Unification-based Conditional Binding Constructs

by

Harvey Abramson

TR 82-7
Proceedings
of the
First International Logic Programming
Conference
Marseille - Sept. 14-17, 1982

August 1982

Department of Computer Science
The University of British Columbia
Vancouver, British Columbia V6T 1W5

Abstract

The unification algorithm, heretofore used primarily in the mechanization of logic, can be used in applicative programming languages as a pattern matching tool. Using SASL (St. Andrews Static Language) as a typical applicative programming language, we introduceseveral unification based conditional binding (ie, pattern matching) constructs and show how these can promote clarity and conciseness of expression in applicative languages, and we also indicate some applications of these constructs. In particular, we present an interpreter for SASL functions defined by recursion equations.

This work was supported by the National Science and Engineering Research Council of Canada.

Unification-based Conditional Binding Constructs

> Pattern matching should not be considered an "exotic extra" when designing a programming language. It is the preferable method for specifying operations on structured data, from both the user's and the implementor's point of view. This is especially so where many user-defined record types are allowed. [Warren,1977].

Most untyped applicative languages (such as "pure LISP" [McCarthy,1965], Lispkit LISP [Henderson,1980], or Scheme [Sussman & Steele,1975]), although they permit the construction of complex data structures, do not have built-in pattern matching. SASL [Turner,1976,1979, & 1981] does not lack pattern matching, but it is constrained so that the pattern matching machinery goes into operation only in a where definition, or when it must be decided which of a set of recursion equations defining a function f is appropriate in an application (f x). Failure of this constrained pattern matching, corresponding to a "definition error", or to applying a function to an inappropriate value, respectively, effectively halts the computation.

What seems desireable rather, is a means whereby one could, without introducing a special function to invoke the pattern matching, match a SASL value against a pattern so that if the pattern match succeeded, one could proceed with the computation (with some information, ie, name bindings, extracted from the pattern match), and if the pattern match failed, one could proceed with some other appropriate computation.

Robinson's unification algorithm, heretofore used primarily in the mechanization of logic, can be adapted for pattern matching in applicative languages. Note that although we shall be using SASL for demonstration, the results hold for, and can be adapted to, any applicative language.

We assume enough familiarity with SASL (which can be characterized by the phrases: applicative language; definition by recursion equations; non-strict functions; lazy evaluation), so that the reader can follow with ease the following specification of the unification algorithm, adapted from the formulation given in [Robinson & Sibert,1980a, 1980b].

```
||data structure for applications
applicative_pair (a,b) = true
applicative_pair x     = false
```

```
||what is an environment?
environment ()           = true
environment ((a,b):e) = environment e &
                         name a &
                         ¬(defined a e)
environment x            = false
||name a yields true for some suitable
||linguistic convention

||is a name a defined in the
||environment e?
defined a () = false
defined a ((a,b):e) = true
defined a ((c,b):e) = defined a e

||what is immediately bound to the
||name a in environment e?
immediate a ((a,b):e) = b
immediate a ((c,b):e) = immediate a e

||what is ultimately bound to the
||name a in environment e?
ultimate a e = defined a e ->
    ultimate (immediate a e) e;a

||substitute in the expression x
||the values ultimately bound to
||any name occuring in x and defined
||in the environment e
||"recreal" abbreviates
||"recursive realization"
recreal x 'impossible" = 'impossible"
recreal (x,y) e =
          (recreal x e, recreal y e)
recreal x e      = defined x e ->
recreal (ultimate x e) e; x


||if there exists an environment e'
||such that
||   recreal a e' = recreal b e'
||then e' = unify a b e
unify a b 'impossible" = 'impossible"
unify a b e =
 (function a) & (function b) ->
 'impossible";
 equate (ultimate a e) (ultimate b e) e
```

```
equate a a e = e
equate a b e = name a -> (a, b): e;
 name b -> (b, a): e;
 ¬ applicative_pair a -> 'impossible";
 ¬ applicative_pair b -> 'impossible";
 (unify x y (unify u v e)
   WHERE (u,x),(v,y) = a,b)
```

For finite objects a, b, and any environment e, the unification algorithm always terminates, either indicating unification of a and b is impossible, or finding an environment extension e' in which a and b can be recursively realized as identical expressions. We force a return of impossible when both a and b are functions, and note that unify does not terminate if both a and b are infinite objects. If just one of a and b is an infinite object, the unification algorithm terminates. An "occurs" check is not included in this specification of unification.

In adapting the unification algorithm to pattern matching in applicative languages, we first make a slight change to it by deleting the phrase

```
name b -> (b, a): e
```

from the definition of equate. This restricts unification so that names to be bound may only occur in the first argument of unify. Unless we say otherwise, when we use "unification" we now mean this restricted version. Later we shall suggest how unrestricted unification can be used in applicative languages.

Our first unification-based binding construct is given by the syntax:

```
a {- b => c; d
```

Here a is a pattern and b, c and d are SASL expressions. The symbol {- is called the "left crossbow". A pattern is an arbitrarily complex list structure containing only names (identifiers) and constants. Evaluation of this expression takes place in an implicit environment ρ. If the SASL object denoted by b can be unified (in the restricted sense!) to the pattern a, then the value of this binding construct is c evaluated in the environment extension ρ' containing the bindings of names in a derived by unification; otherwise, it is d, evaluated in the unextended environment ρ.

Example 1. Refer to the definition of equate given above. In an untyped applicative language such as SASL, one tends to introduce structure checking functions such as applicative pair in order to prevent unify from being applied to inappropriate arguments. It is much more concise to invoke pattern matching explicitly by rewriting equate as:

```
equate a a e = e
equate a b e = name a -> (a, b): e
               name b -> (b, a): e
               (u,x),(v,y) {- a,b =>
               unify x y (unify u v e)
               'impossible"
```

The idea is that if a and b both have the structure of applicative pairs, u, v, x and y get bound to the appropriate parts of a and b, and are subsequently used in the recursive call of unify; otherwise, the failure environment 'impossible" is the value of equate. The function applicative pair can, using this construct, be dispensed with.

For symmetry, we introduce the right crossbow symbol -}, and another conditional binding expression

```
a -} b => c; d
```

defined by

```
b {- a => c; d
```

Analogous to case expressions, we introduce the following notation to express, on the left hand side, trying to match a sequence of SASL objects b1,...,bn against a pattern a, and on the right hand side, trying to match a SASL object a against a sequence of patterns b1,...,bn.

```
{-BIND a AND      |  BIND-} a AND
       b1 => c1   |         b1 => c1
          ...     |            ...
       bn => cn   |         bn => cn
       default    |         default
```

defined by

```
a {- b1 => c1     |  a -} b1 => c1
        ...       |          ...
a {- bn => cn     |  a -} bn => cn
default           |  default
```

where "{-BIND", "BIND-}" and "AND" are new reserved symbols, and c1,...,cn and default are expressions.

Example 2. A SASL function ξ defined in the environment ρ by the recursion equations

```
ξ π1,1 ... π1,i1 = β1
...
ξ πn,1 ... πn,in = βn
```

where the πj,k's are patterns, and the βj's are expressions, could be represented by the SASL list structure

```
'ξ",clause_list,ρ,(),ρ
WHERE clause_list =
(...,((πj,1,πj,2,...,πj,ij),βj),...)
```

The third component ρ of the representation is the environment in which ξ is defined, implementing SASL's static naming convention. See Note 6 below for an explanation of the empty list as the fourth component, and the copy of ρ as the fifth component of the function representation. A somewhat simplified and idealized SASL interpreter for the application of a function fn, represented using the structure just defined, to a list of arguments arglist, is shown in Figure 1 at the end of the report.

Note 1. There are no more clauses to try, so fname is undefined for the original argument list given by matched++arglist. See also Note 6.

Note 2. The representation of the function should have this structure. If not, there is a "compiler" or "representation error" in fname.

Note 3. What are the possible structures for patlist and arglist?

Note 4. All arguments have been matched to patterns, so evaluate the body of the appropriate clause of the definition of the function ξ in the environment extension env'. Evaluation is carried out by a function eval which directly interprets expressions involving the basic operators of SASL, but which recursively calls apply for the interpretation of user-defined functions.

Note 5. There are still patterns to be matched, ie, we have a higher order function, and it is represented by fn.

Note 6. If patlist and arglist do not have the appropriate list structure, there is a "compiler" or "representation error". Otherwise, try to match an arg to a pattern in the environment env': if the match succeeds, recursively apply the interpreter to try and continue matching args and patterns, recording that arg has been attached to the end of the list of arguments already matched, and also recording the bindings induced by the match as env'', an environment extension of env' as the fifth component of the function representation; if the match fails, recursively apply the interpreter to the remaining clauses, recording for this recursive application that no arguments have been matched, replacing any environment extension env' by the original environment env, copied from the third component, as the fifth component of the representation, and that the argument list to which the function is to be applied is matched++arglist. Previous matches have to be undone largely because there is no discipline imposed by SASL on the use of recursion equations in definitions. It is possible, for example, to use different variables in each recursion equation, different numbers of arguments, and, there can be curious dependencies between recursion equations (see [Campbell,1979],[Turner,1981]). The fourth component (initially the empty list) of the representation of a function defined by recursion equations is used by the interpreter to record partial matches of arglist and patlist. Note that in a higher order function (Note 5), the fourth component of the representation is not the empty list.

Full unification can be introduced to applicative languages in more than one way, ie, over different domains for the arguments a and b of unify. One could, for example, introduce the double crossbow symbol {-} and the conditional binding construct

```
a {-} b => c; d
```

where a and b are patterns to be unified. Here names may occur in a and also in b. Such a construct would find use in implementing a resolution theorem prover ([Robinson,1979]) to provide SASL with a logic programming facility.

Alternatively, we could introduce the construct

```
a {-} b => c; d  {-x,y,z,...-}
```

where a and b are SASL expressions, x,y,z,... are free names which may occur in a and b, and which may be bound by unification of a and b. (The specification of free names is somewhat unpalateable, but apparently necessary if a and b are to be any SASL expressions.)

The denotational semantics for all these constructs have been defined and are to be presented in a separate paper [Abramson,1982].

We have shown how Robinson's unification algorithm can be adapted to define conditional binding expressions which give untyped applicative languages a convenient and useful notation for pattern matching. These constructs will be added to our implementation of SASL to test

1. the author's contention that just as the use of patterns in recursion equations allows the SASL programmer to eliminate most explicit uses of hd and tl (LISP's CAR and CDR), the conditional binding constructs will allow the SASL programmer to eliminate most explicit uses of type checking (or structure checking) functions like applicative pair, and

2. the feasibility of adding a logic programming capability to SASL.

A few details of our implementation of SASL may be of interest. It is implemented in Prolog [Roussel,1975], using the Definite Clause Grammar formalism of [Colmerauer,1978] and [Pereira & Warren,1980]. (See also [Warren,1977']). The Definite Clause Grammar formalism and a few associated Prolog predicates are used, following Turner, to "compile" SASL expressions to a string of combinators and global names. This string is evaluated, however, by a normal order reduction machine (also implemented in Prolog), rather than by Turner's normal graph reduction machine: a normal graph reduction machine is feasible in Prolog, but apparently only at the cost of modifying the Prolog data base once for each cycle of the reduction machine.

References

[Abramson 1982]
Abramson, The denotational semantics of several unification-based conditional binding constructs, in preparation, 1982.

[Campbell 1979]
Campbell, W.R., An abstract machine for a purely functional language, Dept. Of Computer Science, University of St. Andrews, 1979.

[Colmerauer 1978]
Colmerauer, A., Metamorphosis Grammars, in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

[Henderson 1980]
Henderson, P., Functional Programming, Prentice-Hall, 1980.

[McCarthy 1965]
McCarthy, J. et al, LISP 1.5 Programmer's Manual, MIT Press, 1965.

[Pereira & Warren 1980]
Pereira, F.C.N. & Warren, D.H.D., Definite Clause Grammars for Language Analysis, Artificial Intelligence, vol. 13 pp. 231-278, 1980.

[Robinson 1979]
Robinson, J.A., Logic: Form and Function, North-Holland and Edinburgh University Press, 1979.

[Robinson & Sibert 1980a]
Robinson, J.A. & Sibert, E.E., LOGLISP- an alternative to Prolog, School of Computer and Information Science, Syracuse University, 1980.

[Robinson & Sibert 1980b]
Robinson, J.A. & Sibert, E.E., Logic Programming in LISP, School of Computer and Information Science, Syracuse University, 1980.

[Roussel 1975]
Roussel, P., Prolog: manuel de reference et d'utilization, Groupe d'Intelligence Artificiel, Universite de Marseille - Luminy, 1975.

[Sussman & Steele 1975]
Sussman, G.J. & Steele, G.L., Jr., Scheme: an interpreter for extended lambda calculus, AI Memo 349, MIT AI Lab, 1975.

[Turner 1976]
Turner, D.A., SASL language manual, Dept. of Computational

Science, University of St. Andrews, 1976, revised 1979.

[Turner 1979]
Turner, D.A., A new implementation technique for applicative languages, Software-Practice and Experience vol. 9 pp. 31-49, 1979.

[Turner 1981]
Turner, D.A., Aspects of the Implementation of Programming Languages: The Compilation of an Applicative Language to Combinatory Logic, Ph.D. Thesis, Oxford, 1981.

[Warren 1977]
Warren, David H.D., Implementing Prolog- compiling predicate logic programs, DAI Research Reports 39,40, University of Edinburgh, 1977.

[Warren 1977']
Warren, David H.D., Logic programming and compiler writing, DAI Research Report 44, University of Edinburgh, 1977.

```
apply (fn,arglist) =
 fname,(),env,matched,env' {- fn                                          ||note 1.
  => fname,'undefined for",matched++arglist;
    (fname,((patlist,body),clauses),env,matched,env') {- fn               ||note 2.
      => (BIND-} patlist,arglist AND                                      ||note 3.
         (),()       => eval body env'                                    ||note 4.
         patlist,() => fn                                                 ||note 5.
         (pattern:patterns),(arg,args)
          => (env'' ¬= 'impossible"                                       ||note 6.
             -> apply ((fname,((patterns,body),clauses),env,(matched++(arg,)),env''),
                       args);
                apply ((fname,clauses,env,(),env),
                       matched++arglist)
             WHERE env'' = unify pattern arg env'
            );
         'representation error: ",patlist,' or ",arglist,nl              ||BIND-} default
        );
    'representation error: ",fname                                        ||note 2 default
   )
```

Figure 1.