Deduction, Strategies, and Rewriting

S. Eker[1], N. Martí-Oliet[2], J. Meseguer[3], and A. Verdejo[2]

[1] Computer Science Lab, SRI International, Menlo Park, USA

[2] Facultad de Informática, UCM, Madrid, Spain

[3] Department of Computer Science, UIUC, Urbana-Champaign, USA

Strategies 2006

Introduction

Question: Why are rewriting strategies relevant to automated deduction?

Key idea # 1: any inference rule is a (possibly conditional) rewrite rule.

Example:

$$\text{orient} \quad \frac{(K, E \cup \{t \approx c\}, R)}{(K, E, R \cup \{t \to c\})} \quad \text{if } t \succ c$$

corresponds to

$$\text{orient} : (K, E\ t = c, R) \longrightarrow (K, E, R\ t \to c) \quad \text{if} \quad t \succ c$$

Main consequence: Rewriting logic is a good logical framework to represent logics and inference systems.

Introduction

Key idea # 2.1: Automated deduction methods (e.g. congruence closure, resolution, etc.) should be specified, not procedurally, as low level algorithms (pointer manipulation, etc.), but declaratively as inference systems which are proved correct regardless of implementation details.

Key idea # 2.2: Specific algorithms to implement a given inference systems should be specified at a high-level as strategies to apply the inference rules.
For example, Shostak's and Downey-Sethi-Tarjan's congruence closure algorithms as strategies to apply congruence closure inference rules (Tiwari, 2000).

Introduction

- This abstract methodology can be and has been applied not just to isolated examples, but to a very wide range of automated deduction methods, e.g.,
  - Unification (Martelly & Montanari, Jouannaud & Kirchner)
  - Congruence closure (Kapur, Tiwari)
  - Nelson-Oppen combination of decision procedures (Tiwari, Conchon & Krstic)
  - Knuth-Bendix completion (Bachmair, Dershowitz & Kirchner, Lescanne)
  - Inductive theorem proving and other Maude tools (Clavel, Durán & Meseguer)

Key idea # 3: Using a high-performance rewriting language implementation, and a strategy language to guide rewriting computations, the above high-level distinction between inference rules and strategies can be directly implemented, providing a way to faithfully represent the high-level specification as an executable prototype or implementation.

Semantics and Pragmatics of Strategy Languages

- Semantics is needed to:
  - ensure that all computations allowed are correct deductions
  - as formal specification of correct implementations

- Pragmatic considerations are important to guide strategy language designs that can deal well with relevant applications.

Our Approach

1. Build a strategy language on top of a rewriting logic language like Maude.
2. Maintain total separation between rules in rewrite theories and strategies in strategy modules.
3. Learn from previous experience in strategy languages in Maude, ELAN, and Stratego, and focus on automated deduction and programming language semantics applications.
4. Develop the notion of generic strategies (e.g. backtracking, map, etc.) applicable not to a single rewrite theory, but to a wide range of rewrite theories.

Our Approach

5. Simple set-theoretic semantics for the strategies of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$:
$$_@_ : Strat \times T_{\Sigma}(X) \longrightarrow \mathcal{P}(T_{\Sigma}(X))$$
6. More concrete reflective semantics = Prototype implementation: strategies defined by (equational) rewrite rules at the meta-level.
7. Implementation in the Maude rewrite engine after sufficient experimentation and mature language design (ongoing).

Rewriting logic in a nutshell

- A rewrite theory (Σ, E, R) consists of a signature Σ, a set E of equations, and a set R of rules.
  - The static part of a system or logic is specified in an equational sublogic of rewriting logic (membership equational logic) by means of the equations E.
  - The system dynamics (transitions or inferences) is specified by means of possibly conditional rules R that rewrite terms, representing parts of the system, into others.
- Maude is an efficient implementation of rewriting logic.
- Maude syntax is user-definable and operators can have attributes like associativity (`assoc`), commutativity (`comm`), and identity (`id:`).

Maude system modules

- The general form of a rewrite rule in Maude is the following:

$$t \longrightarrow t' \;\; \textit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

- Rewrite conditions require search to check their satisfiability.
- Maude system modules specify rewrite theories, e.g.,

```
mod SORTING is
  protecting NAT .
  sorts Pair PairSet .
  subsort Pair < PairSet .
  op (_,_) : Nat Nat -> Pair .
  op empty : -> PairSet .
  op __ : PairSet PairSet -> PairSet [assoc comm id: empty] .
  op length : PairSet -> Nat .
  vars I J V W : Nat .  var PS : PairSet .
  eq length(empty) = 0 .
  eq length((I, V) PS ) = length(PS) + 1 .
  rl [switch] : (J, V) (I, W) => (J, W) (I, V) .
endm
```

The strategy language: Basic strategies

Idle and fail are the simplest strategies.

Basic strategies Application of a rule (identified by the corresponding rule label) to a given term (possibly with variable instantiation).

```
subsort Label < BasicStrat < Strat .
op _[_] : Label Substitution -> BasicStrat .
```

The rule is applied anywhere in the term where it matches satisfying its condition.

Top For restricting the application of a rule just to the top of the term.

```
op top : BasicStrat  -> Strat .
```

The strategy language: Tests and combinators

Tests are strategies that test some property of a given state term, based on matching.

```
subsort Test < Strat .
op xmatch_s.t._ : Term EqCondition -> Test .
op match_s.t._ : Term EqCondition -> Test .
```

`xmatch T s.t. C` is a test that, when applied to a given state term `T'`, succeeds if there is a subterm of `T'` that matches the pattern `T` and then the condition `C` is satisfied, and fails otherwise.

`match` works in the same way, but only at the top.

Regular expressions

```
op _;_ : Strat Strat -> Strat [assoc] .      *** concatenation
op _|_ : Strat Strat -> Strat [assoc comm] . *** union
op _* : Strat -> Strat .                     *** iteration (0 or more)
op _+ : Strat -> Strat .                     *** iteration (1 or more)
```

The strategy language: Conditional

If-then-else is a typical if-then-else, but generalized so that the first argument is also a strategy.

```
op if_then_else_fi : Strat Strat Strat -> Strat .
```

Using the if-then-else combinator, we can define many other useful strategy combinators as derived operations.

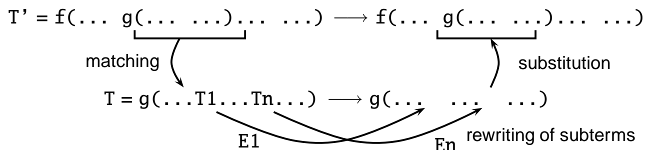
```
op _orelse_ : Strat Strat -> Strat .
eq E orelse E' = if E then idle else E' fi .
op not : Strat -> Strat .
eq not(E) = if E then fail else idle fi .
op _! : Strat -> Strat .
eq E ! = E * ; not(E) .
op try : Strat -> Strat .
eq try(E) = if E then idle else idle fi .
op test : Strat -> Strat .
eq test(E) = if not(E) then fail else idle fi .
```

The strategy language: Decomposition

Strategies applied to subterms with the (x)`matchrew` combinator control the way different subterms of a given state are rewritten.

```
sort TermStrat .
op _using_ : Term Strat -> TermStrat .
op xmatchrew_s.t._by_ : Term EqCondition List(TermStrat) -> Strat .
op matchrew_s.t._by_  : Term EqCondition List(TermStrat) -> Strat .
```

`xmatchrew T s.t. C by T1 using E1, ..., Tn using En` applied to a state term `T'`:



The strategy language: Recursive definitions

Recursion is achieved by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies. For example,

```
strat solve @ X$State .
var S : X$State .
stratdef solve :=
   if match S s.t. isSolution(S)
   then  idle
   else  expand ;
         match S s.t. isOk(S) ;
         solve
   fi .
```

Moreover, recursive strategies can have data arguments: values in the rewrite theory on which the strategies appy can be used as arguments.

The strategy language: Strategy modules and commands

- The user can write one or more strategy modules to define strategies for a system module M.

```
smod STRAT is
  protecting M .
  including STRAT1 .    ...    including STRATp .
  strat E1 @ K1 .
  stratdef E1 := Exp1 .
     ...
  strat En @ Kn .
  stratdef En := Expn .
endsd
```

- The command

```
   srew T using E .
```

rewrites the term T using a strategy expression E.

Examples: Blackboard

- A simple game: A blackboard with several natural numbers written on it. A legal move consists in selecting two numbers in the blackboard, removing them, and writing their arithmetic mean.
- The objective of the game is to get the greatest possible number written on the blackboard at the end.

```
mod BLACKBOARD is
  protecting NAT .
  sort BB .
  subsort Nat < BB .
  op __ : BB BB -> BB [assoc comm] .
  vars M N : Nat .
  rl [play] : M N => (M + N) quo 2 .
endm
```

Examples: Blackboard

- A player can choose the numbers randomly, or can follow some strategy. Possible strategies consist in taking always the two greatest numbers, or the two smallest, or taking the greatest and the smallest.

```
smod BB-STRAT is
  including EXT-BB .  *** extends BLACKBOARD with max, min and remove
  strat maxmin on BB .
  stratdef maxmin := (matchrew B s.t. X := max(B) /\ Y := min(B) by
                         B using play[M <- X ; N <- Y] ) ! .
  strat maxmax on BB .
  stratdef maxmax := (matchrew B s.t. X := max(B) /\ Y := max(remove(X,B)) by
                          B using play[M <- X ; N <- Y] ) ! .
  strat minmin on BB .
  stratdef minmin := (matchrew B s.t. X := min(B) /\ Y := min(remove(X,B)) by
                          B using play[M <- X ; N <- Y] ) ! .
endsm
```

```
Maude> srew 2000 20 2 200 10 50 using maxmin .
result NzNat :  178
Maude> srew 2000 20 2 200 10 50 using minmin .
result NzNat :  1057
```

Examples: Insertion sort

- Arrays are represented as sets of pairs and there is a rule to switch the values in two positions of the array.

```
mod SORTING is
  protecting NAT .
  sorts Pair PairSet .
  subsort Pair < PairSet .
  op (_,_) : Nat Nat -> Pair .
  op empty : -> PairSet .
  op __ : PairSet PairSet -> PairSet [assoc comm id: empty] .
  op length : PairSet -> Nat .
  vars I J V W : Nat .  var PS : PairSet .
  eq length(empty) = 0 .
  eq length((I, V) PS ) = length(PS) + 1 .
  rl [switch] : (J, V) (I, W) => (J, W) (I, V) .
endm
```

Examples: Insertion sort

$Y := 2$
while $Y \leq N$ do
  $X := Y$
  while $X > 1 \ \wedge \ V[X-1] > V[X]$ do
    switch $V[X-1]$ and $V[X]$
    $X := X - 1$
  $Y := Y + 1$

```
smod INSERTION-SORT-STRAT is
  protecting SORTING .

  strat insort : Nat on PairSet .
  stratdef insort(Y) := try(match PS s.t. Y <= length(PS) ;
                            insert(Y) ;
                            insort(Y + 1)) .

  strat insert : Nat on PairSet .
  stratdef insert(X) := idle if X == 1 .
  stratdef insert(X) := try(xmatch (sd(X,1), V) (X, W) s.t. V > W ;
                            switch[J <- sd(X,1) ; I <- X] ;
                            insert(sd(X,1)))
                        if X > 1 .
endsm
```

Examples: Generic backtracking strategy

- Backtracking is a generic strategy useful for solving a problem.

```
sth BT-ELEMS is
  sort State .
  op isOk : State -> Bool .
  op isSolution : State -> Bool .
  strat expand on State .
endsth

smod BT-STRAT{X :: BT-ELEMS} is
  var S : X$State .
  strat solve on X$State .
  stratdef solve :=
     if match S s.t. isSolution(S)
     then  idle
     else  expand ;
           match S s.t. isOk(S) ;
           solve
     fi .
endsm
```

Backtracking instantiation: Labyrinth

```
fmod POSITIONS is
  protecting NAT .
  sort Pos .
  op [_,_] : Nat Nat -> Pos .
endfm

mod LABYRINTH is
  protecting LIST{Pos} .

  op contains : List{Pos} Pos -> Bool .
  ops isSolution isOk : List{Pos} -> Bool .
  op next : List{Pos} -> Pos .
  op wall : -> List{Pos} .
  vars X Y : Nat .  var P Q : Pos .  var L : List{Pos} .

  eq isSolution(L [8,8]) = true .
  eq isSolution(L) = false [owise] .
  eq contains(nil, P) = false .
  eq contains(Q L, P) = if P == Q then true else contains(L, P) fi .
  eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 8 and Y <= 8
                     and not(contains(L, [X,Y])) and
                         not(contains(wall, [X,Y])) .
```

Backtracking instantiation: Labyrinth

```
crl [extend] : L => L P if next(L) => P .

rl [next] : next(L [X,Y]) => [X + 1, Y] .
rl [next] : next(L [X,Y]) => [X, Y + 1] .
rl [next] : next(L [X,Y]) => [sd(X, 1), Y] .
rl [next] : next(L [X,Y]) => [X, sd(Y, 1)] .

eq wall =       [2,1]
                [2,2]
                [2,3]       [4,3] [5,3] [6,3] [7,3] [8,3]

          [1,5] [2,5] [3,5] [4,5]             [7,5]
                                              [7,6]
                                              [7,7]
                                              [7,8]       .
endm
```

Backtracking instantiation: Labyrinth

```
smod LABYRINTH-STRAT is
  protecting LABYRINTH .
  strat expand on List{Pos} .
  stratdef expand := top(extend{dfs(next)}) .
endsm

sview LABYRINTH-BT-ELEM from BT-ELEMS to LABYRINTH-STRAT is
  sort State to List{Pos} .
endsv

smod LABYRINTH-BT-STRAT is
  including BT-STRAT{LABYRINTH-BT-ELEM} .
endsm
```

Case study: Abstract congruence closure

- Congruence closure is a decision procedure for the word problem associated with a finite set of ground equations.
- It also provides a decision procedure for the theory of uninterpreted function symbols.
- We present the abstract formulation of congruence closure given by Tiwari, which allows a high-level proof of its correctness.
- We then show how:
  - the inference rules are rewrite rules;
  - the Shostak and Downey-Sethi-Tarjan algorithms can be specified as strategies in our strategy language;
  - prototypes for them are then obtained in Maude.

Congruence closure inference rules (Tiwari)

Extension $$\frac{(K, E[t], R)}{(K \cup \{c\}, E[c], R \cup \{t \to c\})}$$ if $t \to c$ is a D-rule

Simplification $$\frac{(K, E[t], R \cup \{t \to c\})}{(K, E[c], R \cup \{t \to c\})}$$

Orientation $$\frac{(K \cup \{c\}, E \cup \{t \approx c\}, R)}{(K \cup \{c\}, E, R \cup \{t \to c\})}$$ if $t \succ c$

Deletion $$\frac{(K, E \cup \{t \approx t\}, R)}{(K, E, R)}$$

Deduction $$\frac{(K, E, R \cup \{t \to c, t \to d\}}{(K, E \cup \{c \approx d\}, R \cup \{t \to d\})}$$

Collapse $$\frac{(K, E, R \cup \{s[t] \to d, t \to c\}}{(K, E, R \cup \{s[c] \to d, t \to c\}}$$ if $s \not\equiv t$

Representation in Maude

```
crl [Ext] : < K, E u = v, R >
         => < K + 1, E u' = v, R t -> c >
         if subterm(u) => t                       *** u[t]
         /\ isOpConstants?(t)                     ***  t -> c is a D-rule
         /\ c := c(K)                             *** new constant
         /\ u' := subst(u,t,c) .

crl [Sim] : < K, E u = v, R t -> c >
         => < K, E u' = v, R t -> c >
         if subterm(u) => t
         /\ u' := subst(u,t,c) .

crl [Ori] : < K, E t = c, R >
         => < K, E, R t -> c >
         if t > c .
```

Representation in Maude

```
rl [Del] : < K, E t = t, R >
        => < K, E, R > .

crl [Ded] : < K, E, R t -> c t -> d >
        => < K, E, R c -> d t -> d >
        if c > d .

crl [Col] : < K, E, R u -> d t -> c >
        => < K, E, R u' -> d t -> c >
        if subterm(u) => t
        /\ t =/= u                   *** proper subterm
        /\ u' := subst(u,t,c) .

rl [Com] : < K, E, R t -> c c -> d >
        => < K, E, R t -> d c -> d > .
```

Congruence closure methods

- Shostak's algorithm

$$\mathbf{Shos} = ((\mathbf{Sim}^* \circ \mathbf{Ext}^*)^* \circ (\mathbf{Del} \cup \mathbf{Ori}) \circ (\mathbf{Col} \circ \mathbf{Ded}^*)^*)^*$$

```
stratdef Shos1 := (test(Sim | Ext) ; Sim ! ; Ext !)! ;
                  try( Del | Ori ) ;
                  (test(Col | Ded) ; try(Col) ; Ded !)!  .

stratdef Shos := matchrew S:State by
                    S:State using Shos1 ;
                                  (match S:State orelse Shos) .
```

Congruence closure methods

- Downey-Sethi-Tarjan algorithm

$$\mathbf{DST} = ((\mathbf{Col} \circ (\mathbf{Ded} \cup \{\epsilon\}))^* \circ (\mathbf{Sim}^* \circ (\mathbf{Del} \cup \mathbf{Ori}))^*)^*$$

```
stratdef start := (Ext ; Sim !)! ; Ori ! .

stratdef DST1 := (test(Col | Ded) ; try(Col) ; try(Ded))! ;
                 ( Sim ! ; (Del | Ori))! .

stratdef DST = matchrew S:State by
                  S:State using DST1 ;
                                (match S:State orelse DST) .
```

Execution example

```
Maude> (srew < 0, 'a.S = 'b.S
                  'f['f['a.S]] = 'f['b.S], mtRlS >
        using start ; DST .)

result State :
        < 5, mtEqS, 'a.S -> c(0)
                    'b.S -> c(1)
                    c(0) -> c(1)
                    c(2) -> c(4)
                    c(3) -> c(4)
                    'f[c(1)] -> c(3)
                    'f[c(4)] -> c(4) >
```

Case study: Knuth-Bendix completion

P. Lescanne. Completion Procedures as Transition Rules + Control. In *Proceedings TAPSOFT'89*, LNCS 351, pages 28–41. Springer, 1989.

- Several algorithms for completion are presented using the CAML functional programming language.
- The main differences among the algorithms are
  - the data structure on which the transition rules operate, and
  - the control that describes the way the transitions rules are invoked.
- We have redone this work in Maude by using its strategy language:
  - the data structure is the term being rewritten,
  - transition rules are represented as rewrite rules, and
  - the control is represented as strategies applying the rules in a directed way.

Knuth-Bendix completion algorithms

- We have considered three KB-completion algorithms, as described by Lescanne:
  - N-Completion,
  - S-Completion,
  - ANS-Completion.
- Each algorithm is a refinement of the previous one, obtained by:
  - adding more components to the data structure,
  - adapting the transition rules, and
  - changing the control to make the algorithm more efficient.

Other applications of the strategy language

- Implementation of Cardelli and Gordon's ambient calculus:

> F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed Mobile Ambients in Maude. In H. Cirstea and N. Martí-Oliet (eds.), *6th International Workshop on Rule-Based Programming, RULE 2005*, ENTCS 147(1), 2006.

- Implementation of parallel functional language EDEN:

> M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and its strategies for defining a framework for analyzing Eden semantics. In *The Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS'06*.

- Implementation of Modular Structural Operational Semantics:

> C. Braga and A. Verdejo. Modular Structural Operational Semantics with Strategies. In *Structural Operational Semantics 2006, SOS'06*.

Prototype implementation in Maude

- Using the Maude metalevel, we have implemented a prototype of the strategy language as an extension of Full Maude.
- It consists of several functions that work with a labelled version of the conceptual computation tree produced when applying a strategy `E` to a given state term `T`.
- Internal nodes are labelled with enough information to know which is the next child to be explored (the specific information saved depends on the top constructor in the strategy expression).
- The two main functions are `first` and `next`. The combination of these two functions serves to find all the solutions for the application of a strategy to a given state term.
- The metalanguage features of Maude allowed us to complete the prototype with a user interface where strategy modules can be loaded, and commands can be executed. These commands allow a step-by-step generation of all the possible results of rewriting a term using a strategy.

Core Maude implementation challenges

- In attempting to avoid becoming trapped in failing (but infinite) parts of the search tree, it is desirable to explore many parts of the search tree in parallel.
- Strategy combinators such as `if s1 then s2 else s3 fi` require detection that all attempts to rewrite with `s1` have failed.
- Strategy combinators such as `test(s)` and `not(s)` are only interested in the first solution if one exists, and after a first solution has been obtained, it is desirable from an efficiency standpoint to curtail efforts to find a second solution.
- A conditional rule having a rewrite condition fragment necessitates starting a fresh search procedure for that rewrite condition fragment, however, this could be a “black hole” and needs to be interleaved with alternatives at the parent level.

Sketch of Core Maude implementation solution

- Seaching is done in parallel using a queue of virtual processes on a round-robin scheme.
- Each process works on behalf of a single task.
- A process keeps track of its current term, remaining strategy expression and task, together with other information as needed.
- A process can fork off new processes and tasks as needed.
- Tasks act as the "fan-in" counterpart to processes.
- A task works on behalf of a single "parent" task and keeps track of processes and tasks working for it, together with other information as needed.
- A task gets to run when one of its processes or tasks successfully completes or when its last process or task fails.
- Thus a task can detect failure in a given (finite) search subtree and can also terminate all computation for a given seach subtree.
- The original `srewrite` command acts as the toplevel task and any successful completetions reported to it are solutions to the `srewrite`.

Conclusions

- Have put forward three key ideas:
  1. Rewriting logic is a logical framework to represent logics and inference systems.
  2. Automated deduction methods should be specified by:
     - inference systems and
     - different strategies to apply inference rules.
  3. Automated deduction systems can be prototyped/implemented at a very high level in a rewriting logic language having a strategy language.

Conclusions

- Have also presented a specific strategy language for Maude, illustrating its use in automated deduction.
- The particular strategy language design is not essential. The essential point is that the vision put forward in points 1-3 can be effectively carried out in Maude and in other rewriting languages.
- An interesting future research directions is investigating distributed specifications and implementations of a strategy language, since this offers the promise of more efficient and scalable executions, also in automated deduction.

Examples: CCS operational semantics

- The module below contains the CCS semantics representation without *tricks*.

```
mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .
  sort ActProcess .  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess .
  var L : Label .  var X : ProcessId .  vars P P' Q Q' : Process .
  var A : Act .    var AP : ActProcess .
  rl  [prefix] : A . P => {A}P .
  crl [sum]   : P + Q => {A}P' if P => {A}P' .
  crl [par1] : P | Q => {A}(P' | Q) if P => {A}P' .
  crl [par2] : P | Q => {tau}(P' | Q') if P => {L}P' /\ Q => {~ L}Q' .
  crl [res]   : P \ L => {A}(P' \ L) if P => {A}P' /\ A =/= L /\ A =/= ~ L .
  crl [def]   : X => {A}P if (X definedIn context) /\ def(X,context) => {A}P .
*** transitive closure
  crl [more] : P => {A}AP if P => {A}Q /\ Q => AP .
endm
```

- The first six rules correspond to CCS semantic rules. Rule `more` represents the transitive closure of the CCS transition relation.

Examples: CCS operational semantics

- We have two kinds of transitions, $\longrightarrow$ and $\longrightarrow^*$. But they are represented in Maude by using the same rewrite relation (=>).
- We need to control which rules are used when solving the rewrite conditions in rule `more` above.

```
smod CCS-STRAT is
  protecting CCS-SEMANTICS .

  strat ccs on ActProcess .
  stratdef ccs := top(prefix) |
                  top(sum{dfs(ccs)}) |
                  top(par1{dfs(ccs)}) |
                  top(par2{dfs(ccs) dfs(ccs)}) |
                  top(res{dfs(ccs)}) |
                  top(def{dfs(ccs)}) .

  strat trans on ActProcess .
  stratdef trans := idle | top(more{dfs(ccs) dfs(trans)}) .
endsm
```

Examples: Map, parameterized strategy module

- A strategy $map(S)$, that applies a strategy S once to every element in a list.

```
sth ELEM-STRAT is
  sort Elem .
  strat elem-strat on Elem .
endsth

sth ELEM-STRAT-4-STRUCT is
  inc ELEM-STRAT .
  sort Struct .
  subsort Elem < Struct .
  op _._ : Struct Struct -> Struct .
  op mt : -> Struct .
endsth

smod MAP{X :: ELEM-STRAT-4-STRUCT} is
  strat map on X$Struct .
  var E : X$Elem .  vars S S' : X$Struct .
  stratdef map := match mt
                  orelse match E ; elem-strat
                  orelse matchrew S . S' by
                               S' using map,
                               S using map .
endsm
```

Examples: Map, a concrete instantiation

```
fmod BIN-TREE{X :: TRIV} is
  sort Tree{X} .
  subsort X$Elem < Tree{X} .
  op mt : -> Tree{X} .
  op _^_ : Tree{X} Tree{X} -> Tree{X} .
endfm

mod DOUBLE is
  protecting BIN-TREE{Nat} .
  var N : Nat .
  rl [double] : N => 2 * N .
endm

sview DOUBLE-STRAT from ELEM-STRAT-4-STRUCT to DOUBLE is
  strat elem-strat to term double .
  sort Elem to Nat .
  sort Struct to Tree{Nat} .
  op _._ to _^_ .
endsv

smod DOUBLE-TREE-STRAT is
  including MAP{DOUBLE-STRAT} .
endsm
```

Completion inference rules

Deduce	$$\frac{E, R}{E \cup \{s \approx t\}, R}$$	if $s \leftarrow_R u \rightarrow_R t$
Orient	$$\frac{E \cup \{s \dot{\approx} t\}, R}{E, R \cup \{s \rightarrow t\}}$$	if $s > t$
Delete	$$\frac{E \cup \{s \approx s\}, R}{E, R}$$	
Simplify-identity	$$\frac{E \cup \{s \dot{\approx} t\}, R}{E \cup \{u \approx t\}, R}$$	if $s \rightarrow_R u$
R-Simplify-rule	$$\frac{E, R \cup \{s \rightarrow t\}}{E, R \cup \{s \rightarrow u\}}$$	if $t \rightarrow_R u$
L-Simplify-rule	$$\frac{E, R \cup \{s \rightarrow t\}}{E \cup \{u \approx t\}, R}$$	if $s \overset{\sqsupset}{\rightarrow}_R u$

Completion algorithms

- We consider three algorithms, as described by Lescanne:
  - N-Completion,
  - S-Completion,
  - ANS-Completion.
- Each algorithm is a refinement of the previous one, obtained by
  - adding more components to the data structure,
  - adapting the transition rules, and
  - changing the control with the idea of making the algorithm more efficient.
- In this presentation we do not deal with the two unfailing completion algorithms presented by Lescanne.

N-completion

- A first improvement of the overly abstract inference rules in order to take the computation of critical pairs into account.
- The data structure has three components:
  - E is a set of identities,
    either given identities or computed critical pairs,
  - T is a set of rules
    whose critical pairs have not been computed yet, and
  - R is another set of rules
    whose critical pairs have been computed (marked rules).

N-completion: Rules

```
rl [Deduce] : < R, T, E > => < R, T, E s =. t > . *** if  s <-- u --> t

crl [Orient] : < R, T, E s =. t > => < R, T s -> t, E >  if s > t .

rl [Delete] : < R, T, E s =. s > => < R, T, E > .

crl [Simplify] : < R, T, E s =. t > => < R, T, E u =. t >
 if u := reduce(s, R T) .

crl [R-Simplify] : < R s -> t, T, E > => < R s -> u, T, E >
 if u := reduce(t, R T) .

crl [L-Simplify] : < R s -> t, T, E > => < R, T, E u =. t >
 if u := reduce>(s -> t, R T) .

rl [move] : < R, r T, E > => < r R, T, E > .
```

N-completion: Strategies

```
seq COMP =  success orelse deduce orelse orient .

seq success = match ( < R, mtRlS, mtEqS > ) .

seq deduce = match (< R, r T, mtEqS >) ;
             deduction ;
             simplify-rules ;
             COMP .

seq deduction = matchrew < R, r' T, E > by
                  < R, r' T, E > using (add-crit-pairs(CP(r', R r')) ;
                                        move[r <- r']) .

seq add-crit-pairs(mtEqS) = idle .
seq add-crit-pairs(s1 =. t1 E) = Deduce[s <- s1 ; t <- t1] ; add-crit-pairs(E) .

seq simplify-rules = (L-Simplify | R-Simplify) ! .

seq orient = match ( < R, T, e E > ) ;
             simplify-eqs ;
             ( (match ( < R, T, mtEqS > ) ; COMP)
             orelse (Orient ; COMP) ).

seq simplify-eqs = (Delete | Simplify) ! .
```

N-completion: Example

```
*** g > h > f > a

 op eqs : -> EqS .
 eq eqs = ('h['x:S, 'y:S] =. 'f['x:S]
           'h['x:S, 'y:S] =. 'f['y:S]
           'g['x:S, 'y:S] =. 'h['x:S, 'y:S]
           'g['x:S, 'y:S] =. 'a.S ) .

Maude> (srew < mtRlS, mtRlS, eqs > using COMP .)

result System:
      < 'f['x:S]-> 'a.S
        'g['x:S,'y:S]-> 'a.S
        'h['x:S,'y:S]-> 'a.S,
       mtRlS, mtEqS >
```

S-completion

- An improvement of N-completion where a rule is used for simplification as soon as it has been generated.
- The data structure has now four components:
  - E is a set of identities, like in N-completion,
  - S is a (singleton or empty) set of oriented identities (rules) that are used to simplify other rules,
  - T is a set of rules already used for simplifying, but whose critical pairs have not been computed yet, and
  - R is another set of rules whose critical pairs have already been computed, like in N-completion.

S-completion: Rules

```
rl [Deduce] : < R, T, S, E > => < R, T, S, E s =. t > . *** if  s <-- u --> t

crl [Orient] : < R, T, S, E s =. t > => < R, T, S s -> t, E >  if s > t .

rl [Delete] : < R, T, S, E s =. s > => < R, T, S, E > .

crl [Simplify] : < R, T, S, E s =. t > => < R, T, S, E u =. t >
 if u := reduce(s, R T) .

crl [R-Simplify] : < R s -> t, T, S, E > => < R s -> u, T, S, E >
 if u := reduce(t, S) .

crl [L-Simplify] : < R s -> t, T, S, E > => < R, T, S, E u =. t >
 if u := reduce>(s -> t, S) .

rl [move] : < R, r T, S, E > => < r R, T, S, E > .

rl [concatTS] : < R, T, S, E > => < R, T S, mtRlS, E > .
```

S-completion: Strategies

```
seq COMP =  success orelse simplify-rules orelse deduce orelse orient .

seq success = match ( < R, mtRlS, mtRlS, mtEqS > ) .

seq simplify-rules = match (< R, T, r S, E >) ;
                     (L-Simplify | R-Simplify) ! ;
                     concatTS ; COMP .

seq deduce = match (< R, r T, mtRlS, mtEqS >) ;
             deduction ; COMP .

seq deduction = matchrew < R, r' T, S, E > by
                  < R, r' T, S, E > using (add-crit-pairs(CP(r', R r')) ;
                                           move[r <- r']) .

seq add-crit-pairs(mtEqS) = idle .
seq add-crit-pairs(s1 =. t1 E) = Deduce[s <- s1 ; t <- t1] ; add-crit-pairs(E) .

seq orient = match ( < R, T, mtRlS, e E > ) ;
             simplify-eqs ;
             ( (match ( < R, T, S, mtEqS > ) ; COMP)
             orelse (Orient ; COMP) ).

seq simplify-eqs = (Delete | Simplify) ! .
```

ANS-completion

- S-completion computes all the critical pairs between all rules in R and one rule in T.
- In order to apply simplification as soon as possible, it is better to compute the critical pairs between one rule in R and one rule in T at a time.
- The data structure has now six components:
  - E is a set of identities, like in S-completion,
  - S is a set of simplifying rules, like in S-completion,
  - T is a set of rules coming form S and waiting to enter C,
  - C is a set that contains at most one rule and whose critical pairs are computed with one in N,
  - N is the part of R whose critical pairs have not been computed with C but whose critical pairs with $A \cup N$ have been computed, and
  - A is a set whose critical pairs with $A \cup N \cup C$ have been computed.

ANS-completion: In words

- When E is not empty:
  - First, its identities are simplified by means of all the rules in $A \cup N \cup C \cup T$.
  - Then an identity is oriented and transferred to S.
  - This identity is used to simplify all the rules in A, N, C and T, and then moved to T.
  - The process begins again.
- When E is empty:
  - One rule in T is chosen to go into C.
  - This rule is used to generate all critical pairs with a rule in N, which is then transferred to A.
  - The process begins again with the computed critical pairs that are now in E.

ANS-completion: Rules

```
rl [Deduce] : < A, N, C, T, S, E > => < A, N, C, T, S, E s =. t > . *** if  s <--

crl [Orient] : < A, N, C, T, S, E s =. t > => < A, N, C, T, S s -> t, E >  if s >

rl [Delete] : < A, N, C, T, S, E s =. s > => < A, N, C, T, S, E > .

crl [Simplify] : < A, N, C, T, S, E s =. t > => < A, N, C, T, S, E u =. t >
 if u := reduce(s, A N C T) .

crl [R-Simplify] : < A s -> t, N, C, T, S, E > => < A s -> u, N, C, T, S, E >
 if u := reduce(t, S) .

crl [L-Simplify] : < A s -> t, N, C, T, S, E > => < A, N, C, T, S, E u =. t >
 if u := reduce>(s -> t, S) .

rl [move] : < A, r N, C, T, S, E > => < r A, N, C, T, S, E > .

rl [concatTS] : < A, N, C, T, S, E > => < A, N, C, T S, mtRlS, E > .

rl [AC2N] : < A, N, C, T, S, E > => < mtRlS, A N C, mtRlS, T, S, E > .

crl [fillC] : < A, N, C, T, S, E > => < mtRlS, A N, r, T', S, E >
 if  r := least-rule(T)  /\  r T' := T .
```

ANS-completion: Strategies

```
seq COMP =  success              orelse
            simplify-rules       orelse
            orientation          orelse
            deduce               orelse
            internal-deduction   orelse
            new-loop .

seq success = match ( < A, N, mtRlS, mtRlS, mtRlS, mtEqS > ) .

seq simplify-rules = match (< A, N, C, T, r S, E >) ;
                     (L-Simplify | R-Simplify) ! ;
                     concatTS ;
                     COMP .

seq orientation = match ( < A, N, C, T, mtRlS, e E > ) ;
                  simplify-eqs ;
                  ( (match ( < A, N, C, T, S, mtEqS > ) ; COMP)
                    orelse (Orient ; COMP) ) .

seq simplify-eqs = (Delete | Simplify) ! .
```

ANS-completion: Strategies

```
seq deduce = match (< A, r N, r', T, mtRlS, mtEqS >) ;
             deduction ;
             COMP .

seq deduction = matchrew < A, r' N, r'', T, S, E > s.t. r' := least-rule(r' N) b
    < A, r' N, r'', T, S, E > using (add-crit-pairs(CP(r'', r')) ;
                                        move[r <- r']) .

seq add-crit-pairs(mtEqS) = idle .
seq add-crit-pairs(s1 =. t1 E) = Deduce[s <- s1 ; t <- t1] ; add-crit-pairs(E) .

seq internal-deduction = ( matchrew < A, mtRlS, r', T, mtRlS, mtEqS > by
    < A, mtRlS, r', T, mtRlS, mtEqS > using (add-crit-pairs(CP(r', r')) ;
                                                AC2N) ) ;
                          COMP .

seq new-loop = fillC ; COMP .
```