Pattern Encoding

Encoding Algebraic Laws as Patterns

The laws of algebra can be expressed as transformation patterns. "Equal" means that one form can be freely substituted for the other whenever the patterns match.

Commutative Law

```
x*y = y*x

equal[prod[x,y],prod[y,x]]

P[Q[x,y],Q[y,x]]
```

Associative Law

```
(x*y)*z = x*(y*z)

equal[prod[prod[x,y],z],prod[x,prod[y,z]]]

P[Q[Q[x,y],z],Q[x,Q[y,z]]]
```

Distributive Law

```
x*(y+z) = (x*y) + (x*z)

equal[prod[x,plus[y,z]],plus[prod[x,y],prod[x,z]]]

P[Q[x,R[y,z]],R[Q[x,y],Q[x,z]]]
```

Proof Needs Semantics

This example illustrates why automated proof (and intelligent computation) is very unlikely. We must design programming and verification systems to be *interactive*, so that they augment human intelligence rather than attempting to emulate it. Note that conventional programming puts the interaction in a batch mode.

```
For any natural number n>=2, (n^3-n)/6 is an Integer

      <[n,2] or isInt[div[minus[power[n,3],n],6]]

      L[n,2] or P[Q[R[S[n,3],n],6]]
```

where the semantics is

```
L is <          less than
P is  isInt     type check, to be proved
Q is  /         divide
R is  -         subtract
S is  ^         power
```

A person may elect to factor the expression in question, in order to understand more:

```
(n^3-n) = n*(n^2-1) = n*(n+1)*(n-1) = (n+1)*n*(n-1)
```

A machine can do this, not in order to understand, but as part of a set of automated transformations to be explored.

```
R[S[n,x],n]  =>  T[n,R[S[n,R[x,1]],1]]

             =>  T[n,T[U[n,1],S[n,1]]]

where       T is  *          multiply
            U is  +          add
```

Note how very specific these patterns are. These rules could be generalized, but it is difficult to know in advance in which direction the generalization should be formulated.

Number Facts

A human might next retrieve a collection of esoteric number facts:

> For any three numbers in a row,
> there must be at least one even number
> and at least one number divisible by three
> and these two numbers are not the same number.
>
> so, the product contains divisors of 2 and 3 (i.e. 6)

A machine can't make this step, because there are too many esoteric number facts. Even with very sophisticated meta-knowledge to steer the selection of which number facts to explore first, finding the correct set of facts which leads to a proof is generally not possible. The problem, for example, may be only slightly different, but would then require entirely different esoteric number facts:

```
if n is odd, then (n^3-n)/8 is an Integer

      div[n,2]
            or
      isInt[div[minus[power[n,3],n],8]]

      Q[n,2] or P[Q[R[S[n,3],n],8]]
```

Induction

We simply do not know if there are automated paths, using different proof strategies, which reduce all mathematical problems to trivialities. For example, machines can do *induction*. Rather than recalling esoteric number facts to generate a natural, intelligent proof, we could have gone blindly forth in the above problem, trying an inductive proof:

```
For any natural number n>=2, (n^3-n)/6 is an Integer

      base:        (2^3-2)/6   isInt              (the base case is not n=0)

      general:     (k^3-k)/6   isInt

      show:        ((k+1)^3 - (k+1))/6     isInt

            ((k+1)^3 - (k+1))/6

            = (k^3 +3k^2 + 3k +1 - k - 1)/6

            = (k^3-k)/6 + 3(k^2+k)/6

      if A isInt and B isInt, then (A + B) isInt

            (k^3-k)/6   isInt                    assume general

            3(k^2+k)/6  isInt                    to show

            k(k+1)/2    isInt                    lemma

            ((k+1)^3 - (k+1))/6     isInt        QED

      Lemma:       k(k+1)/2    isInt

            base:        2(2+1)/2    isInt

            general:     k(k+1)/2    isInt

            show:        (k+1)(k+1+1)/2    isInt

                  (k+1)(k+2)/2

                  = (k^2 + 3k + 2)/2

                  = k(k+1)/2 + 2(k+1)/2

            k(k+1)/2    isInt                    assume general

            2(k+1)/2    isInt                    k isInt

            (k+1)(k+2)/2      isInt              QED
```

Existence Proof

An *existence proof* demonstrates that a particular object, or solution, does exist, but the proof does not identify exactly what that object is. Automated systems cannot conduct existence proofs.

Essentially, existence proofs demonstrate *universal principles*, whereas computational proofs demonstrate a *verification* of a particular principle.

Prove there exists a function that is both odd and even.

Odd function: `F[-x] = -F[x]` e.g. sine

Even function: `F[-x] = F[x]` e.g. cosine

Find an `F[x]` such that:

`E[F[R[0,x]], R[0,F[x]]]` and `E[F[R[0,x]],F[x]]`

where `E is =` and `R is -`

An example of such a function is: `F[x] = 0`

For pattern-matching, we are looking for two different matches to the second argument of `E`. Equivalently, we can eliminate `F[-x]` algebraically, so that we are looking for the single pattern:

```
F[x] = -F[x]

E[F[x], R[0,F[x]]]
```

From here, the example of `F[x]=0` is easy to identify, since `R[0,0]=0` and `E[0,0] isTrue`. In general, the problem is to show:

```
Exists F.   E[F[x],R[0,F[x]]]
```

One problem is that the existential quantifier is over *functions*, not variables. This requires *Second Order Logic*, for which theorem provers are not yet well developed. What is the *Domain* of all Functions? How do we enumerate, even recursively, all possible functions?

Verifying a Condition from a Search Program

Suppose that the inner loop of a search program is protected (defined) by the following *assertion* (also called a *guard*, or a *loop invariant*).

```
                                                          Line
(     j<i                                                  1
and   m<=p<=q<=n                                           2
and   All x,y. (m<=x<i and j<y<=n) implies G[x]<=G[y]      3
and   All x,y. m<=x<=y<=j implies G[x]<=G[y]               4
and   All x,y. i<=x<=y<=n implies G[x]<=G[y]               5
)
      implies G[p]<=G[q]                                   6
```

The above conjunctive constraint is an assertion about the state of a program. We have six database items being searched `{i,j,m,n,p,q}`, and a property `G`.

M and n are minimal and maximal bounds for the item ids.
I and j relative bounds between m and n.
P and q are arbitrary entries for which we want to determine
if their G properties are ordered.

We know the following:

(line 1) the bounding items j and i are ordered
(line 2) m,p,q,n are partially ordered

The constraints on the G property (lines 3,4 and 5) state that G is ordered in these cases:

(line 3) all items are bounded below between m and i and above between j and n
(line 4) ordered items are bounded below between m and j
(line 5) ordered items are bounded above between i and n

When all of the orderings are satisfied, the G property is assured to be ordered as well.

Without the modern language capability of including assertions within the code, these constraints must be proved independently (using a logic programming and verification system). The first step in a proof is to add defining axioms for the functions and relations used by the constraints. Here we need to add the ordering axiom

```
x<=y xor y<x
```

Next we form clauses out of multiple constraints on single lines (e.g. line 2)

```
m<=p
p<=q
q<=n
```

Now Skolemize to eliminate quantifiers. The labels {i,j,m,n,p,q} are Skolem constants and the labels {x,y} are universal variables. C9 is the conclusion.

```
C1.   x<=y xor y<x
C2.   j<i
C3.   m<=p
C4.   p<=q
C5.   q<=n
C6.   (m<=x and x<i and j<y and y<=n) implies (G[x]<=G[y])
C7.   (m<=x and x<=y and y<=j) implies (G[x]<=G[y])
C8.   (i<=x and x<=y and y<=n) implies (G[x]<=G[y])
C9.   G[p]<=G[q]
```

Here are the highlights of a backward chaining, natural deduction proof. The form of the conclusion suggests a strategy: immediately bind the universal variables to constants. We seek to find a set of ordering conditions that are subgoals of C6-C8, and that are satisfied by C1-C5.

```
 (1)  G[p]<=G[q]                                              ? C9
 (2)  (m<=p and p<i and j<q and q<=n) implies (G[p]<=G[q])    C6,(1)
 (3)  m<=p and p<i and j<q and q<=n                           ? (2)
 (4)  (m<=p and p<=q and q<=j) implies (G[p]<=G[q])           C7,(1)
 (5)  m<=p and p<=q and q<=j                                  ? (4)
 (6)  (i<=p and p<=q and q<=n) implies (G[p]<=G[q])           C8,(1)
 (7)  i<=p and p<=q and q<=n                                  ? (6)

 (8)  p<i and j<q and q<=n                                    ? (3),C3
 (9)  p<i and j<q                                             ? (8),C5
(10)  p<=q and q<=j                                           ? (5),C3
(11)  q<=j                                                    ? (10),C4
(12)  i<=p and q<=n                                           ? (7),C4
(13)  i<=p                                                    ? (12),C5
```

At this point, satisfying the equations on any one of lines (9), (11), or (13) achieves the goal. In this case, all three lines are needed for the proof, which demonstrates that the guard is not overly restrictive. The proof strategy is case analysis.

```
(14)  q<=j xor j<q                                            C1
(15)  i<=p xor p<i                                            C1
(16)  if q<=j then QED else j<q                               11,14
(17)  if i<=p then QED else p<i                               13,15
(18)  if not QED then (j<q and p<i)                           16,17
(19)  if (j<q and p<i) then QED                               9
(20)  QED in all cases                                        16-19
```

The natural logic in line (18) says: all I need is to satisfy either line (11) or line (13). If I don't succeed, it must be because both `j<q` and `p<i` hold.

Pattern-matching as Computation

Here is a simple example of using Resolution to conduct computation.

The *accumulating factorial function* as resolution clauses:

```
{Fac[0,1]}                                  base

{~Fac[x,r], Fac[S[x],*[S[x],r]}             general
```

Supporting clauses define the successor function and multiplication:

```
{~S[0], 1}

{~S[x], +1[x]}

{~*[x,1], x}

{~*[S[x],y], +[*[x,y],y]}
```

These clauses will need to be renamed with unique variable identifiers. Note that the Successor notation is designed for nested recursive pattern-matching. The numeral `3` is a shorthand notation for `S[S[S[0]]]`.

The *query* form, what is the Factorial of 3?

```
{~Fac[S[S[S[0]]], ANS]}
```

The resolution computation begins by matching the query to the general recursive part of the function definition:

match	`Fac[S[S[S[0]]],`	to	`Fac[S[x],`
binding	`x`	to	`S[S[0]]`
and binding	`ANS`	to	`*[S[x],r],` which is `*[S[S[S[0]]],r]]`

generating the new renamed clause:

```
{~Fac[S[S[0]],ANS1]}          ANS = *[S[S[S[0]]],ANS1]]
```

Using the general clause of the definition two more times yields these new clauses:

match	`Fac[S[S[0]],`	to	`Fac[S[x],`
binding	`x`	to	`S[0]`
and binding	`ANS1`	to	`*[S[x],r]` which is `*[S[S[0]],r]`

new renamed clause:

```
{~Fac[S[0],ANS2]}             ANS1 = *[S[S[0]],ANS2]
```

match	`Fac[S[0],`	to	`Fac[S[x],`
binding	`x`	to	`0`
and binding	`ANS2`	to	`*[S[x],r],` which is `*[S[0],r]`

new renamed clause:

```
{~Fac[0,ANS3]}                ANS2 = *[S[0],ANS3]
```

This last clause matches the base rule:

match	`ANS3`	to	`1`

new renamed clause (the termination of the computation):

```
{}
```

The result itself is embedded in the `ANS` variables, which require backwards substitution and arithmetic evaluation:

```
ANS3 = 1

ANS2 = *[S[0],ANS3] = *[S[0],1] = *[1,1] = 1

ANS1 = *[S[S[0]],ANS2] = *[S[S[0]],*[S[0],1]] = *[2,1] = 2

ANS = *[S[S[S[0]]],ANS1]] = *[S[S[S[0]]],*[S[S[0]],*[S[0],1]]]]
    = *[3,2] = 6
```

Representing a Procedural Program Fragment as a Resolution Pattern

Consider the following pseudo-code fragment for in-place sorting of the elements of an array `A`. The code moves elements indexed `i+1` to `j` one position earlier in the array. The question is to verify that the subscripts of `A` never go outside the bounds of `0` to `n` as the DO loop executes.

```
INTEGER A[n]
INTEGER k
...
k := i+1
DO WHILE (k <= j)
  A[k-1] := A[k]
  k := k+1
END DO
...
```

We assume that `i` and `j` are iteration variables that are ordered and bounded by the array size:

```
0<=i<=n
0<=j<=n
i<j
```

State (Simulating Memory)

The *binding list* (also called the *environment*) represents all variables in memory and their current value. We model it using a recursive nested notation which works well for pattern-matching. The arguments to the `State` function (technically `State` is not a function, it is just a labeled pattern) are the current variable name, its value, and another `State` pattern which contains the rest of the program variables and their bindings.

```
State[<variable>,<value>,State[<next-variable,<next-value>,State[...]]]
```

The *symbolic state* of the example program may look like this:

```
State[i,4,State[k,?,State[j,n,nil]]]
```

The question mark is used to mean “not-initialized”, i.e. the value of `k` is not known. The variable `i` is bound to the numerical value `4`, the variable `j` is bound to the symbolic value `n`.

Arrays

Arrays are slightly more complex that simple binding lists, since *array indexing* must be used for access. Lets assume that the environment is stored symbolically as an *array*. For pattern-matching, it is convenient to represent an array as a recursive nested form, with three arguments (index, value, and rest):

```
Array[<index>, <value>, Array[<next-index>,...]]
```

The pattern representation of an example array `A` follows:

variable	n	k	i	j	x
index	0	1	2	3	4
value	4	?	4	n	0

```
A[0,4,A[1,?,A[2,4,A[3,n,A[4,0,nil]]]]]
```

`Array` has the same pattern structure as `State`; they are both representations of memory. The index of an array serves the same purpose as the memory address used by the CPU, both are pointers to the symbolic name and its value. However, in simulation, the array is treated like a list; the time-savings of indexed access is a physical implementation detail rather than a symbolic quality.

To represent arrays as a `State` pattern, the array is simply treated like a compound variable:

```
State[<array-name>,<array>,State[...]]
```

In the example, the name of the array is `A`:

```
State[A, A[0,4,A[1,?,A[2,4,A[3,n,A[4,0,nil]]]]], State[k...]]
```

Execution (Simulating Process)

We need a representation of *execution*, the action upon the program statement currently being executed in the current binding environment that produces a computational step. In the resolution simulation, all actions are substitutions (unifications), essentially the same action that CPU registers use at the physical level. The simulation executes *symbolic* substitution, the CPU executes *physical* substitution.

```
Execute[<statement>,<environment>]
```

The initial `Execute` process, when we enter into the example program, would look like this:

```
Execute[assign1,
        State[A, A[0,4,A[1,?,A[2,4,A[3,n,A[4,0,nil]]]]],
         State[n,4,
          State[k,?,
           State[i,2,
            State[j,n,
             State[x,0,nil]]]]]]
```

Program Statements

We now provide a general pattern, or clause, for each type of program statement. Each statement type will have the abstract form:

```
<Statement-type>[<statement-name>,<action>,<next-statement>]
```

This pattern identifies the statement *type* by the outer label, and the specific instance of that type by the first argument. `Action` is the unique transformation which defines the statement type. `Next-statement` achieves sequential actions, it is equivalent to the sequence operator "`;`" in procedural languages such as C, and specifies the next program statement in sequence.

The evaluation strategy for program statements in general has the form:

> If the computation is at statement P1 and at state S1
> then execute statement P2 with resulting new state S2.

Environment `S2` is the result of processing statement `P1` in the context of environment `S1`. `S2` is `S1` with changes introduced by `P1`. An abstract example of process would look like this:

```
If Execute[P1,S1] and <Statement-type>[P1,<action>,P2],
      then Execute[P2,<new-state-changed-by-statement>]
```

Variable Declarations

Rather than a conditional execution statement, the initializations included in variable and array declarations are treated directly as equality assertions for match-and-substitute.

```
INTEGER k  INIT:0

Equal[k,0]

INTEGER A[n]

Equal[HighBound[A],n]
```

`HighBound` is the maximal array size of array `A`.

Assignment

The procedural form of assignment translates into a declarative, resolution-based template for the assignment computation. The form of assignment is

```
NEW := OLD

Assign[<statement>,<new>,<old>,<next-statement>]
```

Example: `k := i+1` `assign1`

```
Assign[assign1,k,+[i,1],dowhile1]
```

The *meaning* of the `Assign` pattern is provided through its execution/evaluation template

```
If Execute[P1,S1] and Assign[P1,<new>,<old>,P2],
      then Execute[P2,Save[<new>,Eval[<old>,S1],S1]]
```

This says that if the current program statement `P1` is an `Assign`, then move forward to the next program statement `P2`, by asserting that `P1` has been processed and the result of that processing has changed the current state `S1` into the next state `S2`.

The new patterns, `Save` and `Eval`, manage memory storage and ground evaluation respectively, implementing the match-and-substitute process. `Eval` binds any variables in `<old>` to their current values stored in `S1`. `Save` substitutes the result of `Eval` into the value field of the variable `<new>` in `S1`, constructing `S2` in the process.

Expressed as a resolution clause, the `Assign` execution template looks like:

```
{ ~Execute[P1,S1],
  ~Assign[P1,new,old,P2],
   Execute[P2, Save[new,Eval[old,S1],S1]] }
```

In the example, we are entering the fragment with the program statement `assign1`, binding the new `k` to the old `i+1`, and preparing for the following program statement `dowhile1`.

```
{ Assign[assign1,k,+[i,1],dowhile1] }
```

Resolving, bind

```
P1    to    assign1
new   to    k
old   to    +[i,1]
P2    to    dowhile1
```

New clause:

```
{ ~Execute[fragment-entry,S1],
   Execute[dowhile1, Save[k,Eval[+[i,1],S1],S1]] }
```

`Eval[+[i,1],S1]` finds `i` in `S1` and binds it to its value (which is `4` in the example). Then `+[4,1]` is evaluated to `5`.

`Save[k,5,S1]` stores `5` as the value of variable `k` in `S1`, constructing `S2` in the process.

Note that both `Eval` and `Save` achieve their functionality also through pattern-matching and substitution, using resolution. Details have been omitted.

DoWhile

The `DoWhile` program statement evaluates a test in the context of a binding environment, branching to `P2` or `P3` depending on the outcome of the test.

```
DO WHILE <test> <body> END

DoWhile[P1,<test>,P2,P3]

END DO

DoEnd[P1,P2]
```

Example:

```
DO WHILE (k <= j)              dowhile1
  A[k-1] := A[k]               assign2
  k := k+1                     assign3
END DO                         doend1
...                            fragment-exit
```

```
DoWhile[dowhile1,<=[k,j],assign2,doend1]

DoEnd[doend1,fragment-exit]
```

When the test succeeds, the next program statement *inside* the DO body, `P2`, is queued for execution. When the test fails, the next program statement *outside* the DO body, `P3`, is queued. The execution template for DOWHILE:

```
If Execute[P1,S1] and DoWhile[P1,test,P2,P3]
      then Execute[If[Eval[test,S1],P2,P3],S1]
```

The semantics of DOWHILE is to `Eval test`, and branch using if-then-else to either `P2` or `P3`. The `DoEnd` statement terminates the DOWHILE by moving `Execute` to the next program statement.

```
If Execute[P1,S1] and DoEnd[P1,P2]
      then Execute[P2,S1]
```

Finally, the looping process of DOWHILE, when the exit test fails, is achieved simply by encoding the <next-statement> field of the last statement inside the DO body to be the start of the DOWHILE loop:

```
Assign[assign3,k,+[k,1],dowhile1]
```

Errors

In order to answer questions about array out-of-bounds, we need to treat errors as legitimate program statements. (This need is also reflected in the evolution from C to Java, in that Java incorporates errors as first-class program entities.)

In procedural programming languages which do not support error checking through assertions, the checks must be written directly into the program:

```
INTEGER A[n]
INTEGER k
...
k := i+1
DO WHILE (k <= j)
IF (k-1 > HighBound[A]) ERROR
IF (1 > k-1) ERROR
IF (k > HighBound[A]) ERROR
IF (1 > k) ERROR
  A[k-1] := A[k]
  k := k+1
END DO
...
```

This approach is an obsoleted programming style, in that the ERROR halts the program flow. A more modern approach would be use If-Then-Else to incorporate the error branch directly into the program flow:

```
If[<test>,<throw-to-error-handler>,<next-statement>]
```

The `IfError` statement has the following form:

```
IF <test> ERROR

IfError[P1,<test>,P2]
```

Example: `IF (1 > k-1) ERROR` `iferr2`

```
      IfError[iferr2,>[1,-[k,1]],iferr3]
```

The execution template for `IfError` is identical to that of `If-Then-Else`:

```
if Execute[P1,S1] and IfError[P1,test,P2]
      then Execute[If[Eval[test,S1],Error,P2],S1]
```

The Transcribed Program Fragment

Finally, all the program statement patterns can be assembled into a resolution database. The program statement templates come first, followed by the program itself. Finally the initial `Execute` clause triggers the computational process; the end result is stored in the `State` clause which remains after all executions have been resolved.

Note that all variables in the statement templates have been given unique names. However, what we call program variables {`k,i,j,A`} are symbolic constant labels here (Skolem constants), since they identify an arbitrary but not specific value. Statement names are also constants. Statement constants are carried without change throughout the set of clauses representing the program.

```
{ ~Execute[P1,S1],
  ~Assign[P1,new,old,P2],
   Execute[P2, Save[new,Eval[old,S1],S1]] }

{ ~Execute[P3,S2],
  ~DoWhile[P3,test,P4,P5],
  Execute[If[Eval[test,S2],P4,P5],S2] }

{ ~Execute[P6,S3],
  ~DoEnd[P6,P7],
  Execute[P7,S3] }

{ ~Execute[P8,S4],
  ~IfError[P8,errtest,P9],
  Execute[If[Eval[errtest,S4],Error,P9],S4] }


{ Equal[ HighBound[A],n] }

{ Equal[arrayA, A[0,4,A[1,?,A[2,4,A[3,n,A[4,0,nil]]]]] }


{ Assign[assign1,k,+[i,1],dowhile1] }

{ DoWhile[dowhile1,<=[k,j],iferr1,fragment-exit] }

{ IfError[iferr1,>[-[k,1],HighBound[A]],iferr2] }

{ IfError[iferr2,>[1,-[k,1]],iferr3] }

{ IfError[iferr3,>[k,HighBound[A]],iferr4] }

{ IfError[iferr4,>[1,k],assign2] }

{ Assign[assign2,A[i1,-[k,1]],A[i2,k],assign3] }

{ Assign[assign3,k,+[k,1],dowhile1] }

{ DoEnd[doend1,fragment-exit] }


{ Execute[assign1,
          State[A, arrayA,
           State[n,4,
            State[k,?,
             State[i,2,
              State[j,n,
               State[x,0,nil]]]]]] }
```