Parameterized strategy specification
with Maude

Narciso Martí-Oliet, Isabel Pita, Rubén Rubio, Alberto Verdejo

Facultad de Informática
Universidad Complutense de Madrid

Rio de Janeiro, October 2018
Talk plan

Generic and compositional control specification by means of parameterized strategies in Maude

1. Maude and its strategy language
2. Parameterization
3. Examples
   - Generic backtracking
   - Simplex algorithm
   - \(\lambda\)-calculus and a functional language
   - Line breaking algorithm
   - Flat map and fractals
   - Branch and bound
4. Conclusions
Maude is a high-level language and high-performance system.

- It supports both equational and rewriting logic computation.
- It is a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency.
- It is also a good logical framework, i.e., a metalogic in which many other logics can be naturally represented and implemented.
- Moreover, it is reflective allowing many advanced metaprogramming and metalanguage applications.
Maude specifications

• Functional modules `fmod M is ... endfm` define membership equational logic theories.

• Order-sorted signature `Ω = (K, Σ, S)`.

• Equations and membership axioms:

\[(∀ X) \quad t = t' \quad \text{if} \quad \bigwedge_i u_i = v_i \land \bigwedge_j u_j : s_j\]

• Operator axioms, like commutativity, associativity, and identity.
Maude specifications

• Functional modules \texttt{fmod} \texttt{M is ... endfm} define membership equational logic theories.

• System modules \texttt{mod} \texttt{M is ... endm} are rewriting logic theories.

• \( \mathcal{R} = (\Sigma, E \cup A, R) \) adds rewriting rules \( R \) on top of the equational theory.

• Rules do not have to be either confluent or terminating.

\[
(\forall X) \ t \Rightarrow t' \quad \text{if} \quad \bigwedge_{i} u_i = v_i \wedge \bigwedge_{j} u_j : s_j \wedge \bigwedge_{k} u_k \Rightarrow v_k
\]
Maude specifications

- Functional modules `fmod M is ... endfm` define membership equational logic theories.
- System modules `mod M is ... endm` are rewriting logic theories.
- Strategy modules `smod M is ... endsm` allow finer control to rule rewriting using the strategy language.
Strategy language

• Maude provides commands `rewrite` and `frewrite` to obtain a single rule execution path, and `search` to get all of them.
• But the user may be interested in obtaining those paths satisfying a given constraint. Then, strategies are needed.
• Strategy $\alpha$ is seen as an operation transforming a term $t$ into a set of terms, since the process is nondeterministic in general.
• Strategies can be executed with the command `srewrite t using $\alpha$`.
• The most basic strategy is rule application

\[
\text{top}(\text{label}[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]\{\alpha_1, \ldots, \alpha_k\})
\]
Strategy language

• Regular expressions

\[ \alpha ; \beta \; \alpha | \beta \; \alpha^* \; \text{idle} \; \text{fail} \]

\[
\llbracket \alpha ; \beta \rrbracket(\theta, t) = \text{let } t' \leftarrow \llbracket \alpha \rrbracket(\theta, t) : \llbracket \beta \rrbracket(\theta, t')
\]

\[
\llbracket \alpha | \beta \rrbracket(\theta, t) = \llbracket \alpha \rrbracket(\theta, t) \cup \llbracket \beta \rrbracket(\theta, t)
\]

\[
\llbracket \alpha^* \rrbracket(\theta, t) = \bigcup_{n=0}^{\infty} \llbracket \alpha \rrbracket^n(\theta, t)
\]

\[
\llbracket \text{idle} \rrbracket(\theta, t) = \{t\}
\]

\[
\llbracket \text{fail} \rrbracket(\theta, t) = \emptyset
\]

• Conditionals

\[ \alpha ? \beta : \gamma \]

\[
\llbracket \alpha ? \beta : \gamma \rrbracket(\theta, t) = \begin{cases} 
\llbracket \alpha ; \beta \rrbracket(\theta, t) & \text{if } \llbracket \alpha \rrbracket(\theta, t) \neq \emptyset \\
\llbracket \gamma \rrbracket(\theta, t) & \text{otherwise}
\end{cases}
\]
Strategy language

• Tests

\[ \text{amatch } P \text{ s.t. } C \]

• Rewriting of subterms

\[ \text{amatchrew } P \text{ s.t. } C \text{ by } x_1 \text{ using } \alpha_1, \ldots, x_n \text{ using } \alpha_n \]

\[
\begin{align*}
[mrew](\theta, t) &= \bigcup \left\{ \text{let}_{i=1}^{n} t_i \leftarrow [\alpha_i](\sigma \circ \theta, \sigma(x_i)) : P[x_i/t_i]_{i=1}^{n} \mid 
\sigma \in \text{match}(P, t, C, \theta) \right\}
\end{align*}
\]

• Named strategies with parameters and recursion
Strategy modules

\[ \text{smod } M \text{ is ... endsm} \]

• Strategy declarations

\[ \text{strat } \text{sname} : T_1 \ldots T_n @ T \]

• Strategy definitions

\[ \text{sd } \text{sname}(t_1,\ldots,t_n) := \alpha \]

\[ \text{csd } \text{sname}(t_1,\ldots,t_n) := \alpha \text{ if } C \]
Parameterization

- Functional and strategic requirements are declared in a theory

  \[
  \text{fth } T \text{ is } \ldots \text{ endfth} \quad \text{sth } T \text{ is } \ldots \text{ endsth}
  \]

- Parameterized modules receive arguments bound to a theory

  \[
  \text{fmod } \text{LIST}\{X :: \text{TRIV}\} \text{ is } \ldots \text{ endfm}
  \]

- Views map sorts, operations, and strategies in a theory to their instances in a target module.

- Module instantiation is based on the pushout along a view.

\[
\begin{array}{ccc}
\text{TRIV} & \xrightarrow{\text{Nat}} & \text{NAT} \\
\downarrow & & \downarrow \\
\text{LIST}\{X :: \text{TRIV}\} & \longrightarrow & \text{LIST}\{\text{Nat}\}
\end{array}
\]
Backtracking example

Abstract problem definition

```pseudo
fth BT-ELEMS-BASE is
  protecting BOOL .
  sort State .
  op isOk : State → Bool .
  op isSolution : State → Bool .
endfth

sth BT-ELEMS is
  including BT-ELEMS-BASE .
  strat expand @ State .
endst
```
Backtracking example

Abstract problem definition

\texttt{fth BT-ELEMS-BASE is protect}ing BOOL .
\texttt{sort State .}
\texttt{op isOk : State \rightarrow Bool .}
\texttt{op isSolution : State \rightarrow Bool .}
\texttt{endfth}

\texttt{sth BT-ELEMS is including BT-ELEMS-BASE .}
\texttt{strat expand @ State .}
\texttt{endsth}

Parameterized module

\texttt{smod BT-STRAT\{X :: BT-ELEMS\} is}
\texttt{var S : X$State .}
\texttt{strat solve @ X$State .}
\texttt{sd solve := (match S s.t. isSolution(S)) ? idle}
\texttt{expand ; (match S s.t. isOk(S) ; solve) .}
\texttt{endsm}
Backtracking example – labyrinth

\texttt{mod Labyrinth is}
\texttt{including LIST\{Pos\} .}

\texttt{ops isSolution isOk : List\{Pos\} \to \text{Bool} .}
\texttt{op next : List\{Pos\} \to \text{Pos} .}
\texttt{op wall : \to List\{Pos\} .}

\texttt{vars X Y : Nat .}
\texttt{vars P Q : Pos .}
\texttt{var L : List\{Pos\} .}

\texttt{eq wall = [5,5] [5,6] [5,7] [5, 8] [6,5] [7,5] .}

\texttt{eq isSolution(L [8,8]) = true .}
\texttt{eq isSolution(L) = false [otherwise] .}

\texttt{eq isOk(L [X,Y]) = X \geq 1 \land Y \geq 1 \land X \leq 8 \land Y \leq 8}
\texttt{\text{and not(contains(wall, [X,Y])) \land}}
\texttt{\text{not(contains(L, [X,Y])) .}}

\texttt{crl [extend] : L \Rightarrow L \; P \; \text{if} \; \text{next}(L) \Rightarrow P .}
\texttt{rl [next] : next(L [X,Y]) \Rightarrow [X + 1, Y] .}
\texttt{rl [next] : next(L [X,Y]) \Rightarrow [X, Y + 1] .}
\texttt{rl [next] : next(L [X,Y]) \Rightarrow [sd(X, 1), Y] .}
\texttt{rl [next] : next(L [X,Y]) \Rightarrow [X, sd(Y, 1)] .}

\texttt{endm}
Backtracking example – labyrinth

mod LABYRINTH is
  *** [...]  

  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)] .
endm

smod LABYRINTH-STRAT is
  protecting LABYRINTH .

  strat expand @ List{Pos} .  
  sd expand := top(extend{next}) .
endsm
Backtracking example

Abstract problem definition (once again)

\[
\begin{aligned}
\text{fth BT-ELEMS-BASE is } & \quad \text{sth BT-ELEMS is} \\
\quad \text{protecting BOOL .} & \quad \text{including BT-ELEMS-BASE .} \\
\quad \text{sort State .} & \quad \text{strat expand @ State .} \\
\quad \text{op isOk : State } \rightarrow \text{Bool .} & \quad \text{endsth} \\
\quad \text{op isSolution : State } \rightarrow \text{Bool .} & \\
\end{aligned}
\]

Problem instantiation

\[
\begin{aligned}
\text{view LABYRINTH-BT-ELEM from BT-ELEMS to LABYRINTH-STRAT is} \\
\quad \text{sort State to List\{Pos\} .} \\
\quad \text{op isOk to isOk .} \\
\quad \text{op isSolution to isSolution .} \\
\quad \text{strat expand to expand .} \\
\end{aligned}
\]
Is this really backtracking?

\[
\text{sd} \quad \text{solve} := (\text{match } S \text{ s.t. } \text{isSolution}(S)) \ ? \ \text{idle} \\
\quad : (\text{expand} \ ; \\
\quad \text{match } S \text{ s.t. } \text{isOk}(S) \ ; \\
\quad \text{solve}) .
\]
Is this really backtracking?

sd solve := (match S s.t. isSolution(S)) ? idle
: (expand ;
    match S s.t. isOk(S) ;
    solve).

What is understood by backtracking
Is this really backtracking?

```plaintext
sd solve := (match S s.t. isSolution(S)) ? idle :
  (expand ;
   match S s.t. isOk(S) ;
   solve).
```

What is understood by *backtracking*
Is this really backtracking?

\[ sd \text{ solve} := (\text{match } S \text{ s.t. isSolution}(S) \text{ ? idle} \\
\quad : (\text{expand} ; \text{match } S \text{ s.t. isOk}(S) ; \text{solve}) .) \]

What is understood by backtracking
Is this really backtracking?

\[
\text{sd} \quad \text{solve} := \begin{cases} 
\text{match } S \text{ s.t. } \text{isSolution}(S) \text{ ? idle} \\
\text{: (expand ; } \\
\text{match } S \text{ s.t. } \text{isOk}(S) ; \\
\text{solve) .} 
\end{cases}
\]

What is understood by \textit{backtracking}
Is this really backtracking?

\[
\texttt{sd \ solve := (match \ S \ s.t. \ isSolution(S)) \ ? \ idle} \\
\quad : \ (\texttt{expand} \ ; \\
\quad \quad \texttt{match} \ S \ s.t. \ isOk(S) \ ; \\
\quad \quad \texttt{solve}) .
\]

What is understood by \textit{backtracking}

Is this really backtracking?

\[
\text{sd} \quad \text{solve} := \begin{cases} 
\text{match } S \text{ s.t. isSolution}(S) \text{ ? idle} \\
\text{expand} ; \\
\text{match } S \text{ s.t. isOk}(S) ; \\
\text{solve} \end{cases}
\]

What is understood by \textit{backtracking}
Is this really backtracking?

```
sd solve := (match S s.t. isSolution(S)) ? idle
    : (expand ;
        match S s.t. isOk(S) ;
        solve).
```

What is understood by backtracking
Is this really backtracking?

\[
\text{sd solve} := (\text{match } S \text{ s.t. } \text{isSolution}(S)) \ ? \ \text{idle} \\
: (\text{expand} \\
\quad \text{match } S \text{ s.t. } \text{isOk}(S) \\
\quad \text{solve}) .
\]

What srew does as default (a fair search)
Is this really backtracking?

```
sd solve := (match S s.t. isSolution(S)) ? idle 
: (expand ;
   match S s.t. isOk(S) ;
   solve).
```

What srew does as default (a fair search)
Is this really backtracking?

\[
\text{sd} \quad \text{solve} := (\text{match } S \text{ s.t. isSolution}(S)) \ ? \ \text{idle} \\
\quad : (\text{expand} \ ; \\
\qquad \text{match } S \text{ s.t. isOk}(S) \ ; \\
\qquad \quad \text{solve}) .
\]

What screw does as default (a fair search)
Is this really backtracking?

\[
\text{sd} \; \text{solve} := (\text{match } S \text{ s.t. isSolution}(S)) \; ? \; \text{idle} \\
\quad : (\text{expand} ; \text{match } S \text{ s.t. isOk}(S) ; \text{solve}) .
\]

What srew does as default (a fair search)
Is this really backtracking?

\[
\text{sd } \text{solve} := (\text{match } S \text{ s.t. isSolution}(S)) \ ? \ \text{idle} \\
\quad : (\text{expand} ; \\
\quad \quad \text{match } S \text{ s.t. isOk}(S) ; \\
\quad \quad \text{solve}) .
\]

What srew does as default (a fair search)
Is this really backtracking?

\[ \text{sd} \ \text{solve} := (\text{match } S \ \text{s.t. } \text{isSolution}(S)) \ ? \ \text{idle} \]
\[ : (\text{expand} ; \]
\[ \text{match } S \ \text{s.t. } \text{isOk}(S) ; \]
\[ \text{solve}) . \]

What srew does as default (a fair search)
Is this really backtracking?

\[
\text{sd}\ s\text{olve} := (\text{match } S \text{ s.t. } \text{isSolution}(S)) \ ? \ \text{idle} \\
\quad : (\text{expand} \ ; \\
\quad \quad \text{match } S \text{ s.t. } \text{isOk}(S) \ ; \\
\quad \quad \text{solve}) .
\]

What srew does as default (a fair search)
Simplex algorithm

A method (G. Dantzig, ~1947) for solving linear programming problems.

\[
\begin{align*}
\text{max/min} & \quad c_1 x_1 + \cdots + c_n x_n \\
& \quad a_{11} x_1 + \cdots + a_{1n} x_n \geq b_1 \\
& \quad a_{21} x_1 + \cdots + a_{2n} x_n \leq b_2 \\
& \quad \vdots \\
& \quad a_{m1} x_1 + \cdots + a_{mn} x_n = b_m \\
& \quad x_1, \ldots, x_n \geq 0
\end{align*}
\]

The goal is to find \((x_1, \ldots, x_n)\) satisfying all linear constraints and maximizing (or minimizing) the linear functional \(c_1 x_1 + \cdots + c_n x_n\).
Simplex algorithm
Simplex algorithm

Linear algebra and linear programming infrastructure

Simplex tables and equationally defined operations on them

Functional System Strategy

Parameterized strategies
Simplex algorithm

Rules to build the simplex table

Rules for the simplex algorithm execution: pivoting, phase change, unboundness or termination detection...

SIMPLEX-CONSTR

SIMPLEX-EXECUTION

SIMPLEX-LEXICO

SIMPLEX-PIVOTING

SIMPLEX-STRAT\{X :: PIVOTING-STRAT\}
Simplex algorithm

Strategies to guide all the process

Alternative strategies to pivot including standard loop prevention techniques like Bland and lexicographic rules

We will instantiate this module with multiple views

N. Martí, I. Pita, R. Rubio, A. Verdejo (UCM)
Simplex algorithm – parameters

• A non-deterministic rule \textit{pivot} is defined in SIMPLEX-EXECUTION.

\begin{verbatim}
crl [pivot] : Table ⇒ pivot(Table, Ve, Vl) if
  Ve, R := enterVars(Table) ∧
  Vl, S := leaveVars(Table, Ve).
\end{verbatim}

• The theory requires a strategy pivotingStrat @ SimplexTable.
• The parameterized module controls the whole process with it.

\begin{verbatim}
sd solve := makeTable ; simplex .
sd step := (unbounded | finish | phase2 | unfeas)
  or-else pivotingStrat .
\end{verbatim}
Simplex algorithm – parameters

Strategies impose various restrictions using pivot.

*** Bland rule

\[
\text{sd bland} := \text{matchrew } T \text{ s.t.} \\
Ve := \min \text{Var} (\text{enterVars}(T)) \land \\
Vl := \min \text{Var} (\text{leaveVars}(T, Ve)) \\
\text{by } T \text{ using pivot}[Ve \leftarrow Ve, Vl \leftarrow Vl].
\]

*** Lexicographic rule

\[
\text{sd lexico} := \text{matchrew } T \text{ s.t.} \\
Ve := \min \text{Var} (\text{enterVars}^*(T)) \land \\
Vl := \text{lexVar}(T, \text{leaveVars}(T, Ve), Ve) \\
\text{by } T \text{ using pivot}[Ve \leftarrow Ve, Vl \leftarrow Vl].
\]

Views from PIVOTING-STRAT are defined to instantiate SIMPLEX-STRAT.

\[
\text{view Bland from PIVOTING-STRAT to SIMPLEX-PIVOTING is} \\
\text{strat pivotingStrat to bland. *** or lexico, minmax, etc.}
\]

endv
Analysis of the specified system

- Performance and results comparison between the different strategies.

- Analysis of more complex properties by simulation.

For example, in this case, the number of iterations (pivot executions) until a solution is found can be compared among strategies and to the least possible number. All this can be computed with parameterized strategy modules.

<table>
<thead>
<tr>
<th></th>
<th>Free</th>
<th>Bland</th>
<th>Lexicographic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations less optimum</td>
<td>2.05</td>
<td>2.05</td>
<td>1.47</td>
</tr>
<tr>
<td>Number of rewrites</td>
<td>4246</td>
<td>5195</td>
<td>5191</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>1.84</td>
<td>2.29</td>
<td>2.2</td>
</tr>
</tbody>
</table>
Analysis of the specified system

• Model checking.

For example, given a fixed LP problem and a fixed strategy, model checking the LTL formula $\Diamond$ \textit{solution}, we can ensure that the algorithm never cycles or obtain the trace of a cyclic execution.

```maude
Maude> red modelCheck(cycles, <> solution, 'free) .
reduce in SIMPLEX-MC : modelCheck(cycles, <> solution, 'free) .
rewrites: 6051 in 96ms cpu (94ms real) (63031 rewrites/second)
result ModelCheckResult: counterexample(nil,
   {TSimplex max @ x(5) x(6) x(7) | ...,'pivot}
   {TSimplex max @ x(1) x(6) x(7) | ...,'pivot}
   {TSimplex max @ x(1) x(2) x(7) | ...,'pivot}
   {TSimplex max @ x(3) x(2) x(7) | ...,'pivot}
   {TSimplex max @ x(3) x(4) x(7) | ...,'pivot}
   {TSimplex max @ x(5) x(4) x(7) | ...,'pivot}
)
```

As expected, free pivoting is the only strategy that may cycle in some examples.
\textbf{\textit{\lambda}-calculus}

\textbf{mod} LAMBDA \textbf{is}
\begin{itemize}
\item \textbf{sorts} Var LambdaTerm .
\item \textbf{subsort} Var < LambdaTerm .
\item \textbf{op} \_\_\_ : Var LambdaTerm \rightarrow LambdaTerm \textbf{[ctor]} .
\item \textbf{op} __ : LambdaTerm LambdaTerm \rightarrow LambdaTerm \textbf{[ctor]} .
\item \textbf{rl} [beta] : (\_\_\_. M) N \Rightarrow \text{subst}(M, x, N) .
\end{itemize}
\textbf{endm}

- Reduction can be done with the \texttt{rew} command, but which $\beta$-redex is reduced first matters.

\[ K = (\lambda x. (\lambda y. x)) \quad I = \lambda x. x \quad \Omega = (\lambda x. xx)(\lambda x. xx) \]

\[
\begin{array}{c}
(KI)\Omega \quad \overset{\text{\texttt{rew} (KI)}}{\longrightarrow} \quad (\Omega y. I)\Omega \quad \overset{\text{\texttt{rew} (\Omega)}}{\longrightarrow} \quad I \\
\end{array}
\]
A strategy parameter `reduce` is supposed to make a single $\beta$-reduction in a $\lambda$-term. A parameterized module uses it to calculate normal forms.

```
sth LAMBDA-STRATEGY is
    including LAMBDA .
    strat reduce @ LambdaTerm .
endsth

smod LAMBDA-REDUCE{X :: LAMBDA-STRATEGY} is
    strat fullReduce @ LambdaTerm .
    sd fullReduce := reduce ? fullReduce : idle .
endsm

view Applicative from LAMBDA-STRATEGY to LAMBDA-STRATS is
    strat reduce to applicative .
endv

smod LAMBDA-MAIN is
    protecting LAMBDA-REDUCE{Applicative} .
endsm
\(\lambda\)-calculus – strategies

- **Applicative order (inner rightmost redex first)**
  
  \[
  \text{sd\ applicative} := (\text{matchrew} \ x . M \ \text{by} \ M \ \text{using} \ \text{applicative})
  \ | (\text{matchrew} \ M \ N \ \text{by} \ N \ \text{using} \ \text{applicative})
  \ | \text{or-else } \text{matchrew} \ M \ N \ \text{by} \ M \ \text{using} \ \text{applicative}
  \ | \text{or-else } \text{top} (\beta).
  \]

- **Normalizing strategy (outer leftmost redex first)**
  
  \[
  \text{sd\ normal} := (\text{matchrew} \ x . M \ \text{by} \ M \ \text{using} \ \text{normal})
  \ | \text{top} (\beta)
  \ | \text{or-else } \text{matchrew} \ M \ N \ \text{by} \ M \ \text{using} \ \text{normal}
  \ | \text{or-else } \text{matchrew} \ M \ N \ \text{by} \ N \ \text{using} \ \text{normal}.
  \]

- **By name (normalizing but no reduction inside abstraction)**
  
  \[
  \text{sd\ byname} := \text{top} (\beta)
  \ | \text{or-else } \text{matchrew} \ M \ N \ \text{by} \ M \ \text{using} \ \text{byname}
  \ | \text{or-else } \text{matchrew} \ M \ N \ \text{by} \ N \ \text{using} \ \text{byname}.
  \]

- **By value (only outermost redex and when argument is value)**
  
  \[
  \text{sd\ byvalue} := (\text{match} (\ \ x . M) \ z
  \ | \text{match} (\ \ x . M) (\ \ y . N)) \ ; \ \text{top} (\beta).
  \]
\( \lambda \)-calculus – examples

\[
K = (\lambda x.(\lambda y.x)) \quad I = \lambda x.x \quad \Omega = (\lambda x.x x)(\lambda x.x x)
\]

<table>
<thead>
<tr>
<th></th>
<th>Applicative</th>
<th>Normalizing</th>
<th>By name</th>
<th>By value</th>
</tr>
</thead>
<tbody>
<tr>
<td>((KI)\Omega)</td>
<td>Does not terminate</td>
<td>(I)</td>
<td>(I)</td>
<td>((KI)\Omega)</td>
</tr>
<tr>
<td>(\lambda y.(Iz))</td>
<td>(\lambda y.z)</td>
<td>(\lambda y.z)</td>
<td>(\lambda y.(Iz))</td>
<td>(\lambda y.(Iz))</td>
</tr>
<tr>
<td>((Kz)t)</td>
<td>(z)</td>
<td>(z)</td>
<td>(z)</td>
<td>((Kz)t)</td>
</tr>
<tr>
<td>(Kz)</td>
<td>(\lambda y.z)</td>
<td>(\lambda y.z)</td>
<td>(\lambda y.z)</td>
<td>(\lambda y.z)</td>
</tr>
</tbody>
</table>

Parameterized strategies

Rio de Janeiro, October 2018
REC language
from Chapter 9 of Winskel’s *The Formal Semantics of Programming Languages*

```
RECELanguage
from Chapter 9 of Winskel’s *The Formal Semantics of Programming Languages*

REC-EXPR
  \rightarrow
REC-DEFS
  \rightarrow
REC-RULES
  \rightarrow
REC-STRATEGY
  \rightarrow
REC-STRATS
  \downarrow
REC-MAIN\{X :: REC-STRATEGY\}
```

---

**Functional**

**System**

**Strategy**

REC language
from Chapter 9 of Winskel’s *The Formal Semantics of Programming Languages*

Expressions with integer constants, $+$, $-$, $\ast$, and function calls

Syntax for function definition: $f(x_1, \ldots, x_n) := \langle \text{expr} \rangle$ and auxiliary operations like substitutions

Functional
System
Strategy

Parameterized strategies
Rio de Janeiro, October 2018
Rules for executing conditionals and function calls:

\[
\text{rl} \left[ \text{apply} \right] : \ Q(\text{Args}) \Rightarrow \text{apply}(\text{find}(Q, \text{Defs}), \text{Args}) .
\]

\[
\text{crl} \left[ \text{cond} \right] : \text{if } C \text{ then } E \text{ else } F \Rightarrow \text{if } C = 0 \text{ then } E \text{ else } F \text{ fi if } C : \text{Int} .
\]
The parameter strategy is used to reduce expressions until a normal form.
REC language

sth REC-STRATEGY is
    including REC-RULES .

strat st : List{FunctionDef} @ RecExpr .
endsth

Function definitions are provided as a parameter FL for the strategy. This allows extending the language with local definitions easily.

smod REC-MAIN{X :: REC-STRATEGY} is
    strat reduce : List{FunctionDef} @ RecExpr .

    var FL : List{FunctionDef} .
    vars E F G : RecExpr .

    sd reduce(FL) := (cond ! ; st(FL)) ! .
endsm

where $\alpha! \equiv \alpha^* ; (\alpha ? \text{fail} : \text{idle})$. 
REC language

smod REC-STRATS is
   protecting REC-RULES .

var FL : List{FunctionDef} .
var Args : NeArguments .

*** Unrestricted reduction
sd free(FL) := apply[Defs <- FL] .

*** Call by name
sd byname(FL) := top(apply[Defs <- FL]) .

*** Call by value
sd byvalue(FL) := (matchrew Q(Args) by Args using byvalue(FL))
   | or-else top(apply[Defs <- FL])
   | (matchrew E, Args by E using byvalue(FL))
   | or-else matchrew E, Args by Args using byvalue(FL)
   .
endsm

byname and byvalue definitions only deal with function calls.
The main purpose of the extension is preventing the reductions to be applied in the branches of the conditional.

```
smod STRAT-EXTENSION{X :: REC-STRATEGY} is
  *** eXtended sTrategy
  strat xst : List{FunctionDef} @ RecExpr .

  vars E F G : RecExpr .
  var   FL : List{FunctionDef} .

  sd xst(FL) := st(FL)
  | (matchrew E + F by E using xst(FL))
    or-else matchrew E + F by F using xst(FL)
  | (matchrew E * F by E using xst(FL))
    or-else matchrew E * F by F using xst(FL)
  | (matchrew E - F by E using xst(FL))
    or-else matchrew E - F by F using xst(FL)
  | matchrew if E then F else G by E using xst(FL)
  .

endsm
```
REC language – strategy code reuse

Parameterized strategies
view ByName0 from REC-STRATEGY to REC-STRATS is
  strat st to byname .
endv

view ByValue0 from REC-STRATEGY to REC-STRATS is
  strat st to byvalue .
endv

view ByName from REC-STRATEGY to STRAT-EXTENSION{ByName0} is
  strat st to xst .
endv

view ByValue from REC-STRATEGY to STRAT-EXTENSION{ByValue0} is
  strat st to xst .
endv

view Free from REC-STRATEGY to REC-STRATS is
  strat st to free .
endv
REC language – example

\[ f(x) = \text{if } x \text{ then } 1 \text{ else } (x \times f(x - 1)) \]
\[ g(x, y) = g(y, x) \]
\[ h(x) = 3 \]

- With both byName and byValue, \( f(n) \) computes the factorial of \( n \) for all \( n \geq 0 \).
- But with free rewriting, \( f(n) \) may not terminate.
- With byName, \( h(g(1, 2)) \) will evaluate to 3, while byValue will lead to an infinite execution.
Line breaking algorithm

- Receives a list of words and a line width as input, and provides a list of lines (lists of words) as output.
- Strategies fix the criteria for breaking lines. Multiple results can be obtained from non-deterministic strategies.
- Solutions can be discarded on the fly by its raggedness.
Line breaking algorithm

WORD -> LIST{Word} -> STATE-DATA

LINE -> LIST{Line} -> SET{Property}

WRAPPING

WRAPPING-AUX

Functional

WRAPPING-RULES

System

CUSTOM-WRAP

Strategy

WW-RANGE  WW-INTEGER  WW-GREEDY

WORD-WRAP-ALGOR{X :: CUSTOM-WRAP}
The algorithm state contains both the remaining input and the output lines, and some convenient data.

WL:List{Word} |> LL:List{Line} { WL' }
(count : NC, width : W, words : NW, …)
Two rules are defined to feed lines with words and break them: next and newline
Line breaking algorithm

Custom wrappers decide where to break lines

The algorithm generates the list of lines using the rules and the custom wrapper
Line breaking algorithm – behavior

**break** @ State is the parameter defined in CUSTOM-WRAP. The global control applies **next** and then **break** until the input words are exhausted.

\[
\begin{align*}
\text{next}(\theta, t_0) &= \{t_1\} \\
\text{newline}(\theta, t_1) &= \{t_2\}
\end{align*}
\]

Typically, **break** will do the following:

\[
\text{break}(\theta, t_1) = \begin{cases} 
\{t_1\} & \text{idle} \\
\{t_1, t_2\} & \text{idle|newline} \\
\{t_2\} & \text{newline} \\
\emptyset & \text{fail}
\end{cases}
\]
Line breaking algorithm

- Various strategies are defined in separate strategy modules: a greedy strategy, the uniform space between words is within a range...

\[
\text{sd range} (\text{Min}, \text{Max}) := \text{match } S \text{ s.t. numberWords} (S) = 1 \\
\quad \text{or spaceWidth} (S) \geq \text{Min} ; \\
\quad ( \text{match } S \text{ s.t. spaceWidth} (S) \leq \text{Max} ; \text{newline} \\
\quad \mid \text{idle} \\
\quad ) .
\]

- The parameterized module defines the line breaking algorithm with pruning:

\[
\text{sd } \text{wrap} (\text{RG: Nat}) := ( \text{match } \text{nil } \text{ |> } \text{LL} (\text{SD}) ? \text{idle} \\
\quad : \text{next} ; \quad *** \text{ Adds a new word to the line break; *** Should we break here? (the parameter strategy) match } \text{WL } \text{ |> } \text{LL} (\text{raggedness} : \text{N, SD}) \text{ s.t. } \text{N} \leq \text{RG} ; \quad *** \text{ Prune } \\
\quad \text{wrap} (\text{RG}) \\
\quad ) .
\]
Line breaking algorithm – hyphenation

- We can add a hyphenation strategy on top of any of the previous.
- A theory HYPHENATOR requires a strategy to split words, so that they fit better in a line.
- A parameterized module combines the hyphenation and a breaking strategy to define another breaking strategy.

```plaintext
smod WWRAP-HYPHEN {X :: CUSTOM-WRAP * (strat break to baseBreak),
    Y :: HYPHENATOR} is
  protecting WRAPPING-RULES .
  protecting WORDWRAP-HYPHEN-RULES .

  strat break @ State .
  var WL : List{Word} . var LL : List{Line} . var SD : StateData .

  sd break := test(baseBreak ; match WL |> LL { nil } (SD)) ?
      *** If the breaking strategy breaks the line...
        ((hyphen-state{hyphenate} | idle) ; baseBreak)
      : test(baseBreak) .
endsm
```
Flat map

Apply a strategy \( st \) to each element of a list. The \( st \) result may be a list but they are all flattened as in Haskell’s concatMap and Scala’s flatMap.

```
fth MAP-LIST-BASE is
  including TRIV . *** sort Elt .
  *** A list of elements of the type
  sort List .
  subsort Elt < List .
  op nil : → List [ctor] .
  op __ : List List → List [ctor assoc] .
endfth

sth MAP-LIST is
  including MAP-LIST-BASE .

strat st @ List .
endsth
```
**Flat map – implementation**

`flatMap` can be implemented with extra rules, for example.

```plaintext
mod STRAT-LIST\{X :: MAP-LIST-BASE\} is
  vars E E' : X$Elt .
  vars L L' : X$List .

endm

view MapList0 from MAP-LIST-BASE to MAP-LIST is
  *** identity
endv

smod STRAT-MAP\{X :: MAP-LIST\} is
  protecting STRAT-LIST\{MapList0\}\{X\} .

  var L : X$List .
  var E : X$Elt .

  strat map : @ X$List .
  sd map := try(top(nonempty{st, map})) .
endsm
```
Flat map – fractals

• They are represented as list of positions, and a rule that rewrites a position to the positions of their self-similar copies.

\[
\text{crl } [\text{von-koch}] : A \gg B \Rightarrow A \gg C \quad C \gg E \quad E \gg D \quad D \gg B \\
\text{if } C := \text{fractionPoint}(A, B, 1.0 / 3.0) \\
\wedge D := \text{fractionPoint}(A, B, 2.0 / 3.0) \\
\wedge E := \text{equilateralThird}(C, D)
\]

• Free rule application is not convenient.

• We can use flatMap.
Branch and bound

- The BB-PROBLEM theory imposes the following requirements:
  - **Types**: a PartialResult type, a strict totally ordered type Value, and a FixData type to hold static problem information.
  - **Operators**: getBound to get the cost estimation for a partial solution, `result?` to know if the solution is complete, and `numChildren` to find out how many successors a partial solution has.
  - **Strategy**: a expand strategy on PartialResults which receives the fix data, the bound, and the child index in order to generate a successor.

- A state holds a priority queue and the better solution up to now. Rules and strategies extract the most promising partial solution, process it, and add its successors to the list, until the queue becomes empty.
Branch and bound – problem specification

fth BB-PROBLEM-BASE is
  protecting BOOL .
  protecting NAT .
  including STRICT-TOTAL-ORDER * (sort Elt to Value) .

  sort PartialResult .   *** Partial results
  sort FixData .        *** Fixed data

  *** Expected cost estimation
  op getBound : PartialResult FixData → Value .
  *** Get value or cost for a complete result
  op getValue : PartialResult FixData → Value .
  *** Is it a solution?
  op result? : PartialResult FixData → Bool .
  *** An infinity (or initial bound for the problem)
  op infinity : FixData → Value .
  *** Number of successors
  op numChildren : PartialResult FixData → Nat .
endfth

sth BB-PROBLEM is
  including BB-PROBLEM-BASE .

  *** Generates the successors of a partial result.
  ***
  *** This strategy will be called from 0 until numChildren of the
  *** current partial result. Expand should be deterministic but it
  *** is allowed to fail.
  strat expand : Nat FixData Value @ PartialResult .
endsth
Branch and bound – algorithm execution

\texttt{smod BB-STRAT\{X :: BB-PROBLEM\} is}
\texttt{protecting BB-BASE\{Problem\}\{X\} .}

\texttt{var S : BBState . var F : X$\text{FixData} . var V : X$Value .}
\texttt{var P : X$\text{PartialResult} . vars N M : Nat .}

\texttt{strat solve iteration @ BBState .}
\texttt{strat iterChildn : Nat X$\text{PartialResult} X$\text{FixData} X$\text{Value} @ BBState .}

\texttt{sd solve := initial ; (solution \texttt{or-else} iteration) * ; finish .}

\texttt{sd iteration := matchrew S s.t. M := numChildren(top(S), fixData(S))}
\texttt{\quad \land M > 0 by S using (pop ;}
\texttt{\quad \quad iterChildn(sd(M, 1), top(S), fixData(S), upperBound(S))) .}

\texttt{sd iterChildn(0, P, F, V) := try(border[P <- P]\{expand(0, F, V)\}) .}
\texttt{sd iterChildn(s(N), P, F, V) :=}
\texttt{\quad try(border[P <- P]\{expand(s(N), F, V)\}) ;}
\texttt{\quad iterChildren(N, P, F, V) .}

\texttt{endsm}
Partial Results are paths (list of cities), and Values are distances (natural numbers).

The FixData includes the graph and the precalculated cheapest edge cost, from which the getBound function is calculated.

If the number of cities is $n$, complete solutions are paths of length $n + 1$, and partial solutions have $n$ successors (some of them fail).

expand for the $i$-th child generates the path that visits the $i$-th city next, if it is admissible.
Conclusions

• The advantages of functional and system modules parameterization in Maude are now available for the new strategy modules.

• Generic strategy components can be written to be reused in several system specifications.

• Rewriting systems control can be specified compositionally, allowing the execution and analysis of alternative semantics or behaviors easily.

• Parameterization can be applied to the specification of programming languages, algorithmic schemes, ....
Future work

• Apply parameterization to more examples guided by strategies: to old examples like the Eden programming language, to other fields like communication protocols, ....

• Elaborate parameterizations with richer theories combining strategy and functional parameters.

• Go further in the comparison of performance and system properties using alternative strategies.

http://maude.sip.ucm.es/strategies/
Thank you