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
   - λ-calculus and a functional language
   - Line breaking algorithm
   - Flat map and fractals
   - Branch and bound
4. Conclusions

Maude

```
 \||||||||||||||||||/
--- Welcome to Maude ---
 /||||||||||||||||||\
```

http://maude.cs.uiuc.edu

- 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 $\Omega = (K, \Sigma, S)$.
  - Equations and membership axioms:

$$(\forall X) \quad \begin{matrix} t = t' \\ t : s \end{matrix} \quad \textbf{if} \quad \bigwedge_i u_i = v_i \wedge \bigwedge_j u_j : s_j$$

  - Operator axioms, like commutativity, associativity, and identity.

Maude specifications

- Functional modules `fmod M is ... endfm` define membership equational logic theories.
- System modules `mod 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.

$$(\forall X) \quad t \Rightarrow t' \quad \textbf{if} \quad \bigwedge_i u_i = v_i \wedge \bigwedge_j u_j : s_j \wedge \bigwedge_k u_k \Rightarrow v_k$$

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 α 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** α.
- The most basic strategy is rule application

$$\mathtt{top(label[}x_1 \leftarrow t_1\mathtt{,}\ \ldots\mathtt{,}\ x_n \leftarrow t_n\mathtt{]\{}\alpha_1, \ldots, \alpha_k\mathtt{\})}$$

Strategy language

- Regular expressions

$$\alpha \,;\, \beta \quad \alpha \,|\, \beta \quad \alpha* \quad \texttt{idle} \quad \texttt{fail}$$

$$\llbracket \alpha \,;\, \beta \rrbracket(\theta, t) = \text{ let } t' \leftarrowtail \llbracket \alpha \rrbracket(\theta, t) : \llbracket \beta \rrbracket(\theta, t')$$

$$\llbracket \alpha \,|\, \beta \rrbracket(\theta, t) = \llbracket \alpha \rrbracket(\theta, t) \cup \llbracket \beta \rrbracket(\theta, t) \qquad \llbracket \texttt{idle} \rrbracket(\theta, t) = \{t\}$$

$$\llbracket \alpha^* \rrbracket(\theta, t) = \bigcup_{n=0}^{\infty} \llbracket \alpha \rrbracket^n(\theta, t) \qquad \llbracket \texttt{fail} \rrbracket(\theta, t) = \emptyset$$

- Conditionals

$$\alpha \;\texttt{?}\; \beta \;\texttt{:}\; \gamma$$

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

Strategy language

- Tests

$$\texttt{amatch}\ P\ \texttt{s.t.}\ C$$

- Rewriting of subterms

$$\texttt{amatchrew}\ P\ \texttt{s.t.}\ C\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1\ , \ldots, \ x_n\ \texttt{using}\ \alpha_n$$

$$[\![\texttt{mrew}]\!](\theta, t) = \bigcup_{\sigma \in \mathrm{match}(P, t, C, \theta)} \mathrm{let}_{i=1}^{n} t_i \leftarrow [\![\alpha_i]\!](\sigma \circ \theta, \sigma(x_i)) : P[x_i/t_i]_{i=1}^{n}$$

- Named strategies with parameters and recursion

Strategy modules

```
smod M is ... endsm
```

- Strategy declarations

```
strat sname : T1 ... Tn @ T
```

- Strategy definitions

$$\textbf{sd}\ \mathtt{sname}(t_1, \ldots, t_n)\ \coloneqq\ \alpha$$

$$\textbf{csd}\ \mathtt{sname}(t_1, \ldots, t_n)\ \coloneqq\ \alpha\ \textbf{if}\ C$$

Parameterization

- Functional and strategic requirements are declared in a theory

```
fth T is ... endfth        sth T is ... endsth
```

- Parameterized modules receive arguments bound to a theory

```
fmod LIST{X :: TRIV} is ... 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.

$$\begin{tikzcd} \mathtt{TRIV} \arrow[r, "\mathtt{Nat}"] \arrow[d] & \mathtt{NAT} \arrow[d] \\ \mathtt{LIST\{X :: TRIV\}} \arrow[r] & \mathtt{LIST\{Nat\}} \end{tikzcd}$$

Backtracking example

Abstract problem definition

```
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 .
endsth
```

Backtracking example

Abstract problem definition

```
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 .
endsth
```

Parameterized module

```
smod BT-STRAT{X :: BT-ELEMS} is
  var S : X$State .

  strat solve @ X$State .
  sd solve := (match S s.t. isSolution(S)) ? idle
                : (expand ;
                   match S s.t. isOk(S) ;
                   solve) .
endsm
```

Backtracking example – labyrinth

```
mod LABYRINTH is
  including LIST{Pos} .

  ops isSolution isOk : List{Pos} → Bool .
  op next : List{Pos} → Pos . op wall : → List{Pos} .

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

  eq wall = [5,5] [5,6] [5,7] [5, 8] [6,5] [7,5] .

  eq isSolution(L [8,8]) = true .
  eq isSolution(L) = false [owise] .

  eq isOk(L [X,Y]) = X ≥ 1 and Y ≥ 1 and X ≤ 8 and Y ≤ 8
      and not(contains(wall, [X,Y])) and
      not(contains(L, [X,Y])) .

  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
```

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)

```
fth BT-ELEMS-BASE is
  protecting BOOL .
  sort State .

  op isOk : State → Bool .
  op isSolution : State → Bool .
endfth
```

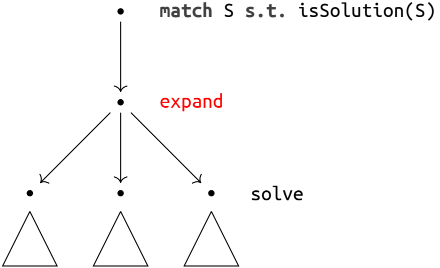
```
sth BT-ELEMS is
  including BT-ELEMS-BASE .

  strat expand @ State .
endsth
```

Problem instantiation

```
view LABYRINTH-BT-ELEM from BT-ELEMS to LABYRINTH-STRAT is
  sort State to List{Pos} .
  op isOk to isOk .
  op isSolution to isSolution .
  strat expand to expand .
endv
```

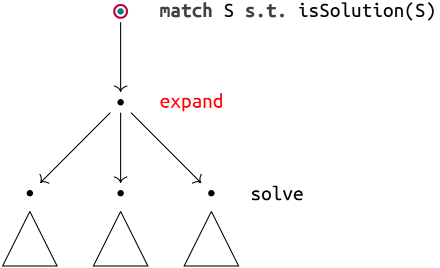
Is this really backtracking?

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



Is this really backtracking?

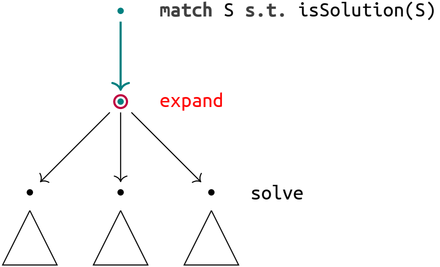
```
sd solve := (match S s.t. isSolution(S)) ? idle
            : (expand ;
               match S s.t. isOk(S) ;
               solve) .
```



What is understood by *backtracking*

Is this really backtracking?

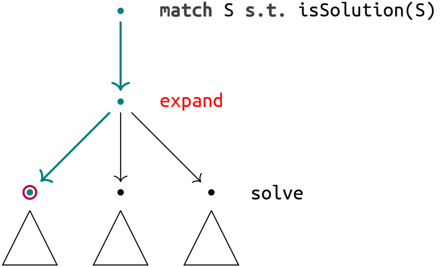
```
sd solve := (match S s.t. isSolution(S)) ? idle
              : (expand ;
                 match S s.t. isOk(S) ;
                 solve) .
```



What is understood by *backtracking*

Is this really backtracking?

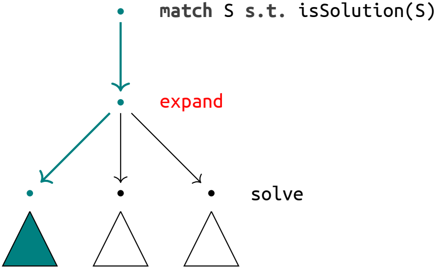
```
sd solve := (match S s.t. isSolution(S)) ? idle
            : (expand ;
               match S s.t. isOk(S) ;
               solve) .
```



What is understood by *backtracking*

Is this really backtracking?

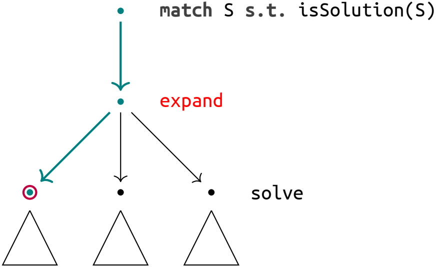
```
sd solve := (match S s.t. isSolution(S)) ? idle
          : (expand ;
             match S s.t. isOk(S) ;
             solve) .
```



What is understood by *backtracking*

Is this really backtracking?

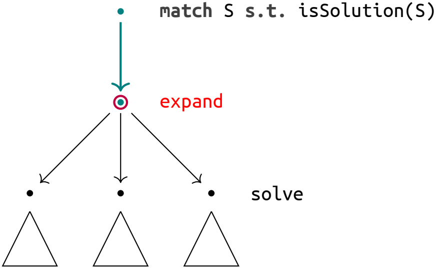
```
sd solve := (match S s.t. isSolution(S)) ? idle
              : (expand ;
                 match S s.t. isOk(S) ;
                 solve) .
```



What is understood by *backtracking*

Is this really backtracking?

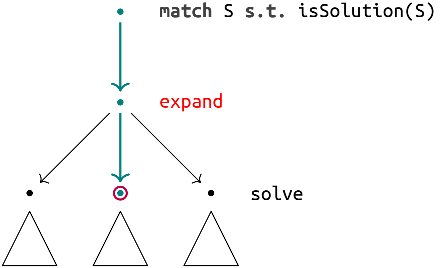
```
sd solve := (match S s.t. isSolution(S)) ? idle
                : (expand ;
                   match S s.t. isOk(S) ;
                   solve) .
```



What is understood by *backtracking*

Is this really backtracking?

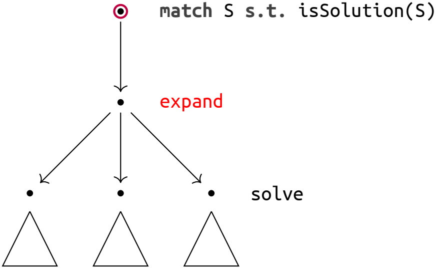
```
sd solve := (match S s.t. isSolution(S)) ? idle
              : (expand ;
                 match S s.t. isOk(S) ;
                 solve) .
```



What is understood by *backtracking*

Is this really backtracking?

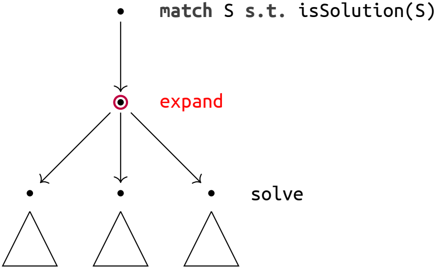
```
sd solve := (match S s.t. isSolution(S)) ? idle
              : (expand ;
                 match S s.t. isOk(S) ;
                 solve) .
```



What srew does as default (a fair search)

Is this really backtracking?

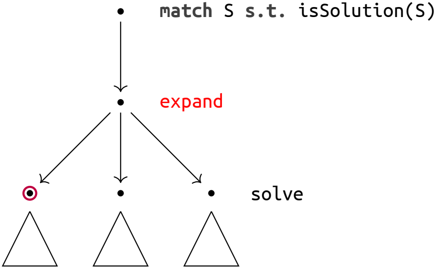
```
sd solve := (match S s.t. isSolution(S)) ? idle
             : (expand ;
                match S s.t. isOk(S) ;
                solve) .
```



What srew does as default (a fair search)

Is this really backtracking?

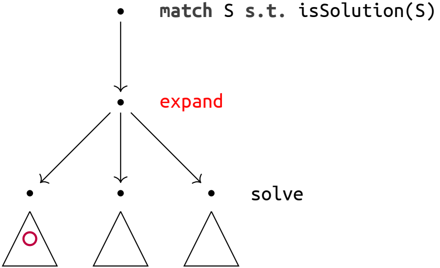
```
sd solve := (match S s.t. isSolution(S)) ? idle
            : (expand ;
               match S s.t. isOk(S) ;
               solve) .
```



What srew does as default (a fair search)

Is this really backtracking?

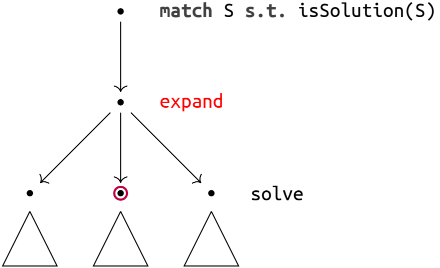
```
sd solve := (match S s.t. isSolution(S)) ? idle
              : (expand ;
                 match S s.t. isOk(S) ;
                 solve) .
```



What srew does as default (a fair search)

Is this really backtracking?

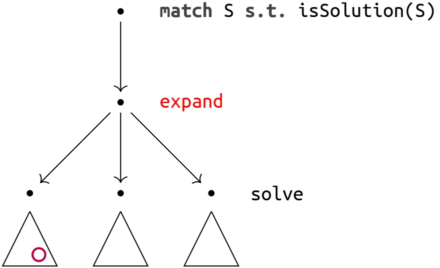
```
sd solve := (match S s.t. isSolution(S)) ? idle
              : (expand ;
                 match S s.t. isOk(S) ;
                 solve) .
```



What srew does as default (a fair search)

Is this really backtracking?

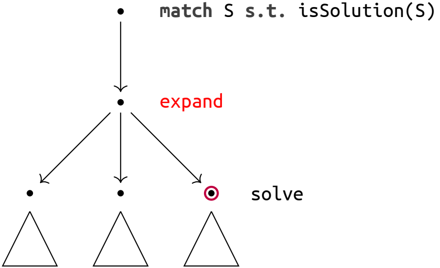
```
sd solve := (match S s.t. isSolution(S)) ? idle
             : (expand ;
                match S s.t. isOk(S) ;
                solve) .
```



What srew does as default (a fair search)

Is this really backtracking?

```
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)

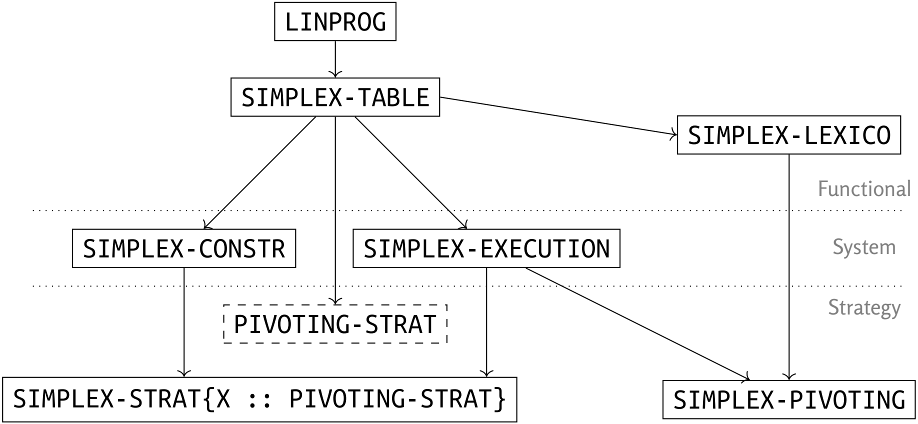
Simplex algorithm

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

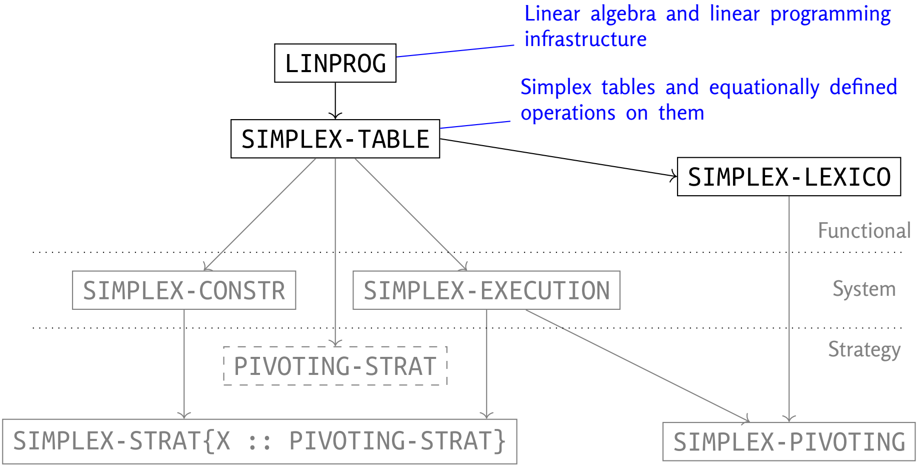
$$
\begin{array}{lccccccccl}
\text{max/min} & c_1 & x_1 & + & \cdots & + & c_n & x_n & & \\
 & a_{11} & x_1 & + & \cdots & + & a_{1n} & x_n & \geq & b_1 \\
 & a_{21} & x_1 & + & \cdots & + & a_{2n} & x_n & \leq & b_2 \\
 & & & & \vdots & & & & & \\
 & a_{m1} & x_1 & + & \cdots & + & a_{mn} & x_n & = & b_m \\
 & \multicolumn{9}{l}{x_1, \ldots, x_n \geq 0}
\end{array}
$$

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

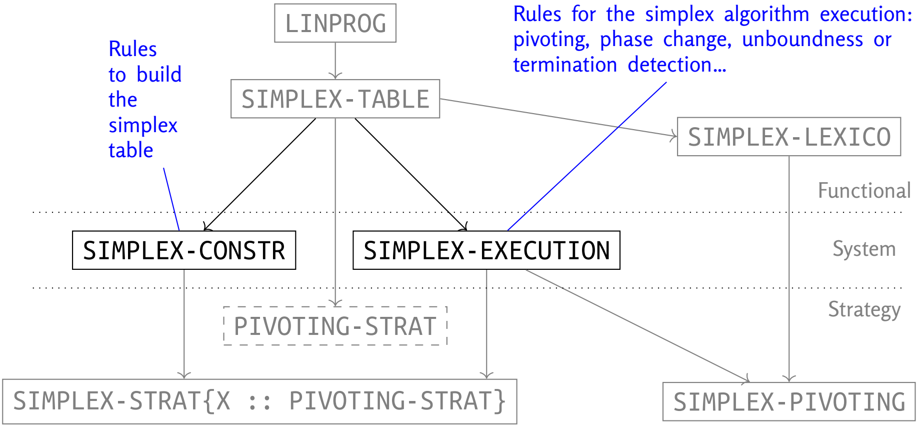
Simplex algorithm



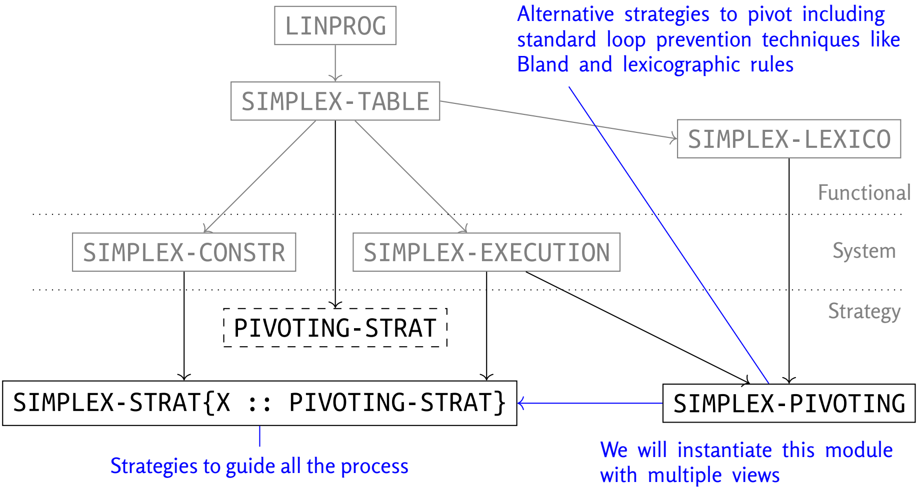
Simplex algorithm



Simplex algorithm



Simplex algorithm



Simplex algorithm – parameters

- A non-deterministic rule pivot is defined in `SIMPLEX-EXECUTION`.

```
crl [pivot] : Table ⇒ pivot(Table, Ve, Vl) if
               Ve, R := enterVars(Table) ∧
               Vl, S := leaveVars(Table, Ve) .
```

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

```
sd solve := makeTable ; simplex .
sd simplex := step ? simplex : idle .
sd step := (unbounded | finish | phase2 | unfeas)
            or-else pivotingStrat .
```

Simplex algorithm – parameters

Strategies impose various restrictions using `pivot`.

```
*** Bland rule
sd bland := matchrew T s.t.
                   Ve := minVar(enterVars(T)) ∧
                   Vl := minVar(leaveVars(T, Ve))
             by T using pivot[Ve ← Ve, Vl ← Vl] .

*** Lexicographic rule
sd lexico := matchrew T s.t.
                   Ve := minVar(enterVars*(T)) ∧
                   Vl := lexVar(T, leaveVars(T, Ve), Ve)
             by T using pivot[Ve ← Ve, Vl ← Vl] .
```

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

```
view Bland from PIVOTING-STRAT to SIMPLEX-PIVOTING is
  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.

(in average)	Free	Bland	Lexicographic
Iterations less optimum	2.05	2.05	1.47
Number of rewrites	4246	5195	5191
Time (ms)	1.84	2.29	2.2

Analysis of the specified system

- Model checking.

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

```
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.

λ-calculus

```
mod LAMBDA is
  sorts Var LambdaTerm .
  subsort Var < LambdaTerm .

  op \_._ : Var LambdaTerm → LambdaTerm [ctor] .
  op __ : LambdaTerm LambdaTerm → LambdaTerm [ctor] .

  rl [beta] : (\ x . M) N ⇒ subst(M, x, N) .
endm
```

- Reduction can be done with the **rew** command, but which β-redex is reduced first matters.

$$\mathbf{K} = (\lambda x.(\lambda y.x)) \qquad \mathbf{I} = \lambda x.x \qquad \Omega = (\lambda x.xx)(\lambda x.xx)$$

$$\begin{array}{lll} & \nearrow (\mathbf{K}\,\mathbf{I})\Omega \circlearrowleft & \\ (\mathbf{K}\,\mathbf{I})\Omega & & \\ & \searrow (\lambda y.\mathbf{I})\Omega & \longrightarrow \mathbf{I} \end{array}$$

λ-calculus – parameterization

A strategy parameter reduce is supposed to make a single β-reduction in a λ-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
```

λ-calculus – strategies

- Applicative order (inner rightmost redex first)

```
sd applicative := (matchrew \ x . M by M using applicative)
  | (matchrew M N by N using applicative)
      or-else matchrew M N by M using applicative
      or-else top(beta) .
```

- Normalizing strategy (outer leftmost redex first)

```
sd normal := (matchrew \ x . M by M using normal)
  | top(beta)
      or-else matchrew M N by M using normal
      or-else matchrew M N by N using normal .
```

- By name (normalizing but no reduction inside abstraction)

```
sd byname := top(beta)
  or-else matchrew M N by M using byname
  or-else matchrew M N by N using byname .
```

- By value (only outermost redex and when argument is value)

```
sd byvalue := (match (\ x . M) z
  | match (\ x . M) (\ y . N)) ; top(beta) .
```

λ-calculus – examples

$$\mathbf{K} = (\lambda x.(\lambda y.x)) \qquad \mathbf{I} = \lambda x.x \qquad \Omega = (\lambda x.xx)(\lambda x.xx)$$

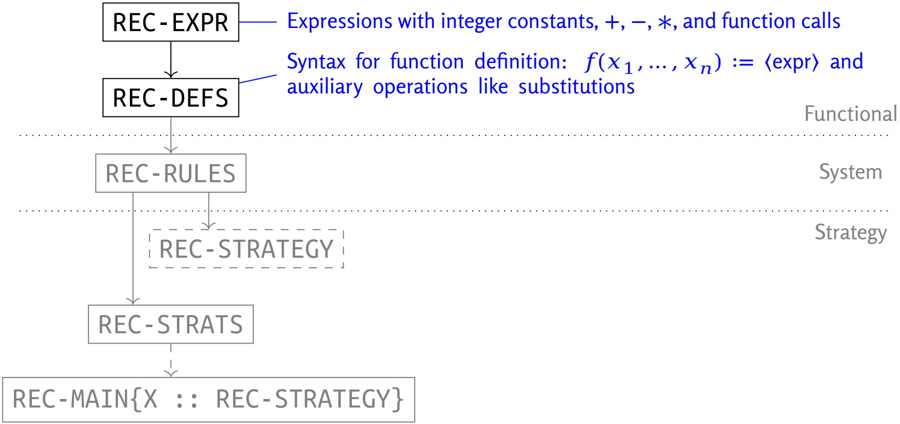
	Applicative	Normalizing	By name	By value
$(\mathbf{K}\,\mathbf{I})\Omega$	Does not terminate	$\mathbf{I}$	$\mathbf{I}$	$(\mathbf{K}\,\mathbf{I})\Omega$
$\lambda y.(\mathbf{I}z)$	$\lambda y.z$	$\lambda y.z$	$\lambda y.(\mathbf{I}z)$	$\lambda y.(\mathbf{I}z)$
$(\mathbf{K}z)t$	z	z	z	$(\mathbf{K}z)t$
$\mathbf{K}z$	$\lambda y.z$	$\lambda y.z$	$\lambda y.z$	$\lambda y.z$

REC language

from Chapter 9 of Winskel's *The Formal Semantics of Programming Languages*

REC language

from Chapter 9 of Winskel's *The Formal Semantics of Programming Languages*

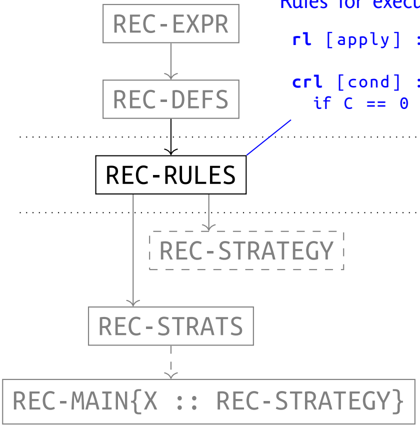


REC language

from Chapter 9 of Winskel's *The Formal Semantics of Programming Languages*

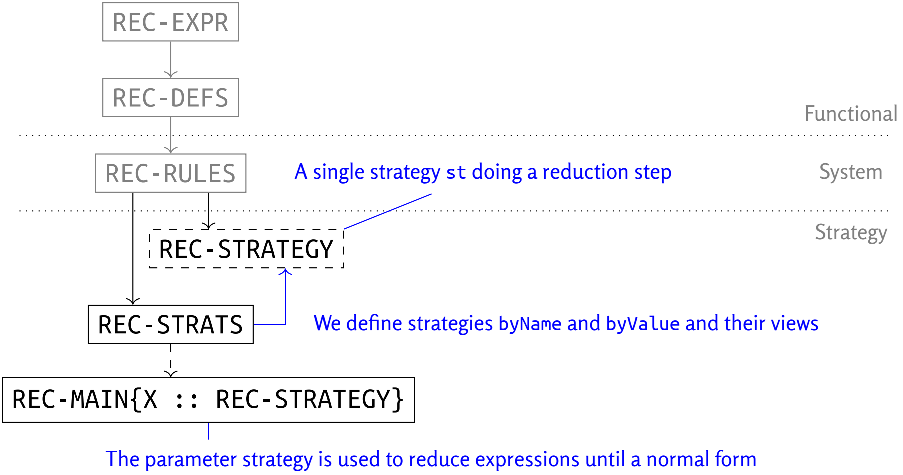
Rules for executing conditionals and function calls:

```
rl [apply] : Q(Args) ⇒ apply(find(Q, Defs), Args) .

crl [cond] : if C then E else F ⇒
  if C == 0 then E else F fi if C : Int .
```



REC language

from Chapter 9 of Winskel's *The Formal Semantics of Programming Languages*



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 $\alpha\,! \equiv \alpha * ; (\alpha$? `fail` : `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
```

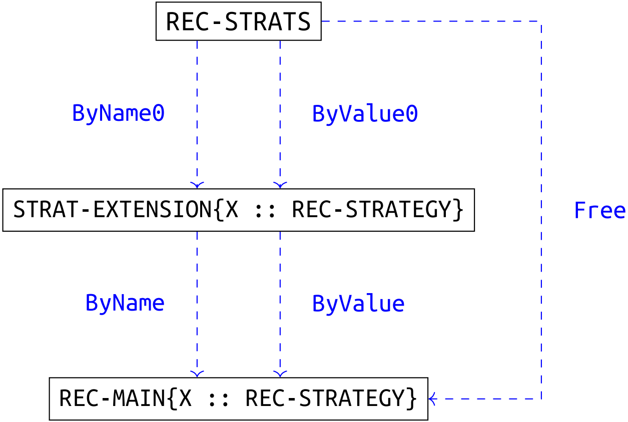
- byname and byvalue definitions only deal with function calls.

REC language – strategy code reuse

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



REC language – strategy code reuse

```
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

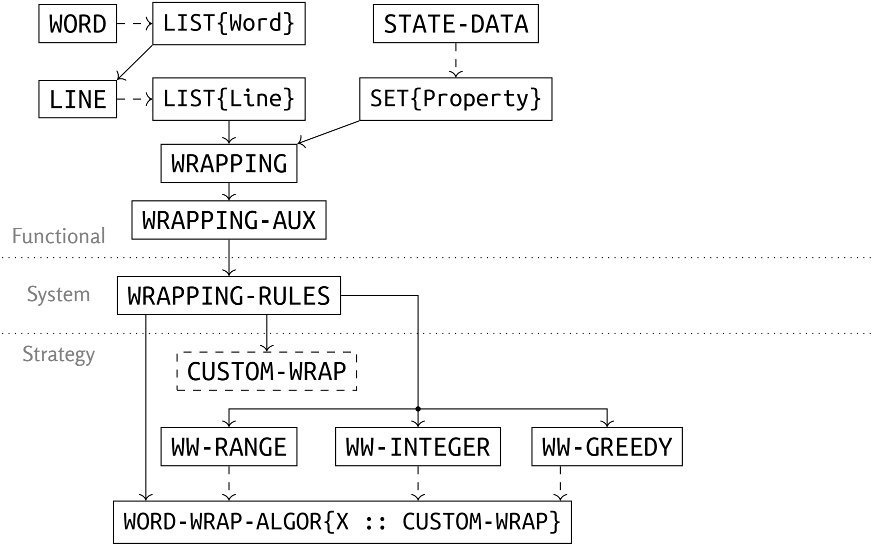
$$
\begin{aligned}
f(x) &= \texttt{if } x \texttt{ then } 1 \texttt{ else } (x * f(x-1)) \\
g(x,y) &= g(y,x) \\
h(x) &= 3
\end{aligned}
$$

- 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.

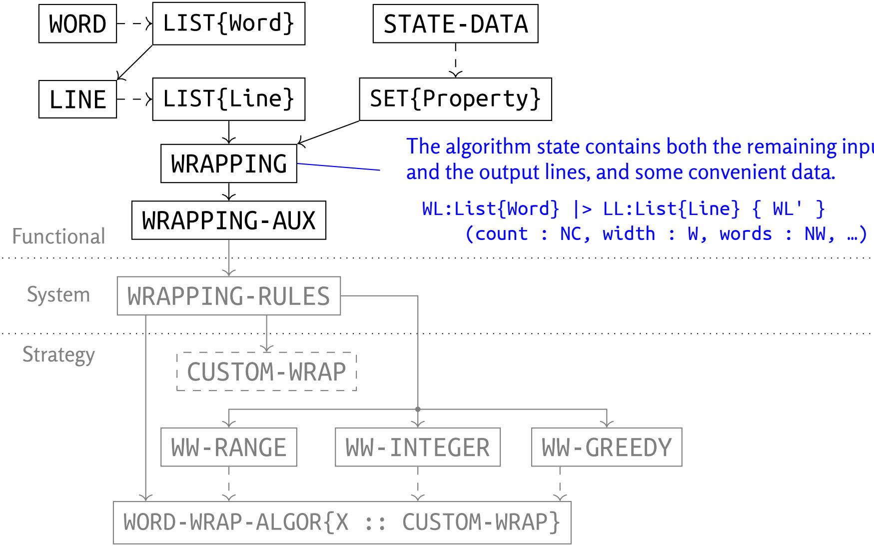
Line breaking algorithm

- 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



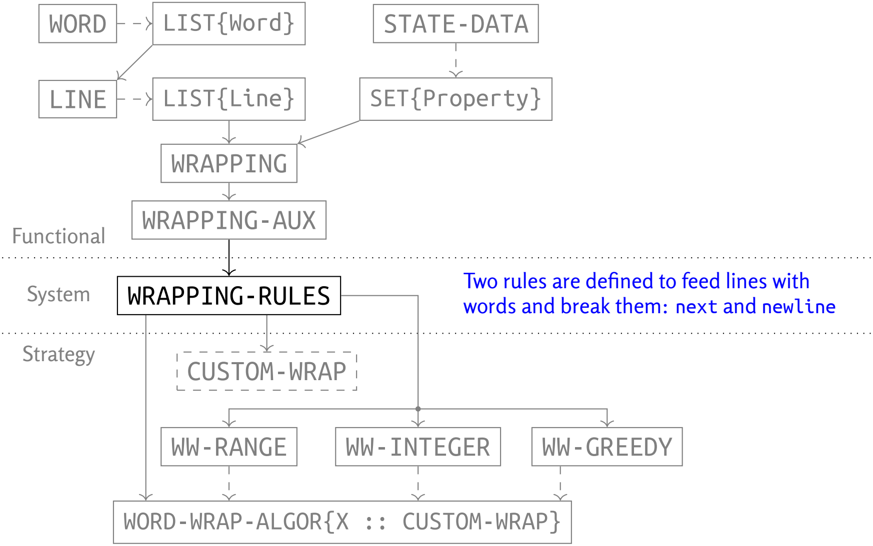
Line breaking algorithm



The algorithm state contains both the remaining input and the output lines, and some convenient data.

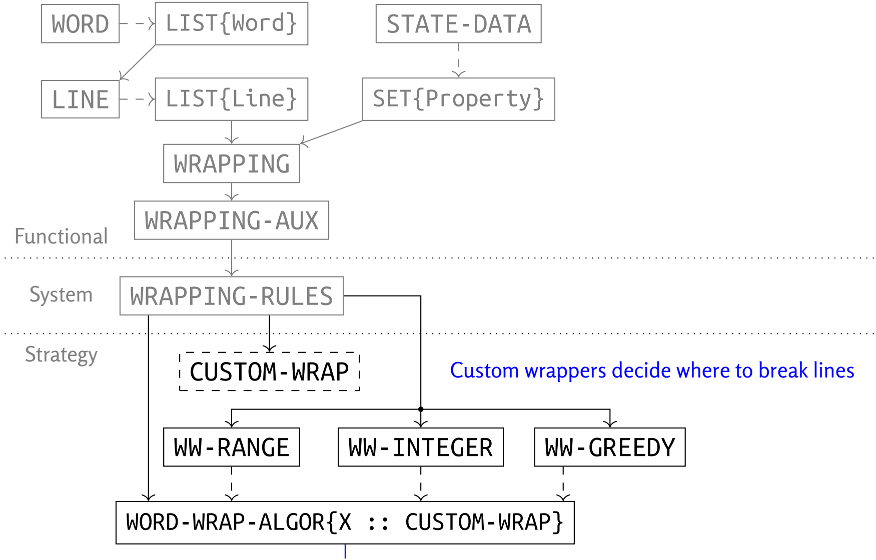
```
WL:List{Word} |> LL:List{Line} { WL' }
    (count : NC, width : W, words : NW, …)
```

Line breaking algorithm



Two rules are defined to feed lines with words and break them: `next` and `newline`

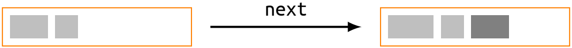
Line breaking algorithm



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.



$$[\![\texttt{next}]\!](\theta, t_0) = \{t_1\} \qquad [\![\texttt{newline}]\!](\theta, t_1) = \{t_2\}$$

Typically, `break` will do the following:

$$[\![\texttt{break}]\!](\theta, t_1) = \begin{cases} \{t_1\} & \texttt{idle} \\ \{t_1, t_2\} & \texttt{idle|newline} \\ \{t_2\} & \texttt{newline} \\ \emptyset & \texttt{fail} \end{cases}$$

Line breaking algorithm

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

```
sd range(Min, Max) := match S s.t. numberWords(S) == 1
                                 or spaceWidth(S) ≥ Min ;
  (
    (match S s.t. spaceWidth(S) ≤ Max ; newline)
    | idle
  ) .
```

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

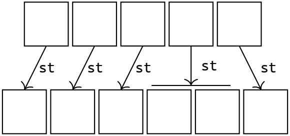
```
sd wrap(RG:Nat) :=  (match nil |> LL (SD) ? idle
  : next; *** Adds a new word to the line
    break; *** Should we break here? (the parameter strategy)
    match WL |> LL (raggedness : N, SD) s.t. N ≤ RG; *** Prune
    wrap(RG)
  ) .
```

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.

```
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

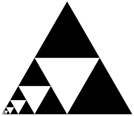
flatMap can be implemented with extra rules, for example.

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

  rl [empty] : nil ⇒ nil .
  crl [nonempty] : E L ⇒ E' L' if E ⇒ E' ∧ L ⇒ L' .
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
```

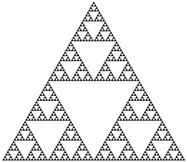
Flat map – fractals

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

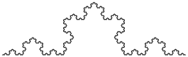
```
crl [von-koch] : A >> B ⇒  A >> C  C >> E  E >> D  D >> B
   if C := fractionPoint(A, B, 1.0 / 3.0)
   ∧ D := fractionPoint(A, B, 2.0 / 3.0)
   ∧ E := equilateralThird(C, D) .
```



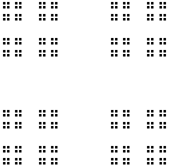
- Free rule application is not convenient.
- We can use flatMap.



Sierpinski triangle



Von Koch curve



Cantor dust

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 sucessors
  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

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

  var S : BBState . var F : X$FixData . var V : X$Value .
  var P : X$PartialResult . vars N M : Nat .

  strat solve iteration @ BBState .
  strat iterChildn : Nat X$PartialResult X$FixData X$Value @ BBState .

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

  sd iteration := matchrew S s.t. M := numChildren(top(S), fixData(S))
         ∧ M > 0 by S using (pop ;
         iterChildn(sd(M, 1), top(S), fixData(S), upperBound(S))) .

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

Branch and bound – travelling salesperson problem

- `PartialResults` 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 analisys 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

Parameterized strategy specification with Maude
Narciso Martí-Oliet, Isabel Pita, Rubén Rubio, Alberto Verdejo
{narciso,ipandreu,rubenrub,jalberto}@ucm.es