algorithmic details to the methods, or both. The resultant data structures of this example are shown graphically in Figure 3. So, how do we map GenVoca components into that result?

The type equation guides the customized assembly. The first component (deqsync) maps a deque abstraction into a synchronized deque. This introduces a semaphore instance variable named sem within the deque class. The second component (deque2c) implements the deque class as a container class. The third component (dlist) implements the container class as a doubly linked list class. It extends the instance variables of the container class to include a head and tail for the list of items in the container. It extends the list of

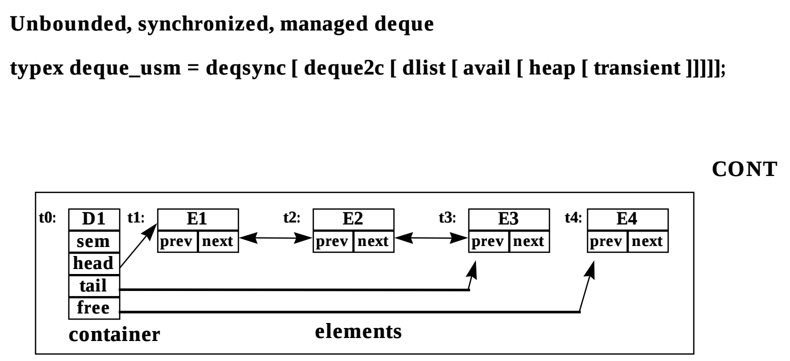
Unbounded, synchronized, managed deque
typex deque_usm = deqsync [deque2c [dlist [avail [heap [transient]]]]];
CONT
t0:
D1
sem
head
tail
free
t1:
E1
prev
next
t2:
E2
prev
next
t3:
E3
prev
next
t4:
E4
prev
next
container
elements

Figure 3
Resultant instance variables and data structures for the deque

instance variables of the elements class with prev and next thereby transforming each element into an entry of a doubly linked list. The avail component adds the free instance variable to the container class with which to keep a list of free storage to be managed by the application. The heap component determines that the memory used for the container's free list comes from a heap. The transient component terminates the type expression and indicates that the container is transient (as opposed to persistent) and therefore, lives only during the execution lifetime of the target application program.

While it is beyond the scope of this document to fully examine examples of GenVoca's large grain components, we can represent a vertical slice showing the generation of a single method of deque -- the addfront method -- and examine how the strategy of dynamic patterns can be modestly adapted to support the composition of GenVoca components. This derivation has been described elsewhere. (See Batory *et al.* 1993 and Biggerstaff 1998b.) We will represent each GenVoca component as a CLOS object, e.g., deque will represent the class for the GenVoca deque component. The other GenVoca components, deqsync, deque2c, etc. will be similarly represented by CLOS objects with names corresponding to the GenVoca component names. Methods and classes within GenVoca components will be represented by AOG-style components[5] (i.e., components defined by Defcomponent as used in previous examples). Each such Defcomponent-base method (e.g., the addfront method of deq2sync) will be broken into two parts: a pattern (i.e., the Defcomponent's parameter list) and a body (i.e., the body part of the Defcomponent definition). The pattern part will be or-ed into the formals slot of the relevant CLOS object (e.g., deq2sync) and the body will be stored in a slot named after the method (e.g., addfront) just as described in the previous discussion on dynamically created patterns (see section 3.6). To keep the example manageably small, we will mostly focus upon a single method within each of the components and demonstrate how to use the AOG pattern-matching engine to accomplish a GenVoca like composition and generation process. Basically, we will use the type equation to guide the choice of individual lower level components (i.e., those lower in the levels-of-abstraction model) whose bodies will be inlined in the bodies of the higher level components.

Let us start (at the top of the LOA model) by defining the addfront method for a deque component, which looks like:

```
(defcomponent addfront (deque  ?d ?e)
  (addfront ?belowme ?d ?e))
```

This is little more than an empty shell whose parameters include the GenVoca component type (i.e., deque) as the first parameter followed by two data parameters, ?d and ?e. These last two parameters will bind, respectively, to an (as yet to be defined) instance of a deque data structure and an instance of an (as yet to be defined) element that is to be added onto the front of the deque. Notice that there is a pattern variable ?belowme in the body of the method definition. This variable is there so that the generator can bind one of the GenVoca components from the type equation list to ?belowme and thereby indicate a pattern that hopefully will match an AOG Defcomponent method (e.g., addfront) in some lower level GenVoca component (e.g., deqsync). Indeed, the very next GenVoca component (i.e., deqsync) contains an addfront method whose parameter list will match the body of deque's addfront if deqsync is bound to ?belowme. The definition of deqsync's addfront method is:

```
 (defcomponent addfront (deqsync ?d ?e)
   (declarelocals ?belowme ?type ?name)
   (wait (sem ?d))
   (addfront ?belowme ?d ?e)
   (signal (sem ?d)))
```

[5] Since **component** is an unavoidably overload word in this context, we will take some care to distinguish GenVoca components (which are horizontal slices of classes and methods) from AO components (which are macro-like methods built on CLOS objects).

This definition contributes a pattern (i.e., (declarelocals ?belowme ?type ?name)) that will eventually declare a local variable[6] to hold the new instance of the element to be added. This local variable cannot really be determined until we know the details of storage management, which will be defined in some lower layer of the LOA model. In addition to the newly declared local variable, this GenVoca component contributes a synchronization wrapper that waits on a newly introduced field of ?d (i.e., the sem or semaphore field). The sem field introduction falls outside of what we can show in this example but its introduction is very much in the spirit of these steps that are filling out addfront.

The next layer down (deque2c) does not really contribute much new code for this method except that it defines deque2c's addfront method in terms of the insertfront method in some component below deque2c. In the deque2c component, the method insertfront reflects the fact that the definition is dealing with a vanilla container component rather than a deque.

```
 (defcomponent addfront (deque2c  ?d ?e)
    (insertfront ?belowme ?d ?e))
```

The next layer (i.e., dlist) causes the vanilla container of the deque2c layer to be implemented as a doubly-linked list. This requires the introduction of a local variable g whose definition will have to be plugged in above the current node in the tree. The variable g is created by the declarelocals method shown below and it will be passed back up the tree to the point in the addfront method of deqsync that is reserved for it. How this actually happens is beyond the scope of this example but suffice it to say that the plisp operators in the parameter lists of both the declarelocals and insertfront methods are creating the structures by which this can be accomplished. Notice also that the plisp operator in the insertfront method is causing a new CLOS object of type element and print name "g" to be created for each application of the insertfront method. That object will be bound to the plisp let variable g for the lifetime of the plisp operator. It will also be bound to the pattern variable ?g. It ends up in the target program as the target program variable "g".

```
(defcomponent declarelocals (dlist  ?e ?g $(plisp (starvarbinding '?element element)))
  (declare ?element (deref ?g)))


(defcomponent insertfront (dlist  ?d ?e
                                  $(plisp (let (g) (starvarbinding '?g
                                                     (progn (dsdeclare element g)
                                                            (push   `(declarelocals ,dlist ,element ,g)
                                                                          uplink)
                                                             g)))))
  (insertfront ?belowme ?d ?e ?g)
  (= (prev ?g) NULL)
  (if (!= (= (next ?g) (head ?d)) NULL)
       (then (= (prev (next ?g)) ?g)))
  (if (== (head ?d) NULL)
      (then (= (tail ?d) ?g)))
  (= (head ?d) ?g))
```

The insertfront method of dlist contributes the code that hooks the newly minted element g onto the front of the deque (i.e., the code appearing just after the call to the next lower insertfront method). Simultaneously (but external to this example slice), the new fields of next and prev are added to element, and the new fields head and tail are added to the container class that implements the deque.

The next lower component, avail, introduces the free storage list to be managed by the application and adds the code to get free storage blocks from the free list if there are any available blocks. Simultaneously, but

[6] This design is not very general but it is a concession to make the example simple. The general solution involves active transformations that search up the tree for the proper scope to place the declarations. This solution is beyond the scope of this example.

not shown in the example, the avail component also introduces the free field as an instance variable of the container class and the next_free field as an instance variable of the element class. free will point to the free storage list and the next_free field will serve as the pointer chain field of the free list.

```
(defcomponent insertfront (avail  ?d ?e ?g)
  (if (free ?d)
      (then (= ?g (free ?d))
            (= (free ?d) (next_free ?g))
            (= (data ?g) ?e))
      (else (insertfront ?belowme ?d ?e ?g))))
```

The heap component introduces code that indicates that the elements will be allocated from some as yet to be determined heap via the abstract routine “alloc”.

```
(defcomponent insertfront (heap  ?d ?e ?g)
  (= ?g (alloc ?belowme (sizeof ?e)))
  (= (data ?g) ?e))
```

The transient layer determines that the heap will be the C heap and therefore, the components will be transient. They will only live for the execution lifetime of the application program. This is accomplished by mapping the call to the abstract routine alloc into a call to the concrete C routine malloc.

```
(defcomponent alloc (transient ?e)
  (malloc ?e))
```

There is a GenVoca element method whose definition determines the form of local declarations for elements.

```
 (defcomponent declarelocals (element ?type ?declaree)
   (declare ?type ?declaree))
```

Now the heart of the mechanism that will compose these components is a slight variation of the strategy of the previous section. It looks like

```
(setf fparms (formals component))
(setf renamelist (make-change-list fparms '(belowme op)))
(setf fparms (applysubstitution fparms renamelist))
(setf bindingswithformals (append `((?fp ,fparms) (?belowme ,component)) bindings))
(with-matching `$(pdeclare (?belowme ?fp ?op) $(pat ?fp))
                         body bindingswithformals
                        (if success
                            (progn
                                (setf newbody (applysubstitution (slot-value component op)  renamelist))
                                (setf subbindings   (removebindings '?fp
                                                       (removebindings '?belowme
                                                          (removebindings '?op newbindings))))
                                (setf newbody (applysubstitution newbody subbindings)))))

….)))
```

In this case, we use the formals slot of the current GenVoca component from the type equation as the pattern to be matched and the Lisp variable body (which presumably contains an AST subtree) as the data to be matched. Here the current component from the type equation will be the value of the Lisp variable component. As an example, the Lisp variable component might have the value “dlist,” in which case dlist will be bound to ?belowme for the matching operation. So, if component is the GenVoca dlist component and body is the AST subtree

```
        (insertfront ?belowme d e)
```

where d and e are CLOS instances of deque and element, then the pattern in the formals slot will match with ?op bound to insertfront, ?d bound to d and ?e bound to e. Thus, the insertfront slot of dlist will contain the defcomponent body shown earlier in the example. That body will be used (suitably instantiated) to replace the current body structure in the AST. The defcomponent variables in the parameter pattern will be renamed to synchronize with the renamed parameters in the parameter pattern, just like in the previous section. Then the newbindings (with bindings for ?fp, ?belowme, and ?op removed) are substituted in the body. Removing ?belowme, etc. from the bindings allows any ?belowme variables in the body to refer to components that are lower in the type equation than the current one.

On the other hand, if there is no match, the code on the fail path (not shown) will leave the body expression unchanged (e.g., it will leave the ?belowme at the current level unbound) so that some lower level GenVoca component can match and replace body's value in the AST.

The code shown is the heart of the algorithm needed to perform GenVoca compositions. But the algorithm in which this code occurs must accomplish several other objectives.

- It must walk over the whole AST executing the code segment shown above (i.e., substituting bodies for method calls).
- It must assure that declarations generated in lower layers but which must migrate to higher level scopes in the final AST will get plugged in at the right place.
- It must assure that any such plugged in declarations are allowed any further GenVoca composition and reduction that might be required by their definitions.

Applying the overall algorithm to "(addfront ?belowme d e)" using the type equation and the componentry described in the example will produce a body for the deque's addfront method that has the AST form of:

```
(PROGN
        (DECLARE ELEMENT (DEREF G))
        (WAIT (SEM D))
        (PROGN
                 (IF (FREE D)
                        (THEN  (= G (FREE D))
                               (= (FREE D) (NEXT_FREE G))
                               (= (DATA G) E))
                        (ELSE (PROGN
                               (= G (MALLOC (SIZEOF E)))
                               (= (DATA G) E))))
                (= (PREV G) NULL)
                (IF (!= (= (NEXT G) (HEAD D)) NULL)
                        (THEN (=  (PREV (NEXT G)) G)))
                (IF (== (HEAD D) NULL)
                        (THEN  (= (TAIL D) G)))
                               (= (HEAD D) G))
                (SIGNAL (SEM D)))
```

From this AST form, it is easy to generate code for common programming languages such as C, C++, or Java.

3.9 Finding Two Ifs with a Common Condition

This example illustrates the degree of recognition leverage provided by the pattern abstractions in the face of large-scale patterns that have a high degree of variability. But first a little motivation for the example is needed.

The AOG generator uses *tag-driven* transformations, which means that tags (essentially property lists) attached to elements of the AST trigger the transformations that they refer to. Some of these tag-driven transformations (i.e., those wrapped with an _On expression of the form "(_On *event* *transform*)") are only triggered when the optimization event specified by *event* occurs. For example, a tag like

```
(_On CFWrapUp (_MergeCommonCondition))
```

might be attached to an if-then statement. When the AOG generator posts the named event[7] "CFWrapUp", the generator will schedule the transformation "_MergeCommonCondition" to be run with the if-then statement that contains the tag bound to the variable ?me. _ MergeCommonCondition will look for two contiguous if-then statements that have a common conditional expression. One of those if-thens will have the value of ?me.

This situation represents a large-scale, complex pattern with lots of variation. For example, the second if-then statement (represented by the pattern "ifthen2" below) may or may not have an else clause and each clause within it may or may not have a tag list. Furthermore, the overall pattern covering both if-thens (which is represented by the pattern "pairofifs") has to deal with the two ifs in either order (i.e., ?me last or ?me first). It also has to climb up the tree to find the place in the AST above both ifs. This level of variability is typical of large scale transformations and would be rather difficult to deal with without the kind of abstractions provided by the pattern matcher.

The variability of having a tag list or not is handled by the subpatterns of the form

$(por (tags $(remain *?clausetaglist*)) $(psucceed))

which will succeed with the whole tag list bound to some ?clausetaglist (e.g., ?thentaglist2) if there is a taglist for the particular clause. If there is no taglist, the por just succeeds, which will result in a value of nil for the specific ?clausetaglist.

Similarly, the two different formats of the if–thens are handled by two separate cases and the one that succeeds will bind ?case2 to a literal indicating which case it is. That is, it will bind ?case2 to "ifte" if there is an else clause and to "ift" if there is not.

```
(defparameter ifthen2
        '$(por (if ?mycond
                  (then ?thenexpr2 $(por (tags $(remain ?thentaglist2)) $(psucceed)))
                  (else ?elseexpr2 $(por (tags $(remain ?elsetaglist2)) $(psucceed)))
                  $(por (tags $(remain ?iftaglist2)) $(psucceed))
                  $(bindconst ?case2 ifte))
               (if ?mycond (then ?thenexpr2 $(por (tags $(remain ?thentaglist2)) $(psucceed)))
                  $(por (tags $(remain ?iftaglist2)) $(psucceed))
                  $(bindconst ?case2 ift))))
```

The pairofifs pattern is a bit subtle. It spans up to ?me (but not yet passing over ?me within the main match) and then uses the remain operator to bind the point in the AST list just above ?me so that the following match (i.e., the pmatch expression) can look for both ifs. The pattern then does a separate, more detailed match to see if the pattern ifthen2 immediately follows ?me and if so, binds it to ?otherif. The second case looks for ?me and ?otherif in the reverse order.

```
(defparameter pairofifs
        '$(por $(pand ($(spanto ?pre ?me) $(remain ?abovepair))
                      $(pmatch (?me $(bindvar ?otherif #.ifthen2)) ?abovepair))
               $(pand ($(spanto ?pre $(bindvar ?otherif #.ifthen2)) $(remain ?abovepair))
```

[7] Optimization events can be implicit (e.g., substitution of an expression) or explicit (i.e., posted) events.

```
$(pmatch (?otherif ?me) ?abovepair))))
```

This example illustrates the compactness and utility of the pattern abstractions in the face of large-scale patterns that must cover highly varied sets of cases. This large-scale pattern expressed in low-level code would represent an onerous programming job and would be hard to maintain in the face of change.

3.10 Finding the Target of a Data Flow

This example illustrates a pattern that is dealing with a complex pattern of relationships among program parts. But first, let us motivate the example and explain the contextual assumptions. Sometimes an optimizing transformation will break up a functional expression by 1) removing a subexpression, 2) generating an out-of-expression assignment of the functional subexpression to a temporary variable, and 3) replacing the removed subexpression in the original expression by the temporary variable name. This transformation introduces a new data flow dependency, which the optimizing transformation is responsible to explicitly indicate using tags. The transformation does this by introducing a "_Flows" tag on the temporary variable on the left-hand side of the assignment thereby indicating the source of the flow. It also introduces a "(_requires *listoftagnames*)" on the statement of the original expression thereby indicating the sink of the flow. Later transformations look for such patterns to determine whether or not they can perform their intended transformation. So, an expression like

B = (sqrt ((conv A S)^2 + (conv A SP)^2)

could get converted into something like

T1 = (conv A S)^2;
T2 = (conv A SP)^2;
B = (sqrt ((T1 * T1) + (T2 * T2));

in order to allow the *reduction in strength optimization*[8] on the square operations. In addition, tags would get added to indicate the source and sink of the introduced data flows plus a tag "(_On MigrationOfMe (_CheckFoldFlows))" that anticipates the opportunity for refolding the temporary variables should their values ever get reduced to a constant. So, the final form with tags is:

T1 (tags (_Flows) (_On MigrationOfMe (_CheckFoldFlows))) = (conv A S)^2;
T2 (tags (_Flows) (_On MigrationOfMe (_CheckFoldFlows))) = (conv A SP)^2;
B = (sqrt ((T1 * T1) + (T2 * T2)) (tags (_Requires (T1 T2)));

The definitions of the subexpressions (conv A S)^2 and (conv A SP)^2 will eventually get inlined, reorganized and simplified by the generator. Eventually, that manipulation will trigger the _CheckFoldFlows transformation. _CheckFoldFlows is looking for opportunities to perform constant folding via refolding data flows from the temporary variables in which one or more the right hand sides of the assignments have been reduced to a constant. However, it is not so easy to verify the enabling conditions because the data flow sources may have been duplicated and moved into various newly created program blocks. To verify the enabling conditions, the transformation has to find the sink of the data flow (i.e., the _Requires tag) and then find all other data flow sources in order to determine if the data flow pattern can be transformed.

The following pattern is used to find the sink starting from one instance of a temporary variable as a data flow source. The instance of the flow source (the current position of the generator in the AST) is initially bound to ?flowsource.

[8] A *reduction in strength* optimization replaces an operator that is expensive to compute like the "power" operator with a lower cost operator like the "multiply" operator for special cases of the power operator. In the case seen here, power can be replaced by multiply because the value of the exponent is "2".

```
(let ((binds ‘((?flowsource ,item))) (root (eval toroot)))
  (with-matching
   '$(preorderwithin
          ($(spanto ?x
                $(bindvar ?stmt
                   ($(spanto ?y
                         (tags $(spanto ?z
                                  $(pand (_requires $(remain ?rlist))
                                          $(plisp (member ?flowsource (car ?rlist))))
                                                ))))))))

   root binds
   .....)
```

This pattern does a preorder traversal of the expression tree searching from the root down using the operator preorderwithin. Basically, it is looking for a statement via the subpattern

```
…($(spanto ?x
          $(bindvar ?stmt  …
```

with tags

```
…($(spanto ?y
          (tags …
```

such that on that taglist is a _requires tag whose list contains the ?flowsource variable instance (e.g., T1)

```
…$(spanto ?z
          $(pand  (_requires $(remain ?rlist))
                  $(plisp (member ?flowsource (car ?rlist))))
```

The bindings preserve pointers to the statement (?stmt) and the list of flow source names (?rlist). The other variables ?x, ?y, and ?z are not needed in the later computations.

3.11 Finding All Data Flow Sources

Next, the transformation needs to find ALL data flow sources for the specific sink list, that is, all items with “_Flows” tags that are listed in the _requires list. Again the pattern does a preorder traversal of the expression tree looking for any leaf that has a _flows tag. But additionally, it has to check to make sure that the flow source found (i.e., ?rawitem) is in fact one that is on the requires list (?rlistx) and not some other data flow. If all of these conditions are met, it stores a pointer to the statement containing the flow source on the list stored in the global variable tflowstmts. Then it calls pfail to backtrack to look for the next flow source instance.

```
(let ((binds `((?rlistx ,rlist))))
   (setf gblocklist (setf tflowstmts (setf tflows (setf tflowstructs nil))))
   (with-matching
    '$(pand
      $(preorderwithin
        $(bindvar ?stmt
           ( $(pand $(spanto ?x ?item)
                 $(pmatch (leaf ?rawitem (tags $(spanto ?z (_flows)))) ?item)
                 $(plisp (if (member ?rawitem ?rlistx)
                          (progn  (setf tflows (cons ?rawitem tflows))
                                  (setf tflowstructs (cons ?item tflowstructs))
                                   t) nil))))))
      $(plisp (if (member ?rawitem ?rlistx)
                  (progn  (setf tflowstmts (cons ?stmt tflowstmts)))) t)
      $(pfail))
   root nil binds  ....)
```

3.12 Finding a Loop Containing Me

Another common search requirement of tag driven transformations is to look up the tree for some pattern. MatchUpward is a macro that steps up the expression tree a level at a time looking for a pattern. In this example, it starts at "at" and will proceed to the root of the tree ("root") if necessary. The initial binding of the Lisp list "vbls" to the pattern variable ?loopvbls allows the pattern to make sure that it has found the correct loop by requiring that the intersection of the specified variables (i.e., the specvbls) and the loop's control variables is not null.

```
(MatchUpward '$(bindvar ?stmt
                        $(pand   $(por ( $(por = setf setq)
                                         $(spanto ?x
                                            $(preorderwithin
                                               (_sum ?loopvbls
                                                     $(spanto ?y
                                                           $(por ?me
                                                                $(within ?me)))))))
                                       ($(por loop  _forall ) ?loopvbls
                                               $(spanto ?x $(por ?me $(within ?me)))))
                                 $(plisp (intersection ?specvbls ?loopvbls :test #'equal))))
                  at root `((?specvbls  ,vbls)))
```

In this pattern, ?me is bound to the current statement and the expected pattern is something like

(setf *somevariable* (_sum ?loopvbls ….. ?me …..))

or of a form like

(_forall ?loopvbls ….. ?me …..)

where _sum and _forall are internal AST operators for looping constructs. It also requires that the variables in the ?specvbl list have at least one element in common with the loop's variables. The pattern also allows different kinds of assignments (e.g., setq or =) as well as different kinds of loops.

3.13 Patterns used in Defining Transforms

The AOG generator organizes pattern directed transformations two ways: 1) by transformation stage to which it applies and 2) by object oriented class to which it belongs. For example, the "fusion3" transformation stage generates code from abstractions on the AST.

Deftransform is a macro used to define transformations. It has the general form:

(Deftransform *transformname transformstage classoftransform parameterpattern*
    body)

Deftransform uses the trick of dynamically creating a pattern to discriminate the particular transformation to be called. For the *transformstage* stage (e.g., the "fusion3" stage), the generator stores the formal *parameterpatterns* in the *transformstage* slot (e.g., the fusion3 slot) of the *classoftransform* object. It stores the body of the transformation in the *transformname* slot of *classoftransform*. Recall that the Defcomponent macro handles the parameter pattern and component body in a similar manner, storing them, respectively, in the "formals" slot and the *componentname* slot of operator or operand class with which the component is associated.

Exactly which subgroup of transformations are enabled (i.e., *classoftransform*) at a given subtree in the AST tree depends on the pattern of that subtree. Often the operator at the subtree is the main determiner of which class of transforms is enabled. For example, “dsopertors” is the class of all domain specific operators. Other more specialized classes, capture behaviors particular to subgroups of operators. For example, the “parallelops” class is the more specialized class of all operators that will compile into a certain kind of parallel data flow pattern. The GenerateLoopStructure transformation of the fusion3 stage belongs to the general class dsoperators and it is defined as follows.

```
(Deftransform GenerateLoopStructure fusion3 dsoperators
  $(pand #.QuantifiedExpr1
         $(psuch dimensions ?1iter1 (_Range ?low1 ?high1))
         $(por $(plisp (equal ?acase1 '1D))
               $(psuch dimensions ?1iter2 (_Range ?low2 ?high2))))
        …body…)
```

The parameter pattern of GenerateLoopStructure depends on the additional patterns:

```
(defconstant QuantArray1 '$(por $(pand (aref ?aname1 ?1iter1 ?1iter2)
                                      $(bindconst ?acase1 2D))
                                $(pand (aref ?aname1 ?1iter1)
                                      $(bindconst ?acase1 1D))))
(defconstant QuantifiedExpr1
   '$(pand ?op1 $(plisp (starvarbinding '?quantifier1 (gettag ?op1 '_Q)))
          $(pmatch ($(spanto ?x1 _forall) _forall ?vbl1 #.QuantArray1)
                   ?quantifier1)))
```

The parameter pattern does a lot of work. It is processing an expression that looks like

(… (tags … (_forall *variable-expression* (aref *arrayname iterator1 iterator2*) …) …)

where *iterator2* is optional. The QuantifiedExpr1 gets the tag value (via “gettag”) associated with the attribute “_Q” and binds it to ?quantifier. It then uses pmatch to analyze the structure of ?quantifier’s value, binding the *variable-expression* data to ?vbl1 for use in the computation of the transform’s body and allowing the pattern QuantArray1 to match the remainder of ?quantifier’s value. The “(aref *arrayname iterator1 iterator2*)” part of the data is processed by the QuantArray1 pattern binding *arrayname*, *iterator1*, and *iterator2* respectively to ?aname1, ?1iter1, and ?1iter2. The literal “1D” or “2D” gets bound to ?acase to identify the dimension of the array reference. The GenerateLoopStructure transform is going to need the low and high values for the ranges of the iterators, so the main pattern uses psuch to perform those bindings. Notice the use of ?acase to determine whether or not there is a second iterator whose high and low range values need to be bound.

This matching should not be completely mysterious because we have seen similar pattern cliches used in the explanation of psuch (Figure 1) and in the Defcomponent parameter lists. Those patterns too are matching array references and navigating to the array’s iterators to get high and low values of ranges. The context is only modestly different in this case.

The point of this example is that one can get the pattern matcher to do a lot of the work of data set up and enabling condition checking. While I would not claim that the patterns are instantly understandable, I would claim that their declarative form certainly makes them easier to understand than the equivalent procedural code distributed through out and hidden within the body of the transform proper.

3.14 Inference

Some AOG generator transformations must do some lightweight inference in order to check enabling conditions or to carry out the transformation. One such is the *SplitLoopOnCases* transformation.

SplitLoopOnCases is designed to transform one loop whose body is split into two or more cases into several separate loops that incorporate the case logic into the loop control logic. That is, suppose that we have a loop expressed in an AST form which looks like:

```
(forall (k j) (_suchthat  (_member k (_range 0 (- m 1)))
                          (_member j (_range 0 (- n 1))))
          (if  (or P1 P2)  (then  body1)
                             (else body2)))
```

The _suchthat clause contains a conjunction of logical specifications that determine the iterative behavior of iteration variables k and j. In this case, the k iterator will range from a low of 0 to a high of (m – 1) and j will range from 0 to (n – 1). Under certain conditions, the logical clauses P1 and P2 may simply alter the ranges of k or j. Say P1 is the clause that tests to see if j is equal to 0, i.e., the clause “(= j 0),” and P2 is “(= j (- m 1)).” The SplitLoopOnCases transformation seeks to transform this kind of loop so that the generator can generate code exploiting the MMX instructions of the Pentium chip, where the MMX instructions allow for parallel computation. Unfortunately, the if test in the body, may prevent the exploitation of the MMX instructions. Thus, SplitLoopOnCases transformation seeks to remove this impediment. More specifically, the SplitLoopOnCases would seek to transform the above form into

```
(forall (k j) (_suchthat   (_member k (_range 0 (- m 1)))
                           (_member j (_range 0 (- n 1))) (= j 0)) body1x)
(forall (k j) (_suchthat   (_member k (_range 0 (- m 1)))
                           (_member j (_range 0 (- n 1))) (= j (n-1))) body1y)
(forall (k j) (_suchthat  (_member k (_range 0 (- m 1)))
                          (_member j (_range 0 (- n 1)))) body2)
```

so that the if-then test does not occur inside the loop. The bodies of the first two loops are modified by substitution of the constant values of j asserted in the respective clauses. The first loop handles the special case when j equals 0, the second handles the special case when j equals (n – 1), and the third handles the general case, when (0 < j < (n – 1)). Now the transform has to simplify the iteration description of the first two loops so that they are expressed in a form that the final code generation transforms can handle. They need to be expressed in terms of ranges. To do so, it has to infer that j is fixed and therefore that part of the loop control can go away if body1 is transformed by substituting 0 for j everywhere in the body and then partially evaluated to remove redundancies. There are some additional enabling conditions that must be verified but for the simplicity of this example, we will ignore them. So, how does the transform make this inference? It makes use of the *fixediterator* rule, which has the following form:

```
(<- (fixediterator ?i ?imember ?iequal ?ilow ?ihigh)
     $(pand ($(spanto ?grunge (_suchthat $(remain ?such))))
            $(pmatch ($(spanto ?x $(bindvar ?imember (_member ?i (_range ?ilow ?ihigh)))) )
                      ?such)
            $(pmatch ($(spanto ?y $(bindvar ?iequal $(por (= ?i  ?c) (= ?c ?i)))) ) ?such)))
```

and assumes that the matcher is matching a list whose first element is the _suchthat clause of a loop. This rule is successful when an iteration variable occurs both in a _member expression and an equal (i.e. “=”) expression. On success, this rule binds ?i to the iteration variable, ?imember to the _member expression, and ?iequal to the equality expression. An example of the rule’s behavior is shown in the following:

```
(with-matching '$(with-rules (loopcontrol)
                  $(pprove (fixediterator ?v ?mem ?eql ?low ?high)))
                  '((_suchthat (_member k (_range 0 (- m 1))) (_member j (_range 0 (- n 1))) (= j 0)))
                  nil
                  (print success)
                  (pprint newbindings)
                  (terpri) ; print newline
                  (format t "~%?v=~A, ?mem=~A, ?eql=~A, ?low=~A, ?high=~A"
                        v mem eql low high))
```

When this example is executed, it will produce output like:

```
T
((?G227209   ((_MEMBER K (_RANGE 0 (- M 1))) (_MEMBER J (_RANGE 0 (- N 1)))))
 (?EQL (= J 0)) (?G227207 0)  (?G227202 ((_MEMBER K (_RANGE 0 (- M 1)))))
 (?MEM (_MEMBER J (_RANGE 0 (- N 1)))) (?HIGH (- N 1)) (?LOW 0)
 (?V J) (?G227201 NIL)
 (?G227200   ((_MEMBER K (_RANGE 0 (- M 1))) (_MEMBER J (_RANGE 0 (- N 1)))
                  (= J 0)))
 (?G227206 ?HIGH) (?G227205 ?LOW) (?G227208 ?EQL) (?G227203 ?MEM)
 (?G227204 ?V))

?v=J, ?mem=(_MEMBER J (_RANGE 0 (- N 1))), ?eql=(= J 0), ?low=0, ?high=(- N 1)
```

Using the fixediterator rule to perform the inference for the previous example, the SplitLoopOnCases transform is able to simplify the example to:

```
(forall (k) (_suchthat   (_member k (_range 0 (- m 1))))  body1x)
(forall (k) (_suchthat   (_member k (_range 0 (- m 1))))  body1y)
(forall (k j) (_suchthat  (_member k (_range 0 (- m 1)))
                          (_member j (_range 0 (- n 1)))) body2)
```

where

body1x = body1 with 0 substituted for J everywhere and the result partially evaluated, and
body1y = body1 with (n-1) substituted for J everywhere and the result partially evaluated.

To accomplish the complete transformation, the SplitLoopOnCases transformation has to do some additional analysis work that we have ignored as not being relevant to this example. However, the same techniques describe in these examples are typical of the techniques used to accomplish that additional analysis.

There are a number of rules analogous to the fixediterator rule that are used for other cases (e.g., when one or the other end of a range is an excluded value) but they work analogously to the SplitLoopOnCases transformation.

4 Implementation

4.1 Continuation Macros

4.1.1 Overview

The pattern matcher is written in a “continuation passing style” (CPS) of programming. It uses the continuation macro building blocks described in Graham 1994. The basic notion is that the main pattern matching function (i.e., pattern) and all of the pattern matching operators are defined via an the =defun macro, which is defined as

```
(defmacro =defun (name parms &body body)
  (let ((f (intern (concatenate 'string "=" (symbol-name name)))))
    `(progn
      (defmacro ,name ,parms
            `(,',f *cont* ,,@parms))
```

```
                    (defun ,f (*cont* ,@parms) ,@body))))
```

Fundamentally, for each function foo defined with =defun, two artifacts are created: 1) a conventional Lisp defun for function named "=foo" that has an addition argument *cont* which is designed to hold the current continuation, and 2) a macro that will turn calls to foo into calls to =foo by adding the *cont* argument to each call. For example, an =defun of foo might look like

```
(=defun foo (P1 P2) (bodystuff) (=values answer))
```

and would compile into the expression

```
(PROGN (DEFMACRO FOO (P1 P2) (LIST '=FOO '*CONT* P1 P2))
          (DEFUN =FOO (*CONT* P1 P2) (BODYSTUFF) (=VALUES ANSWER)))
```

The *cont* argument is a continuation that is passed to the =foo function. When =foo is ready to return a value, it does so by the =values function, which is defined as

```
(defmacro =values (&rest retvals)
  `(funcall *cont* ,@retvals))
```

The =values function "returns" by calling the current continuation. So, the expression

```
(=VALUES ANSWER)
```

in =foo will be expanded to

```
(FUNCALL *CONT* ANSWER)
```

There are several other useful functions that provide the =function analogs of lambda, multiple-value-bind, funcall, and apply. Basically, the continuation machinery provides a framework for the backtracking machinery. For greater detail, the reader is referred to Graham, 1994.

4.1.2 Restrictions on Usage

The continuation machinery is not as fully general as Scheme continuations for example but the restrictions are pretty easy to live with and do not levy a significant burden on the programmer. The restrictions are:

1. The parameter list of a function defined with =defun must consist solely of parameter names. Keywords for example are not handled.
2. Functions that make use of continuations, or call other functions that do, must be defined with =lambda or =defun.
3. Such functions must terminate either by returning values with =values or by calling another function that obeys this restriction.
4. If a =bind, =values, =apply or =funcall expression occurs in a segment of code, it must be a tail call. Any code evaluated after an =bind should be put in its body. So if we want to have several =binds one after another, they must be nested.

4.2 Backtracking Macros

Given the continuation machinery, the program provides a global variable *paths* that keeps track of choice points for backtracking. Additionally, it defines several flavors of choosing macros and functions that will record choice points on the *paths* stack for purposes of backtracking. Fundamentally, these macros and functions store callable lambda expressions that record the choices that have not yet been explored for each choice point.

Fail and cut functions are defined for manipulating the *paths* stack. The fail function basically calls the function on top of the *paths* stack. That function was constructed by some choice routine and its job is to reformulate the remaining choices (if any) into a similarly callable function, put that function back on the *paths* stack in anticipation of the next failure, and finally, call the continuation of the current choice.

The CPS style of programming never does a true function return until the pattern matching is completely finished (either with success plus bindings or with total failure). When that point is reached, the resulting value transparently returns through all previous continuations, those that have succeeded and those that have failed. Each such completed continuation has effectively run off the end of its definition thereby allowing the overall resulting value to sail through all of them back to the first call.

The pattern matcher and all of the operators are written in terms of the CPS machinery, the choice routines, the fail routine, and the cut routine. As long as all matcher routines follow a few simple rules, they can be large oblivious to the CPS complexity on which they are built. For full details of the underlying machinery, see Graham, 1994.

4.3 Binding and Unification

The unification and unification-based variable binding algorithms are from Wilsensky, 1986. See the appendix for a list of support routines.

4.4 With-Matching Macro

The form of an invocation of the with-matching macro is

(with-matching *pattern data bindinglist body*)

The with-matching macro introduces and computes values for three local Lisp variables that are available to the body of the macro:

1) **result**, which is a two-tuple containing the success/fail flag and the binding list, if any,
2) **success**, the success/fail flag, and
3) **newbindings**, the binding list produced by the matcher, if the match was successful.

Consider the example:

```
(with-matching '(?x ?y $(ptest numberp)) '(a b 10) nil
               (print success))
```

This example will be macro-expanded into:

```
((LAMBDA (#:G880 #:G881 #:G882 Y X)
   (SETF #:G880
        (MATCH '(?X ?Y
                (*PAT*
                 (LAMBDA (*CONT* DATA BINDINGS)
                  (FUNCALL '=PTEST
                          *CONT*
                          '(NUMBERP)
                          DATA
                          BINDINGS))))
               '(A B 10)
               NIL))
   (SETF #:G881 (SUBOF #:G880))
   (SETF #:G882 (SUBOF (SIBOF #:G880)))
   (LOCALBIND (?Y ?X) #:G882)
   (LET ((RESULT #:G880) (SUCCESS #:G881) (NEWBINDINGS #:G882))
     (PRINT SUCCESS)))
 NIL NIL NIL NIL NIL)
```

To protect the three Lisp variables from clashes in recursive uses of the with-matching macro, the macro wraps the body with a Lisp let in which these variables are the local let variables. The values computed for them are assigned to anonymous variables (i.e., #:G880 #:G881 #:G882) which are used to initialize the three local variables. This scheme allows recursive uses of the with-matching macro such that each use of these three variable names is the correctly scoped. Since each pattern variable (i.e., ?x and ?y in the example) will potentially be used in the Lisp computations within the body of the macro, corresponding Lisp let variables are introduced into the outer let that wraps the overall result of the macro-expansion. The localbind macro is used to generate the assignment statements that initialize these let variables using the bindings of the corresponding pattern variables. In the example, the statement

```
(LOCALBIND (?Y ?X) #:G882)
```

will be expanded into

```
(PROGN (SETF Y (GETBINDING '?Y #:G882))
            (SETF X (GETBINDING '?X #:G882)))
```

which at run-time will get the bindings for ?x and ?y and assign them to the Lisp let variables X and Y, respectively.

5 Related Work

The pattern matcher borrows ideas from many places. Certainly, the overall control structure, binding regime, and inference mechanisms are similar to those of Prolog (Clocksin and Mellish, 1987, Malpas 1987) and Icon (Griswold, *et al.*, 1996). The binding regime uses unification, which goes back to Robinson’s 1965 paper on Resolution theorem proving. The matching paradigm has similarities to other systems that rely on pattern matching (Kotik *et al.*, 1986). Operators such as Spanto and Remain were motivated by the analogous Snobol4 operators, which I always found to be incredibly direct and useful. (Griswold *et al.*, 1971).

There are somewhat less direct relationships with Wile’s work. Popart (Wile, 1994) adopts a grammar theoretical basis for its representation and is operationally slanted more toward syntax-directed parsing and language translation architectures. Wile’s more recent work (Wile, 1997) pushes the representation toward purer abstractions and seeks a theoretical basis for the relationships among and manipulations upon such abstract representations.

Motivation for the pattern matcher arises from both the transformation and program generation communities. See Feather, 1987 and Partsch, 1990.

There are related systems from the program analysis community (See Crew, 1997 and Devanbu, 1992) although the analysis systems and generation systems typically differ in their control architectures. That is, the program representation and analysis machinery are the primary controllers of program analysis systems and other functionality is subservient to the analysis machinery. In contrast, the representation and matching machinery is a small part of the AOG program generator and is subservient to the computational goals of generation. The fundamental difference in these contrasting cases is which computational objective drives the system, the matching/analysis or the generation.

6 Summary

Fundamentally, the patterns in this system are the *logical schema* of physical ASTs. The serve to hide the specific details of the physical AST structures and the variations within that those AST structures thereby reducing the complexity of the transformation code that operates on the ASTs. In that sense, these patterns are for the AOG generator system much like the logical schemas are for database systems.

7 References

1. **Allegro CL for Windows, Version 3 User Manual**, Franz Inc., 1995.

2. Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas, *Scalable Software Libraries*, **Symposium on the Foundations of Software Engineering**. Los Angeles, CA, December, 1993.

3. Ted J. Biggerstaff, *Anticipatory Optimization in Domain Specific Translation*, International Conference on Software Reuse, June, 1998a.

4. Ted J. Biggerstaff, *A Perspective of Generative Reuse*, **Annals of Software Engineering**, Baltzer Science Publishers, AE Bussum, The Netherlands, 1998b.

5. Ted J. Biggerstaff, *Composite Folding in Anticipatory Optimization*, Microsoft Research Technical Report, MSR-TR-98-22, 1998c.

6. W. F. Clocksin and C. S. Mellish, **Programming in Prolog**, Springer-Verlag, 1987.

7. Crew, R. F., ASTLOG: *A Language for Examining Abstract Syntax Trees*, In Proceedings of the USENIX Conference on Domain-Specific Languages, ASTLOG: A Language for Examining Abstract Syntax Trees Santa Barbara, California, October 1997.

8. Devanbu, P. *GENOA- A Language and Front-End independent source code analyzer generator,* Fourteenth International Conference on Software Engineering, Melbourne, Australia, 1992.

9. Martin Feather, *A Survey and Classification of some Program Transformation Approaches and Techniques,* In **Program Specification and Transformation**, Elsevier (North-Holland), IFIP, 1987.

10. Paul Graham, **On Lisp: Advanced Techniques for Common Lisp**, Prentice-Hall, 1994.

11. Paul Graham, **The ANSI Common Lisp**, Prentice-Hall, 1996.

12. R. E. Griswold, J. F. Poage, I. P. Polonsky, **The Snobol4 Programming Language**, Prentice-Hall, 1971.

13. Ralph E. Griswold, Madge T. Griswold, **The Icon Programming Language,** Prentice-Hall, 1996

14. Sonya E. Keene, **Object Oriented Programming in Common Lisp: A Programmers Guide to the Common Lisp Object System**, Addison-Wesley, 1989.

15. Helmut A. Partsch, **Specification and Transformation of Programs**, Springer-Verlag, 1990.

16. John Malpas, **Prolog: A Relational Language and it Applications**, Prentice-Hall, 1987.

17. Gordon B. Kotik, A. Joseph Rockmore, and Douglas R. Smith, **Use of Refine for Knowledge-Based Software Development**, Western Conference on Knowledge-Based Engineering and Expert Systems, 1986.

18. J. A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle,* **JACM**, pp. 23-41, 1965.

19. Guy L. Steele Jr., **Common Lisp: The Language (Second Edition)**, Digital Press, 1990.

20. David Wile, *POPART: Producer of Parsers and Related Tools, Systems Builders' Manual*, Draft Technical Report, UCS/ISI, 1994. See http://www.isi.edu/software-sciences/wile/Popart/popart.html.

21. David Wile, *Toward a Calculus for Abstract Syntax Trees* , In Proceedings of a Working Conference on Algorithmic Languages and Calculi, IFIP Working Group 2.1, February, 1997, Strasbourg, France.

22. Robert Wilensky, **Common LISPcraft**, W. W. Norton & Company, 1986.

8 Appendix: Elements

8.1 Pattern Operators

- **$(bindconst ?x constant)** will bind the constant value to ?x and succeed, allowing the pattern to advance without advancing the data.
- **$(bindvar ?x pattern)** will cause whatever the pattern argument matches to be bound to the ?x variable.
- **$(is variable expression)** An expression is evaluated by Lisp and its value bound to the variable.
- **$(none a b c)** succeeds if matched item is neither a nor b nor c.
- **$(pand pat1 pat2 pat3...)** will succeed if all patterns match.
- **$(papply function ?arg1 ?arg2 ... ?argn)** applies function to the args without advancing the data. The pattern containing it succeeds or fails based on the result of the application.
- **$(pat variable)** this matches the current item against the pattern bound to variable. This allows the data to contain patterns that describe other portions of the data.
- **$(pcut)** User invoked cut to cause search to abandon remaining choices at last choice point.
- **$(pdeclare <let list> pattern>** This allows with-matching to produce a proper let list for a pattern that is not available until execution time.
- **$(pfail)** User invoked fail causes search to backup to last choice point and select the next choice.
- **$(plet <let list> pattern)** This assures that the ?variables mentioned in the let list are local to the pattern. It accomplishes this by replacing the mentioned variables with gensym-ed variables (at run time).
- **$(plisp <list of Lisp statements using pattern ?variables>)** executes the list of Lisp statements succeeding or failing based on the return value of the Lisp code (nil-for fail, else-succeed).
- **$(pmark)** puts a mark on the choices stack which is where the next cut will be stopped.
- **$(pmatch pattern data)** This allows us to recursively match a pattern against data that we have just bound.
- **$(pnot pattern)** Succeeds if pattern fails, otherwise fails.
- **$(por a b c)** will match a or b or c (each may be data or a pattern).
- **$(pprove goal)** will invoke a rule whose consequent matches goal.
- **$(preorderwithin pattern)** will succeed if there is an instance of pat anywhere within the current subtree. Search order is top down.
- **$(psucceed)** Allows the user to assure success.
- **$(psuch slotname ?vbl <pattern of slotname contents>)** This allows us to include restrictions on the slotname of the entity bound to vbl.

- **$(ptest <lisp function of one argument>)** will call the lisp function on the on the current item succeeding or failing based on the value of the function. The function may be a lambda expression.

- **$(ptrace <lisp expression> label)** This produces trace printout of the label if the lisp expression evaluates to true.

- **$(remain ?x)** will bind ?x to the remainder of list after the current element.

- **$(spanto ?x pat)** will bind ?x to the remainder of parent list up to but not including that expression that matches pat.

- **$(within pattern)** will succeed if there is an instance of pat anywhere within the current subtree. Search order is from the leaves up the tree.

- **$(with-rules (ruleset) pattern)** defines the starting ruleset for use with any pprove operator in pattern.

- **(<- consequent antecedent)** defines a rule whose antecedent defines a method for achieving the goal expressed by the consequent.

8.2 Pattern Constructors

- **(DSTypePat <DSTypeName>)** creates a structure of the form

```
$(PLET (?SELF)
        $(PAND ?SELF $(PLISP (ISDSTYPE ?SELF (QUOTE <DSTypeName>)))))
```

- **(TypePat <typename>)** creates a structure of the form

```
$(PLET  (?SELF)
        $(PAND ?SELF $(PLISP (SUBTYPEP (TYPE-OF ?SELF) (QUOTE <typename>)))))
```

8.3 Service Routines

- **(AddBinding Variable Item Bindings)** adds the pair (Variable Item) to Bindings and returns the result.
- **(=apply fn &rest args)** adds *cont* onto the front of the args list and then applies fn to that list.

- **(ApplySubstitution expr sigma)** returns expr with bindings on sigma substituted for ?variables.

- **(=bind parms expr &body body)** is the analog of the Lisp multiple-value-bind such that the parms are bound to the values returned by expr and then body is evaluated with those bindings.

- **(cb fn choices)** if there are choices, cb will push a lambda onto the *paths* stack that calls (cb fn (cdr choices)) and then funcalls fn on (car choices). If there are no choices left, it fails.

- **(change-qvars expr)** performs pattern variable renaming returning a rewritten form of expr in which each ?variable is replaced by a ?gensymvbl where the string “gensymvbl” is formed by gensym.

- **(choose &rest choices)** is a primitive macro at the same conceptual level as fail implementing a non-deterministic choice point. It returns the first choice when first eval-ed and one choice thereafter for each call to fail until the choices are exhausted when it fails.

- **(choose-bind var choices &body body)** is a macro that creates a call to cb with the fn function defined as #'(lambda (,var) ,@body) and with choices as the third argument. This saves having to write one named function out of line for each use of cb.

- **(ComposeSubstitutions theta tau)** composes substitutions per Robinson's specification. Circularities are removed.

- **(cut)** removes the last choice point from the *paths* variable.

- **(devariablize QSymbol)** if QSymbol is a symbol in pattern variable form (i.e., whose symbol name is of the form ?variable), it returns an interned symbol whose symbol name is of the form Lisp variable (i.e., without the dollar sign).

- **(=defun name parms &body body)** creates a defun for a function =name that has *cont* prepended to the parms list and a defmacro that turns invocations of name into invocations of =name by adding *cont* to the front of the arglist.

- **(fail)** funcalls the lambda expression on the top of the *paths* stack or returns the failure symbol if *paths* is empty.

- **(=funcall fn &rest args)** does a funcall of fn with *cont* added at the front of the args list.

- **(fullbind* expr binds)** substitutes throughout expr, the values associated with any variable on the binds binding list. Values are the non-variable, recursive closure of the bindings on binds. Even though a variable may be bound to a chain of other values, if there is no non-variable value at the end of that chain, a non-interned gensym symbol will be substituted in the position of that variable in expr.

- **(getbinding x binds)** returns the value of the binding of x or nil if none.

- **(getbinding* x binds)** returns two values: 1) the value of the recursive closure of the bindings of x or nil if there is no non-variable value in the binding chain, and 2) nil if x is not bound to a value, or x's first binding pair from the binds list if x is bound.

- **(isbound x bindings)** tests to see if x is bound anything including nil.

- **(isbound* x bindings)** tests to see if x is bound to a non-variable value returning the binding pair if so, and recursively, if x is bound to a variable, it returns isbound* of that variable.

- **(IsDSType CLOSobj DSType)** returns true if CLOSobj is a CLOS object whose type is a subtype of DSType.

- **(IsVariable Item)** tests to see if item is a symbol whose symbol name starts with "?".

- **(=lambda parms &body body)** is the analog of lambda with *cont* (the continuation variable) added at the front of the parameter list.

- **(localbind vbls bindings)** creates let block defining the variables on vbls as Lisp let variable and assignment statements that will bind the values of the ?variable pattern variables to the corresponding Lisp let variable.

- **(match pat data bindings)** initial matching routine that saves *paths* to preserve any current matching operation, does a set up for a new matching operation with backtracking, and then calls pattern to do the work. On regaining control it restores *paths* and returns the results from pattern.

- **(MatchUpward pat at lroot &optional (bindings nil))** step up tree from at (which is initially bound to ?me) until pattern matches or the top of the tree is reached. It expects ?stmt in the pat to bind to the AST subtree at which the match succeeds. The value of lroot is a variable that points to the top of the AST tree.

- **(=pattern pat data bindings)** main recursive matching routine with backtracking. Returns a two-tuple (successflag newbindings).

- **(Pop1Char symbol)** returns an interned symbol whose symbol name is the same as symbol with the first character removed.

- **(ppc-pat pat window)** decompiles a pattern pat and prints it in the window stream window.

- **(PseudoDollar dollararg)** does at macro-expansion time what $dollararg would do at read-time.

- **(qvars-in expr)** returns a list of all ?variables in expr in their devariablized form (i.e., without the question mark).

- **(q?vars-in expr)** returns a list of all ?variables in expr in their variablized form (i.e., with the question mark).

- **(RemoveBindings Variable Bindings)** removes any bindings of Variable from the bindings list and returns a new binding list.

- **(ReSetBinding Variable Value Bindings)** alters the binding of Variable to be Value and returns a new binding list.

- **(RuleSet (name superset) …rules…)** defines and names a set of rules. If superset is not nil, it is the ruleset that will be tried by pprove if all rules in the name ruleset fail.

- **(starvarbinding <?variablename> <value>)** creates a binding pair (<?variablename> <value>) to be returned to the pattern matcher from plisp.

- **(undefined x)** tests x to see if it is an uninterned symbol.

- **(Unify Expr1 Expr2)** Wilensky's version of Robinson's unification algorithm. Returns two-tuple (successflag newbindings).

- **(UnifyWithBindings Expr1 Expr2 Bindings)** Wilensky's version of Robinson's unification algorithm with a set of starting bindings. Returns two-tuple (successflag newbindings).

- **(=values &rest retvals)** does a funcall to *cont* with retvals as the argument list.

- **(variablize Symbol)** returns an interned symbol whose symbol name is of the form ?Symbol.

- **(VariableMatch Variable Item Bindings)** unification-based variable matching per Wilensky's algorithm. Returns two-tuple (successflag newbindings).

- **(with-matching pat data bindings &rest body)** for all ?variables in pat, it creates corresponding Lisp let variables scope with initialization. Also establishes local variables for capturing results of match operation, calls match and evals body in the scope of all localize variables.