UNIFORM — A LANGUAGE BASED UPON UNIFICATION WHICH UNIFIES (MUCH OF) LISP, PROLOG, AND ACT 1*

Kenneth M. Kahn

UPMAIL, Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, University of Uppsala
Uppsala Sweden

ABSTRACT

Uniform is an AI programming language under development based upon augmented unification. It is an attempt to combine, in a simple coherent-framework, the most important features of Lisp, actor languages such as Act 1 and SmallTalk, and logic programming languages such as Prolog. Among the unusual abilities of the language is its ability to use the same program as a function, an inverse function, a predicate, a pattern, or a generator. All of these uses can be performed upon concrete, symbolic, and partially instantiated data. Uniform features automatic inheritance from multiple 3uper classes, facilities for manipulation of programs, a limited ability to determine program equivalence, and a unification-oriented database.

I A LANGUAGE BASED UPON UNIFICATION

Uniform is based upon the idea of an extensible unification procedure. All programs are extensions to the unification process. Unification plays the roles of pattern matching, evaluation, message passing, inheritance, and symbolic evaluation. In the process of unifying the factorial of 3 with an integer n, n is unified with 6. The concatenation of x and the list (c d) unifies with the list (a b c d), resulting in x being unified with the list (a b). Unifying the nth element *of* the list of all prime numbers with 5 results in n unified with 3. Unifying the reverse of a list of variables x and y with a list z, unifies z with a list of y and x. Unifying a member of set x with a member of set y yields a member of the intersection of x and y. Unifying a description of red chairs with a description of big chairs produces a description of big red chairs. And so on.

A. Unification———What it is

Unification was invented for use in resolution theorem provers <1> and has since been used in a few programming languages. Two well-known examples are Prolog which is a programming language based upon resolution theorem proving <2> and Qlisp <3>. In these languages unification is only one facility among others. In Uniform unification is augmented so that no other mechanism such as resolution, automatic-backtracking, or evaluation is needed.

Unification is the process of generating the most general common instance of a set of descriptions. It is implemented as a process that returns a "unifier" which if applied to any of the original descriptions produces the sought after instance. In the most common case, the unifier is simply an environment describing variable bindings and it is applied to a description by substituting its values for the variables in the description. As an example,*

```
(unifier-of (foo x 'a r)
            (foo 'b y r))
```

produces "(unifier (x 'b) (y 'a))" which if substituted into either description produces the instance "(foo 'b 'a r)". Unification is defined to produce the most general instance which means that the instance produced must unify with all other possible instances.

Pattern matching, which has played such a large role in most AI languages, is a special case of the unification of two descriptions. The pattern is a description containing variables which is matched against a description without any variables. If the match succeeds, the unifier produced is a set of match bindings which when substituted into the pattern produces the other description.

Unification has many advantages over pattern matching. There is no distinction between patterns which bind variables and patterns which use the value of variables. This allows the meaning of a pattern to be determined dynamically and is crucial in using programs in multiple ways and directions.

Patterns can be matched against patterns. This is important for dealing with partial information. It is also useful in determining if one pattern is a special case of another. Uniform uses unification this way so that by default more specific knowledge is used first.

The order in which sub-problems (recursive calls to unify) are made does not affect the outcome. This gives the implementation a freedom of optimization not possible in most pattern matchers. Parallel implementations of unification which could take advantage of the parallel hardware of the future are also possible.

*The research reported herein was supported by the Swedish Board for Technical Development (STU).

•As in Lisp, Uniform follows the convention that constants are numbers or are quoted and variables are unquoted symbols.

B. How Unification is Augmented In Uniform

Unification is a syntactic process. Its only concern is that forms have the same "head", their arguments unify, constants are equal and recursively that bindings unify. It cannot unify the sum of x and 3 with 7 by unifying x with 4. Uniform's augmented unification leaves it up to the forms involved to unify as they see fit where the traditional syntactic unification process is used only as a default. Except for a small set of primitive types this augmentation is described in Uniform by the user.

The only constraint upon augmentations to the unification of two descriptions is that the resulting unifier produce equivalent descriptions when applied. For example, the unifier describing x as 4 produces 7 and the sum of 4 and 3 when applied to 7 and the sum of x and 3. Either the sum of 4 and 3 is considered equivalent to 7 or else the augmentation that produced the unifier is invalid.

A few of the primitive forms stretch this view of unification. For example, the primitive form "(print)" will unify with anything and as a side effect the other form will be printed. The primitive "(ground)" will either unify with any constant or will eventually fail if the other is a variable or contains variables that are never bound. There are primitives for dynamically creating unification variables and forms, primitives for escaping to Lisp's Eval (a theoretically unnecessary, yet very useful, primitive), for forcing sequentiality (critical in situations involving input-output or other side effects), for determining if two things are the exact same object (Lisp's EQ, critical when dealing with circular structures and important for efficiency), and for the logical connectors "and", "or", and "not". Without this small set of primitives, unification would not be adequate as the sole basis of a programming language.

Uniform's augmented unification not only has the unusual features of delegating to the forms involved but is based upon a new very parallel algorithm. Sub-unifications of corresponding arguments are computed independently and the resulting unifiers are unified. A unifier is either a set of bindings or a set of possible unifiers whose members are computed only upon need. This mechanism replaces the automatic backtracking in Prolog and Qlisp. In addition, when the descriptions cannot be unified the algorithm produces failure descriptions which are used internally and by the user for debugging. Unification is also used to implement the occur check and the "ground" primitive. The algorithm is described further in <4>.

C. How Uniform uses Augmented Unification

The top level loop of Uniform is, analogous to Lisp, a "read unify print" loop. The user types a form which is unified with previous user assertions. If the relation is either "=" (are they unifiable?) or "unify" (what is the most general common instance?) then the unification algorithm is applied to the arguments. If the unification fails, it looks in the user's assertions for possible paths between the two descriptions and follows such paths in a shortest-first manner.* The steps of a path are typically described using the "=" relation which declares that its arguments are unifiable. If an assertion has constraints (a body) they are checked. If the unification is successful the unifier produced is applied to the user's problem and the resulting instance is printed.

Here is a short sample session:

```
User:    (unify (foo x) (foo 'a))
Uniform: (foo 'a)
User:    (= (plus x 3) 7)
Uniform: (= (plus 4 3) 7)
User:    (= (foo 3) (foo 4))
Uniform:
(or ;the problem is one of the following:
 (failure 'arguments-do-not-unify (foo 3) (foo 4>>
 (failure 'constants-do-not-unify 3 4))
```

The other action a user can take is to extend unification by presenting assertions. For example, the following is how unification can be extended to handle factorial:

```
(assert (= (factorial 0) 1))
(assert (= (factorial n)
           (• n (factorial (- n 1)))))
```

The first assertion extends unification so that anything which unifies with factorial of 0 is unifiable with anything which unifies with 1. The second clause states that anything that unifies with factorial of anything also unifies with anything which unifies with the product of that anything and factorial of one minus that anything. The "n" in the second clause is universally quantified.

This definition of factorial can be used in many ways as illustrated below. How this is realized is described in <4>. An example of its use as a function is:

```
User:    (= (factorial 3) (integer n))
Uniform: (= (factorial 3) (integer 6))
```

If we unified "(factorial 3)" with "n" in the above example then "n" would have been unified with "(factorial 3)" instead of 6. The use of "integer" gives the user greater control of the unification at the price of having to type variables. To alleviate this the type "primitive" can be used which unifies only with primitive types. Use of the "primitive" type corresponds to evaluation in Lisp.

Our description of factorial can be used as an inverse function (or a pattern which matches only integers which are factorials) by unifying "(factorial n)" with ground forms. The value of "n" or a failure description is computed reasonably efficiently due to some cleverness in the general description of "•". Factorial of a constant can be used as a pattern without necessarily being evaluated. In unifying "(factorial 1000000)" with

•Shortest-first differs from breadth-first in that the system maintains a data structure so that it knows the minimum length of all possible paths between two types of forms and tries the shortest ones first.

17, a constraint that 1000000 be a divisor of 17 is generated after 2 steps and causes immediate failure.

D. What is interesting about Unification

I am developing Uniform both to produce a simple yet powerful AI language and to explore unification. Unification is interesting not only as a basis for computation but as a source of insight into questions about specialization, generalization, object merging, inheritance and multiple super classes.

Traditional unification is concerned with creating the most general syntactic instance of a set of descriptions. Class and instance is defined purely in terms of the form of the descriptions. The more instantiated versions of a class are its instances. From an AI point of view, the semantic, not syntactic, instances are what is interesting. A description of a particular equilateral triangle is a semantic instance of the prototype equilateral triangle, regular polygon, triangle, closed figure, line drawing, geometrical figure and so on. The semantic unification of regular polygons and triangles should produce equilateral triangles. Unification not only generates an instance of two descriptions but produces a unifier, a description of a viewpoint under which the two descriptions are the same. For syntactic unification, the viewpoint is an environment giving bindings and constraints to the variables in the descriptions. A view of polygons and triangles that makes them the same is a description of equal-sidedness and three-sidedness. A viewpoint is not the same as the instance. An equilateral triangle is more than equal-sided and three-sided, it is also a closed geometrical figure. Attaining this kind of semantic unification is a direction this research is headed.

As another example consider the unification of two trivial Lisp programs: (cons head tail) with (list first second). The instance of the two programs can be described in two ways, as (cons first (list second)) and as (list first (car tail)). What is often of more interest than the instance is the viewpoint (or unifier) which produced it. In this example, the viewpoint identifies "first" with "head", "tail" with "(list second)", and "second" and "(car tail)".

Unification can be used to dynamically determine who is an instance of who. Suppose descriptions x and y unify to produce a common instance z. If z equals x but not y, then x is an instance of y.

Unification is a process that provides some insight into not only classes and instances but also equivalence. The unifier *of* two equivalent descriptions is an empty environment. If the unifier of two descriptions is an environment which only binds variables to variables, then the two description are equivalent except for variable names.

The augmentations of unification in Uniform are almost always stated as equivalences between programs. For example, consider a definition of "Append" in Uniform,

```
(assert (= (append () back)
           back))
(assert (= (append (cons first rest) back)
           (cons first (append rest back))))
```

Occasionally an augmentation of unification will explicitly describe an instance. This is expressed as a second-order unification as illustrated in the following example,

```
(assert
 (= (unify (a-divisor-of n) (a-divisor-of m))
    (a-divisor-of (greatest-common-divisor m n))))
```

This example is computationally interesting since can be used in appropriate cases to intersect virtual sets very efficiently. "(a-divisor-of 6)", for example, is logically, but not computationally, equivalent to "(a-member-of (set 1 -1 2 -2 3 -3 6 -6))".

The dual of the unification process is generalization. Unification finds the most general description which is a specialization of some descriptions, while generalization finds the most specific description which has as specializations the descriptions. Surprisingly generalization of two descriptions can be implemented using the unifier of the descriptions. When a unifier is just a set of bindings, it is used in a backwards fashion to substitute constants for variables. Generalization is more complicated when different variables are bound to the same constant or when the unification fails. This is described further in <4>.

Generalization is well-known to be a useful process. Winston's program which learned the concept of an arch by generalizing structural descriptions is a classic example. One of my goals is that a program such as Winston's would be significantly shorter and simplier, if written in Uniform. Much of the program would augment generalization beyond syntax in a similar manner to how unification is augmented currently in Uniform.

E. Unification and Circularity

Another of Uniform's extensions of unification is in the handling of circularity. Traditionally unification is defined to perform an "occur check" on every variable binding. The check causes a failure if a variable is bound to something which contains that variable, ever comes to contain that variable in the future, or recursively is bound to some structure which contains variables that fail the check. Because of the expense of making such a check most implementations of Prolog do not make the check.

In Uniform's augmented unification, the user can specify for each type of form whether the occur check should be performed. For example, "plus" does not perform the check since there is nothing wrong with unifying x with the sum of x, y and z provided the sum of y and z can unify with 0. Without the check the system needs to be able to unify circular structures. In the following example "cons" does not do the check. The variable

x is bound to "a" consed with itself while y is bound to "a" consed with a cons of "a" and itself. The two structurally different infinite lists of "a"s unify successfully.

```
(and (unify x (cons 'a x))
     (unify y (cons 'a (cons 'a y)))
     (unify x y))
```

What is difficult about unifying "x" and "y" above is avoiding an infinite recursion. By being careful about the order in which sub-forms are unified it is possible to avoid this. Intuitively, the algorithm assumes that a unification will succeed before working on the sub-unifications so that if the same problem appears again it will succeed immediately without causing further recursions <4>.

Occur checks can also be used to prevent certain types of inconsistencies. Consider the following example from Peano arithmetic:

```
;a successor of any x is greater than x
(assert (> (successor x) x))
 ;is there a y greater than its successor?
(> y (successor y))
```

If one considers binding "y" to "(successor (successor y))" an inconsistency, then "successor" should perform the occur check to prevent it. Another view is that "y", as an infinite number of applications of "successor", is a representation of infinity. Infinity is greater than (and also less than) its successor.

II RELATIONSHIPS WITH OTHER LANGUAGES

Uniform was designed and developed with the initial goal of incorporating the most important features of Lisp, Act 1, and Prolog into a single coherent framework. Future plans include the incorporation of the notions of descriptions, frames, and experts as found in <5>, constraints as in <6>, partial evaluation as in <10>, and quasi-parallelism and graphics as found in Director <7>. My basic research strategy begins with the belief that many existing languages and systems contain very general and powerful facilities, but each one has only a small subset of the union of these facilities. Furthermore, a simple coherent union of these facilities is both possible and desirable. Boley's research on the FIT language shares this research strategy <8>. FIT, however, is based upon a generalized notion of variables, assignment, pattern matching and demons.

A. Uniform and Lisp

Uniform, having been built upon MacLisp, in a trivial sense incorporates all of Lisp's abilities. There is a primitive for interfacing directly with Lisp. Of course, using this primitive one loses the ability to run programs symbolically or backwards.

The more important way in which Uniform incorporates Lisp is the ability to write Uniform programs that look very much like Lisp. For example, one can write "Append" in Uniform as follows,

```
(assert
 (r (append front back)
    (cond ((null front) back)
          ((cons (first front)
                 (append (rest front) back))))))
```

This program does not mean the same as the corresponding Lisp program. It states an equivalence not an evaluation step. Unification does not have eval's sense of direction or notion of simplification. Instead of evaluation, a form is unified with a variable constrained to be of a particular data type such as integer or s-expression or constrained to consist of only primitive data types. This corresponds closely to lazy evaluation in Lisp. Lacking the control information implicit in Lisp, Uniform's interpreter is in general slower. In return the append program above can be executed symbolically, used in pattern matching, run backwards, append any kind of list, and be used in judging program equivalence.

The reason the "Append" program looks like Lisp is that the primitives "Null", "First", "Rest", and "Cond" can be written in Uniform. The primitives of Lisp that currently are difficult to write include those that perform side-effects such as "Setq", "Rplaca" and array operations. It is possible to implement them fairly straight-forwardly in Uniform, but the implementation is unacceptably inefficient. Essentially a cell is represented as a list of its previous values where the last cons contains the current value and a variable for future bindings. Other possibilities for implementing side-effects in Uniform are to use tail-recursion optimization or reference counts to know when it is safe to re-use the current structure rather then copy. This shortcoming of Uniform is an active area of research.

The most essential property of Lisp, the ability to run large programs efficiently is lacking in Uniform. A compiler is planned and it is hoped that it will produce acceptable code. Compilers for SmallTalk <9>, Director <7>, and Prolog <2> contain relevant techniques as do compilers based upon partial evaluation such as the one for the Lima pattern matcher <10>. Another area of research that hopefully will lead to an acceptable level of efficiency is work within logic programming languages for describing and using control or meta-level information <11>.

B. Uniform and Actor Languages

Computer languages based upon computational entities called "actors" offer modularity, parallelism, full extensibility of both data and functions and a simple but powerful computational semantics. An early version of Uniform was attempted in Act 1 <12>, a language that takes the idea of actors to the extreme. Many of the facilities of Act 1 would have been available in Uniform, including its excellent primitives for describing concurrent computation. Unfortunately the current implementation of Act 1 is too slow to build a practical interpreter upon it.

Act 1 is a message passing language based upon the convention that actors be able to respond to "eval" and "match" messages. Uniform can be viewed as a language in which forms pass "unify" messages between themselves and their parts. As we saw in the previous section, unification subsumes evaluation. Unification clearly subsumes the match messages in Act 1 since pattern matching is Just the special case of unification where one of the forms contains no variables.

One of the important features of actor languages is the ability to describe a new data structure and have old programs use it without modification. This is a consequence of the fact that programs depend only upon the behavior of data in response to messages. A list is any actor which can answer "first" and "rest" messages. The analogous statement about Uniform is that a list is any form that can unify with "(cons x y)". For example, suppose we want to define a new kind of list which internally is represented by two lists, one for the original members and one for those deleted. The advantage of these "delete lists" is that deletion becomes a very cheap and pure operation in return for a little overhead on other operations.

They can be defined as follows in Uniform:

```
(assert
 (= (delete-list deleted (cons first rest))
    (rules first
           ((member deleted)
            ;is already deleted so skip it
            (delete-list deleted rest))
           ((?) ;otherwise the first element is ok
            (cons first
                  (delete-list deleted rest))))))
```

This is all that is needed to run any program that works on lists since it provides a path from "delete-list" to "cons". If we describe how to delete items for "delete-list" it will be used before the delete operation defined for "cons". Notice that this way of implementing lists as anything that can unify with a "cons" of two variables subsumes the inheritance mechanism in languages like SmallTalk and Director. Uniform always tries first the shortest path between two structures. The path to delete list's delete operation is shorter than one through cons's delete so it is followed first. Of course sub-classes are possible. If x-lists only unify directly with y-lists which unify with z-lists, then definitions of operations upon x-lists will be used before those for y-lists which in turn will be used before z-list's definitions.

This same mechanism works for multiple super classes. If we define how horizontal-dashed-lines unify with horizontal-lines and with dashed-lines then operations upon either one can be applied to horizontal-dashed-lines. Since Uniform follows shortest paths first, the multiple super classes are searched in a breadth-first fashion.

One very important part of some of the actor languages is the user definable control structures and ability to compute in parallel <13> <12>. This is a serious deficiency of Uniform. The plan is to add such information as advice to the interpreter as to how to go about doing the unifications. This approach is similar to one taken in Metalog <11>. The appeal of separating logic from control is that a user can develop and test the logic or competence of a program before adding control information to improve its performance <14>. Also different uses of the same program may be helped by different control information.

C. Uniform and Logic Programming

In recent years a number of logic programming languages have appeared. Most notable is Prolog, a programming language which resembles Planner <2>. (One of Prolog's major improvements over Planner is its use of unification.) Programs in Prolog are axioms in the first-order predicate logic restricted to Horn clauses. Programs are executed by a resolution theorem prover. What is special about Prolog is that it is intended as a general purpose programming language meant to compete with compiled Lisp as well as with Planner-like languages. The objection to logic as being an excessively constrained manner of reasoning is irrelevant to its worth as a programming language. One would not want to build AI programs upon an "informal" Lisp. The objection to logic that it is not concerned with control over the use of knowledge is a serious one. There are many advantages however, to having a programming language based upon logic with a separate control component for improving performance such as IC-Prolog or Metalog <11>.

When compared with Lisp, Prolog has many advantages and a few very serious disadvantages. Prolog shares with Uniform the ability to use the same program in many ways. For example, the Prolog definition of "append" can be used not only to compute the result of appending two lists together but can also be used as a predicate to verify that the result of appending two lists is a third list, as a generator of pairs of lists that append to a particular list, as a way of finding the difference between two lists, and as a generator of triples of lists such that the first two append to form the third.

Prolog has a few other features which Lisp lacks. Among them are the ability to compute with partially instantiated structures, a convenient way to handle multiple outputs, and the use of pattern matching instead of explicit list construction and selection. On the negative side, Prolog implementations are the result of a much smaller implementation effort than the major Lisp dialects and correspondingly lack good programming environments, i/o facilities, adequate arithmetic, and the like. Attempts to embed Prolog in Lisp (e.g. QLOG <11>) may alleviate this. Among Prolog's more fundamental problems are a dependence upon automatic backtracking, a lack of user control over search, and a lack of an efficient substitute for impure operations.

Uniform was developed with the goal of capturing and improving these positive aspects of Prolog. Uniform supports all the uses of a definition that Prolog does and an additional few For example, Uniform's definition of "append" is equivalent to Prolog's and can also be used as an implementation of segment patterns.[1] In addition, it is all the knowledge about "append" the system needs to answer questions about program equivalence.

*For example, the Uniform description (append (list x) (list x) y '(center) y) (or equivalently using read macros *(x x y* 'center !y)) matches a list whose first and second elements are the same and the rest of the list has the symbol "center" in the middle surrounded by equal list segments.

For example, work is under way ao that Uniform can successfully unify the following for all lists x and y.

```
(s (append (reverse x) y)
   (reverse (append (reverse y) x)))
```

In Uniform one can augment the unification of relations other than the "a" relation and so can write in Prolog's relational, as opposed to a functional, style. The following is a Uniform program for defining the "grandparent" relation (which can be used as the "grandchild" relation too).

```
(assert (grandparent-of grandchild grandparent)
        ;the above is true if the following holds
        (parent-of a-parent grandparent)
        (parent-of grandchild a-parent))
```

The program says that two variables are in the grandparent relation if a child of the first variable unifies with a parent of the second. As in Prolog the variables "grandparent" and "grandchild" are universally quantified and "a-parent" is existential (by virtue of not being in the "head").

III A DETAILED EXAMPLE

As a simple example that exemplifies many of Uniform's features let us consider an implementation of association lists. It is a typical example of how the same program can be used to construct a data structure and to compute with the same data structure. Besides reducing the amount of programming it makes it impossible for the accessing programs to be based upon a mistaken notion of how the structure is constructed.

The following is an implementation of association lists:

```
(assert association is in the front of the list
 (= (association-of key
                    (cons (list key value) rest))
    value))
(assert ;otherwise "cdr" down the list
 (s (association-of key (cons first rest))
    (association-of key rest)))
```

This program can best be understood by seeing how it can be used. First let us build a list of associations between objects and colors. We can associate sky with blue in a list colors by unifying "(association-of 'sky colors)" with the symbol "blue". The variable "colors" is unified with (cons (list 'sky 'blue) rest-1) or in an alternative syntax "([sky 'blue] !rest-1]". In other words, "colors" is a list whose first element is a list of "sky" and "blue" and the rest is an unbound variable. If we next associate grass with green and ocean with blue, the "colors" list will be bound to "[sky 'blue! ('grass 'green] ['ocean 'blue] !rest-33". Were we to add any of the associations already In colors it would succeed without making any new bindings.

The association list can be used in many ways. If we associate grass with the variable "grass-color", it will be bound to green. Or if we associate the variable "blue-thing" with blue, blue-thing will be bound to sky. If that falls later or other alternatives are desired, then blue-thing will be unified with ocean.

If that alternative fails, a weakness of logic programming is revealed. The problem of unifying (association-of blue-thing colors) with 'blue is interpreted as "is it possible that blue-thing is associated with blue and if so how". If we reject an answer it is interpreted as "is it possible in some other way". If an association list "ends" with a variable, then the answer is always yes. The first answer was "yes if blue-thing is sky", the second answer was "yes if blue-thing is ocean" and the next answer was "yes if colors is []'sky 'blue] ['grass ' green] [*ocean 'blue] [blue-thing 'blue] frest-4]". In other words, "yes, if colors is an association list with the variable blue-thing associated with blue".

This problem was revealed when in answer to the question "what can be blue" Uniform replied, "the sky, the ocean and any blue thing". It was the question that was at fault, not the answer. It should have been "what is known to be blue". This cannot be expressed in logic programming languages (without finding and binding all the variables in colors to unique constants). Uniform has a primitive relation to distinguish the two interpretations. The primitive "known" marks the incoming unifier so that a failure results if any of its unbound variables are unified.

A strength of Uniform is that one can define and use an association list which has variables in it. Dealing with partial knowledge is very important in AI. One can express, for example, that John's apple is either red, green or yellow by associating John's apple with a variable for the color of John's apple and unifying that variable with the disjunction of red, green, and yellow. Later if we learn more about the color of John's apple we can specify it further, in the meanwhile we can use what is known about his apple.

IV CONCLUSIONS AND FUTURE RESEARCH

A surprising result of this research is that unification, a process of generating the most general instance of a set of descriptions, can be such a powerful basis for a programming language. Unification unifies the essence of Lisp, Act 1, and Prolog into a simple coherent framework.

Uniform is far from complete.* Some of the

•Extending Prolog to allow a functional style of programming would not be too difficult <11>. More difficult would be to support these functional definitions in unification.

•At the time of this writing, the implementation does not run all the examples in this paper. The unification algorithm and Its primitive types, the path following, and the top-level works and is capable of running examples such as "append", "association-of", and "grandparent". Factorial, the "rules" primitive, and the "known" primitive do not work yet. Examples involving them have been hand-simulated.

major avenues of future research follow.

Developing and incorporating the dual of unification, generalization, into the language. The duality between unification and generalization is striking and the ability to implement them both using the same mechanism is surprising.

A shortcoming of Uniform and Prolog is their inability to use negative information. In the previous example of association lists we cannot prevent a key from having more than one association. Uniform will be extended to be able to use the following:

```
(assert
 (not (= (association-of
          key
          (cons (list key value) rest))
       (not value))))
```

This would cause the unification to fail if the key is found but the values do not unify. Negative information can be used in a default strategy which concurrently tries to unify two descriptions and to show that they are not unlfiable.

In the process of unification a variable may acquire multiple constraints. As a default, Uniform simply conjoins them. If later an attempt is made to give the variable a value, then the constraints disappear if the value satisfies them, otherwise it fails. Inconsistent constraints do not cause failure unless there is an attempt to use them. If the constraints have a unique solution then only that value can unify with them, but the system does not compute that value. The unification of constraints appears to be a natural place to use some of the constraint satisfaction techniques found in Steele and Sussman's constraint system <6> and the XPRT system <5>.

We have explored the unification of descriptions of programs. Exploring the unification of other complex structures such as frames, scripts, and units should be equally valuable. Much of what systems such as FRL, XPRT, SAM and KRL do is match complex declarative structures with others. Unification, a more general and powerful process than pattern matching, promises to be very useful for dealing with these structures.

As has been pointed out elsewhere (<14> among others) there is much to be gained by separating the control and logic components of an algorithm. Uniform programs have much less control information in them than equivalent Lisp or Act 1 programs. A general search strategy is used as a default so that the factual or competence component of programs can be developed and tested without the added complexity of being concerned with efficiency. The efficiency can be put in later and kept lexically separate from the rest of the program. A compiler is planned which will be written in Uniform and produce Lisp code.

It is very difficult to evaluate the worth of computer language based solely upon small programs. Prolog, for example, becomes less desirable as the size of the programs grow due to its impoverished notion of errors and debugging and its reliance upon automatic backtracking. Work has been done to alleviate these particular failings in Uniform, but experience in using Uniform is lacking. Concurrent with the research suggested above, Uniform needs to be tested by writing large complex programs in it. In this respect Uniform is way behind Lisp et. al.

REFERENCES

<1> Robinson, J. "A Machine-oriented Logic Based on the Resolution Principle", Journal of the ACM, 12:1, January 1965

<2> Warren, D. "Implementing Prolog — compiling predicate logic programs", Department of Artificial Intelligence, University of Edinburgh, D.A.J. Research Report 39, May 1977

<3> Sacerdoti, E. et. al. "Qlisp: A Language for the the Interactive Development of Complex Systems", SRI Technical Note 120, 1976

<4> Kahn, K., "Implementing Uniform — an AI language based upon Unification", forthcoming

<5> Steels, L., "Reasoning Modeled as a Society of Communicating Experts", MIT AI Lab TR-542, June 1979

<6> Steele G., "The definitions and implementation of a computer programming language based on constraints", MIT AI Lab TR-595, August 1980

<7> Kahn, K., "Director Guide", MIT AI Memo 482B, December 1979

<8> Boley, H. "Five Views of FIT programming", Fachbereich Informatik, University of Hamburg, Nr. 57, September 1979

<9> Goldberg, A., Kay A. editors, "Smalltalk-72 Instruction Manual" The Learning Research Group, Xerox Parc, March 1976

<10> Emanuelson, P. "Performance enhancement in a well-structured pattern matcher through partial evaluation", ph.d. thesis, Software Systems Research Center, Linkoping University, Sweden, 1980

<11> Tarnlund, S-A. ed., Proceedings of the Logic Programming Workshop, John von Neumann Computer Science Society, July 1980

<12> Lieberman, H., "A Preview of Act 1", submitted for publication

<13> Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, 8:3, June 1977, 323-364

<14> Kowalski, R., "Algorithm = logic + control", Communications of the ACM, 22:7, 1979