Building theorem provers using rewriting logic.

Carlos Olarte
ECT - UFRN, Natal, Brazil

Natal, August 2019

Why Rewriting Logic?

(Naive system) for bounded contraction.

$$\frac{\Delta, A^{k+1} \vdash C}{\Delta, A^{k} \vdash C}\ \mathsf{k}$$

[CGT08] rule

$$\frac{\Delta, \Delta_1^{k+1} \vdash C \quad \ldots \quad \Delta, \Delta_1^{i_1}, \ldots, \Delta_k^{i_k} \vdash C \quad \ldots \quad \Delta, \Delta_k^{k+1} \vdash C}{\Delta, \Delta_1, \ldots, \Delta_k \vdash C}\ \mathsf{k}$$

[CLOP16] rule (lazy system)

$$\frac{\Upsilon_1 \qquad \Upsilon_2 \qquad \cdots \qquad \Upsilon_k}{\boxed{L}; \Delta; E \vdash C \ / \ \boxed{L^k}; E^k}\ \mathsf{k}$$

where

$$\Upsilon_1 = \boxed{L :: \boxed{\cdot}; \boxed{\cdot}; \boxed{\Delta :: E}}; \cdot; \cdot \vdash C \ / \ \boxed{L^1 :: \boxed{\Gamma^1}; \boxed{\Omega_\Delta :: \Omega_E}; \boxed{\Delta^1 :: E^1}}; \cdot$$

...

Logical Frameworks

Consider the following inference rule:

$$\frac{\Gamma \vdash F \quad \Delta \vdash G}{\Gamma, \Delta \vdash F \otimes G} \otimes_R$$

Horn Clauses (Prolog)

```
prove Upsilon (F tensor G) :- split Upsilon Gamma Delta,
                              prove Gamma F, prove Delta G .
```

Rewriting Logic (Maude)

```
rl [tensorR] : Gamma, Delta |- F x G =>
               (Gamma |- F) , (Delta |-G) .
```

Gap between what is represented and its representation

Rewriting Logic can rightfully be said to have "ϵ-representational distance" as a semantic and logical framework.(José Meseguer)

Where is the Magic ?

Rewriting logic: Equational theory + rewriting rules

- Linear Logic (no weakening / contraction)
```
op _,_ : Context Context -> Context[assoc comm id: empty]
```
- Propositional logic
```
op _,_ : Context Context -> Context[assoc comm id: empty]
 --- idempotency / contraction
 eq F:Formula, F:Formula = F:Formula .
```
- Lambek calculus without exchange

```
op _,_ : Context Context -> Context [assoc  id: empty] .
```

The general point is that, by choosing the right equations , we can capture any desired structural axiom. (José Meseguer)

Determinism vs Non-Determinism

Back to the tensor rule:

$$\frac{\Gamma \vdash F \quad \Delta \vdash G}{\Gamma, \Delta \vdash F \otimes G}\ \otimes_R \qquad \frac{\Gamma, F_1, F_2 \vdash G}{\Gamma, F_1 \otimes F_2 \vdash G}\ \otimes_L$$

Equations

Deterministic (invertible) rules that can be eagerly applied.

```
eq [*L] : Gamma, F1 * F2 |- G    =    Gamma, F1 , F2 |- G .
```

Rules

Non-deterministic (non-invertible) rules where backtracking is needed.

```
rl [*R] :Gamma, Delta |- F x G =>
          (Gamma |- F), (Delta |-G) .
```

RL distinction between equations and rules gives a specifier additional expressive power to discriminate between deterministic and nondeterministic inference rules. (José Meseguer)

Motivation

Rewriting Logic is a simple computational logic that can naturally express both concurrent computation and logical deduction.

One can think of [RL] as an "ecumenical movement" with no sectarian ax to grind: it makes no commitments to specific concurrency mechanisms. Is it better to be synchronous or asynchronous? Is message-passing the best communication mechanism? [...] Rewriting logic remains politely silent about the choices made in each model, but tries to be as flexible as possible in representing different choices. (José Meseguer).

This tutorial...

- Slides: I will partially use the material from the school on Rewriting System at PUJ (`https://www.javerianacali.edu.co/eventos/international-school-rewriting-2018`)
- I will use simple examples. Some real tools in the end of the presentation.
- Recommended reads:
  - José Meseguer: Twenty years of rewriting logic. J. Log. Algebr. Program. 81(7-8): 721-781 (2012).
  - Peter Csaba Ölveczky: Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude. Springer 2017.
- Maude: `http://maude.cs.illinois.edu/`

Outline

Outline

Rewriting Logic: Computational and logical side

- Unified modeling formalism for transition systems
  - expressiveness (data representations and dynamic behavior)
  - executability and verifiability
  - generic and language-specific formal analysis tools in Maude

a semantic framework in which different models of concurrency, systems, programming languages, etc can be naturally represented, executed and analyzed.

- Logical framework (and meta-logic)
  - create executable environments for different logics
  - advanced meta-programming and metalanguage applications

a logical framework where different logics and automated deduction procedures can be represented, mechanized, and reasoned about.

Outline

Rewrite Theories

A rewrite theory is the specification unit in rewriting logic. It defines:

- A signature (to build terms)
- A set of equations and structural axioms (deterministic computations).
- A set of (conditional) rewriting rules (concurrent/dynamic behaviors).

Equational theories (algebraic specifications)

Definition (Equational theory (Σ, E))

- Σ, the signature, describes the syntax of the theory: types and the operation/function symbols involved;
- E is a set of equations between expressions (called terms) in the syntax of Σ.

Some variants depending on the types (called sorts) supported:

- unsorted (or single-sorted) signatures have only one sort
- many-sorted signatures allow different sorts (`Integer`, `Bool`, etc).
- order-sorted signatures are many-sorted signatures that, in addition, allow inclusion relations between sorts, such as `Nat` < `Int`.

Maude

- A high-performance rewriting logic engine
- A system module defines a rewrite theory $\mathcal{R}$.
- Executes admissible system modules (confluence and termination of E, coherence of R w.r.t. E, ...)
- Several generic formal analysis tools
  - rewrite, search, LTL model checker, ...
  - ITP, CRC, SCC, etc.
- Extended with SMT solver (CVC and Yices).

Hubert Garavel and collaborators benchmarked several functional and rule-based declarative languages. Although Maude is an interpreted language, it ranks second in overall performance, closely after Haskell.

Maude Functional Modules

The theory (Σ, E) is declared with the syntax

```
fmod NAME is
 Sigma --- Signature
 Eqs   --- equations
endfm
```

Such theories can be seen as functional programs that compute with the equations Eqs.

Example: Natural Numbers

```
fmod NATURAL is
    sort Natural . --- Type / Sort
    --- Constructors for the data type
    op 0 : -> Natural [ctor] .
    op s : Natural -> Natural [ctor] .
    --- Functions/operations on Natural Numbers
    op plus : Natural Natural -> Natural .
    --- Logical Variables
    vars N M : Natural .
    --- Defining equations for Plus
    eq plus(N,0) = N .
    eq plus(N,s(M)) = s(plus(N,M)) .
endfm --- file: 1-nat.maude

--- Some tests and commands
parse s(s(0)) .
red plus(s(s(0)),s(s(0))) .
```

Mixfix Notation

```
fmod NATURAL is
    --- [...]
    op _+_ : Natural Natural -> Natural .
    op _*_ : Natural Natural -> Natural .
    --- Logical Variables
    vars N M : Natural .
    --- Defining equations for Plus
    eq N + 0 = N .
    eq N + s(M) = s(N + M) .
    --- Equations for Times
    eq N * 0 = 0 .
    eq N * s(M) = N + (N * M) .
endfm --- file: 2-nat.maude

--- Some tests
red s(s(s(s(0)))) + s(s(0)) .
red s(s(s(s(0)))) * s(s(0)) .
```

Order-Sorted Signatures

Defining partial operators and Kinds

```
fmod Nat is
   sorts Nat NzNat .
   subsorts NzNat < Nat . --- subsort relation
   op 0 : -> Nat [ctor] .
   op s : Nat -> NzNat [ctor] .
   op p : NzNat -> Nat .
   vars N M : Nat .
   eq p(s(N)) = N .
endfm --- file: 3-nat-pred.maude
-------
red p(0) .
result [Nat]: p(0) --- a Kind! ( [Nat] )
red p(s(s(0))) .
result NzNat: s(0) --- The least sort.
red p(s(0)) .
result Nat: 0
------
```

Order-Sorted Signatures

Order-sorted signature $\Sigma = ((S, <), F)$

$<$ is irreflexive and transitive; (S, F) is a many-sorted signature.

Ad-hoc overloading

```
op _+_ : Nat Nat -> Nat .
op _+_ : Bool Bool -> Bool.
```

subsort overloading:

```
op _+_ : Nat Nat -> Nat .
op _+_ : Nat NzNat -> NzNat . --- More information
```

Kind completion and making sense of errors

- For each connected component $[s]$ of $(S, \leq)$ a fresh sort $\top_{[s]}$ is added (denoting errors). Each operator f is also lifted to $f : \top_{[s1]} \ldots \top_{[sn]} \rightarrow \top_{[s]}$.

Outline

Terms and equational theories

- Terms are expressions built from the signature.
- A ground term is a term without variables.
- In (Σ, E), all equations $t = t' \in E$ are (implicitly) universally quantified:

$$(\forall x_1 : s_1, ..., x_n : s_n) t = t'$$

- Equational deduction is the systematic replacement of equals by equals using the given equations E. For instance, in a ring:

$$y + (z + [0 + (1 * x)]) = y + (z + [(1 * x) + 0]) = y + (z + [1 * x])$$
$$= [y + (z + x)] = (y + z) + x$$

- Term rewriting: equations are always applied from left to right. Proofs are then the sequence of steps needed to show that $t \rightarrow^* t'$.

Terms and equational theories

- ... but for the equality relation $=_E$ we need both, $E_{\rightarrow}$ and $E_{\leftarrow}$.
- Certain equations are problematic, e.g.,:$x + y = y + x$.
- Solution: rewriting modulo axioms (e.g., commutativity, associativity and identity).
- Equational theory: (Σ, E, B) where $B \subseteq \{\texttt{comm}, \texttt{assoc}, \texttt{id}\}$.

Let's see how to build the typical data structures needed to represent sequents (and any system).

Lists

```
fmod MY-LIST is
 protecting NAT .
 sorts NeList List . --- List and non-empty list of Nat's

 --- A Nat IS a (unary) list
 subsort Nat < NeList < List .

 op nil : -> List [ctor] .
 --- __ is the empty syntax (e.g., 3 2 4 is a list!)
 op __ : List List -> List [ctor assoc id: nil prec 25] .
 --- Ditto: same attributes from the previous declaration
 op __ : NeList List -> NeList [ctor ditto] .
 op __ : List NeList -> NeList [ctor ditto] .

 vars E E' : Nat . vars L' L : List . var C : Nat .
```

Lists (II)

```
--- Append in lists
op append : List List -> List .
op append : NeList List -> NeList .
op append : List NeList -> NeList .
eq append(L', L) = L' L .
--- First element of the  list
op head : NeList -> Nat .
eq head(E L) = E .
--- Tail of the list
op tail : NeList -> List .
eq tail(E L) = L .
--- Note the neat way of picking the last element
op last : NeList -> Nat .
eq last(L E) = E .
--- Removing the last element
op front : NeList -> List .
eq front(L E) = L .
```

Lists (III)

```
    --- In List
    op occurs : Nat List -> Bool .
    eq occurs(E, nil) = false .
    eq occurs(E, E' L) =
       if E == E' then true else occurs(E, L) fi .
endfm
parse 3 . --- NzNat: 3
--- But it can be also a list
red occurs(3 , 3) . --- result Bool: true
--- (Assoc) We don't need parentheses!
red 3 (2 1) 4 3  . --- result NeList: 3 2 1 4 3
--- Id: (canonical representation)
red  3 nil 2 nil 1 nil nil . --- result NeList: 3 2 1
```

Parametric Modules

Of course, the list must work for any arbitrary data type.

```
--- Functional theory
fth TRIV is
  sort Elt .
endfth
--- View
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
--- Parametric module
fmod LIST{X :: TRIV} is
  sorts NeList{X} List{X} .
  subsort X$Elt < NeList{X} < List{X} .

  op nil : -> List{X} [ctor] .
  ...
```

Multisets

```
fmod MMSet is
 protecting NAT .
 sorts NeMSet MSet .
 subsort Nat < NeMSet < MSet .

 op empty : -> MSet [ctor] .
 op _,_ : MSet MSet -> MSet [ctor assoc comm id: empty] .
 op _,_ : NeMSet MSet -> NeMSet [ctor ditto] .
 ...
 op insert : Nat MSet -> MSet .
 eq insert(E, S) = E, S .
```

Multisets (II)

```
--- Owise: if none of the previous rules applies
op delete : Nat MSet -> MSet .
eq delete(E, (E, S)) = S .
eq delete(E, S) = S [owise] .

--- Due to commutative (and associativity),
--- note the simplicity of this definition
op _in_ : Nat MSet -> Bool .
eq E in (E, S) = true .
eq E in S = false [owise] .

op union : MSet MSet -> MSet .
   op union : NeMSet MSet -> NeMSet .
   op union : MSet NeMSet -> NeMSet .
   eq union(S, S') = S, S' .

endfm
```

Multisets (III)

```
---some tests
--- Note that the order is irrelevant
red 3 , 2 , 5 , 3 ,5 . --- result NeMSet: 2, 3, 3, 5, 5
red union( (1,2,3) ,  (2,3,4) ) . --- result NeMSet: 1, 2, 2,
red 3 in (1,2,3) . --- result Bool: true
red delete (3, 3) . --- result MSet: empty
```

Sets

```
fmod MY-SET is
    protecting NAT .
    sorts NeSet Set .
    subsort Nat < NeSet < Set .

    op empty : -> Set [ctor] .
    op _,_ : Set Set -> Set [ctor assoc comm id: empty] .
    op _,_ : NeSet Set -> NeSet [ctor ditto] .
    var E : Nat .
    --- Idempotency: quite easy due to comm and assoc of ","
    eq E , E = E .
    ...
endfm

red 3 , 2 , 5 , 3 ,5 . --- result NeSet: 2, 3, 5
red union( (1,2,3) , (2,3,4) ) . ---result NeSet: 1, 2, 3, 4
```

Outline

Executability conditions

- In (Σ, E, B), we oriented the equations as $\vec{E}$ leading to a rewrite theory $(\Sigma, \vec{E}, B)$ (that simplifies terms).
- ... but, if $t \to^* t'$, is it the case that $t =_{B \cup E} t'$?
- This is the case if 4 conditions are met.

Executability conditions.

Let $t \to^* t' \in R$.

1. $vars(t') \subseteq vars(t)$.
2. R is confluent modulo B.
3. R is sort-decreasing: $(t\theta : s$ then $t'\theta : s)$ — No lost of information
4. R is terminating modulo B.

Executability conditions

- The system is deterministic: answers, regardless the reduction strategy, are unique.
- For all term t, there exists a normal form t' s.t. $t \to t'$ and $t' \not\to$. Such normal (or canonical) forms are unique.

Some additional conditions on B:

- A B-Matching algorithm must exist (given a complete set of unifiers for any terms t, t').

In order to check $t =_{E \cup B} t'$, it is enough to compute the normal forms $can(t)$ and $can(t')$ and check if they are equivalent (modulo B) which is decidable due to the B-Matching algorithm.

Functionality

The canonical forms are the values computed by $(\Sigma, B, \vec{E})$. Those are computed by the `red` command in Maude.

Outline

Rewrite Rules

Definition (Rewrite Theory)

A tuple $\mathcal{R} = (\Sigma, E \cup B, R)$ consisting of:

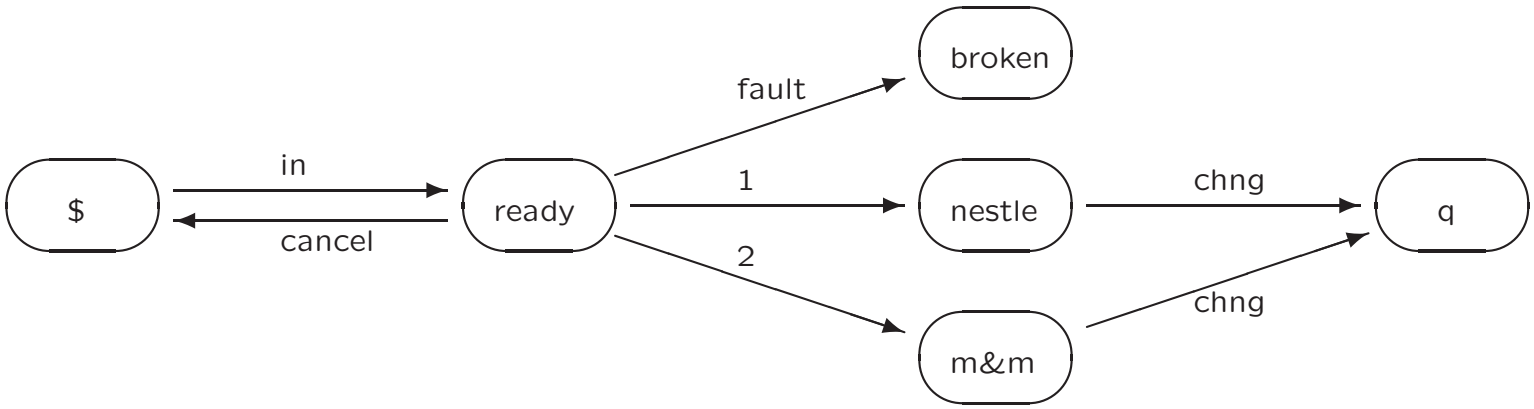
- $(\Sigma, E \cup B)$ is an equational logic theory
- B is a set of structural axioms (assoc, comm, id).
- R is a set of labeled conditional rewrite rules $l \longrightarrow r$ `if` C
- $(\Sigma, E \cup B)$ specifies states and deterministic computations
- R specifies dynamic, concurrent behaviors

The most general form of conditional rules:

```
rl [name] : t => t' .
crl [name] : t => t' if cond .
--- (also for equations)
eq t = t' .
ceq t= t' if cond .
```

System Modules

A non-deterministic system




```
mod CANDY-AUTOMATON is
    sort State .
    ops $ ready broken nestle m&m q : -> State .
    rl [in] : $ => ready .  rl [cancel] : ready => $ .
    rl [1] : ready => nestle .
    rl [2] : ready => m&m .
    rl [fault] : ready => broken .
    rl [chng] : nestle => q .
    rl [chng] : m&m => q .
endm
```

Some useful commands

```
rew $ . --- rewrite ...  result State: q
rew [1] $ . --- 1 step of rewriting
--- Breadth first search
search $ =>* m&m . --- searching ... Solution 1 (state 3)
search $ =>* S:State . --- 0 or more steps
search $ =>+ S:State . --- 1 or more steps
search $ =>! S:State . --- terminating states
--- conditions
search $ =>* S:State such that S:State =/= m&m .
```

Rewrite rules as transitions

- Rewrite rules do not have an equational interpretation. They are understood as transitions that in general cannot be reversed.
- E and R are totally different, with different semantics.

Let $\mathcal{R} = (\Sigma, E \cup B, R)$. $\mathcal{R}$ proves sequents of the form $t \rightarrow t'$ ($\mathcal{R} \vdash t \rightarrow t'$) where $t, t' \in T_{\Sigma(X)}$ and $\rightarrow$ is obtained from :

- Reflexivity and transitivity:

$$\frac{}{t \rightarrow t} \qquad \frac{t_1 \rightarrow t_2 \qquad t_2 \rightarrow t_3}{t_1 \rightarrow t_3}$$

- Equality:

$$\frac{u \rightarrow v \qquad E \cup B \vdash u = u' \qquad E \cup B \vdash v = v'}{u' \rightarrow v'}$$

- Congruence and replacement (for a rule $r : t \rightarrow t'$):

$$\frac{t_1 \rightarrow t_1' \cdots t_n \rightarrow t_n'}{f(t_1, ..., t_n) \rightarrow f(t_1', ..., t_n')} \qquad \frac{x_1\theta \rightarrow x_1' \cdots x_n\theta \rightarrow x_n'}{t(\vec{x})\theta \rightarrow t'(\vec{x})[x_1'/x_1, ..., x_n'/x_n]}$$

Rewrite rules as transitions (II)

Computationally

a rewrite theory specifies a concurrent system, whose set of states is the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies a parameterized family of concurrent transitions in the system.

Logically

a rewrite theory specifies a logic, whose set of formulas is the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies an inference rule in the logic.

Now we are ready to specify our first theorem prover!

Outline

Outline

Classical Propositional Logic

We shall implement a proof search procedure (bottom-up) for the system G3cp.

State — Equational part

- Formulas, including atomic propositions.
- Contexts (just sets of formulas)
- Sequents: $\Gamma \vdash \Delta$.

Some examples of rules:

$$\frac{}{\Gamma, p \vdash p, \Delta}\ I \qquad \frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \wedge G \vdash \Delta}\ \wedge_L \qquad \frac{\Gamma \vdash F, \Delta \quad \Gamma \vdash G, \Delta}{\Gamma \vdash F \wedge G, \Delta}\ \wedge_L$$

System G3cp (Equational part)

```
fmod FORMULA is
 pr QID . --- Maude's Qualified Identifiers ('a,'b, etc).
 sort Prop . --- Atomic propositions
 sort Formula . --- Formulas
 subsort Qid < Prop < Formula .
 --- Constructors
 op _/\_ : Formula Formula -> Formula [ctor assoc prec 30] .
 op _\/_ : Formula Formula -> Formula [ctor assoc prec 40] .
 op _->_ : Formula Formula -> Formula [ctor prec 50] .
 op False : -> Formula [ctor] .
 --- Other connectives
 vars F G : Formula .
 op ~_ : Formula -> Formula [prec 25] .
 eq ~ F = F -> False .
 op _<->_ : Formula Formula -> Formula [prec 55] .
 eq F <-> G = (F -> G) /\ (G -> F) .
endfm
```

System G3cp (Equational part) (II)

```
--- Contexts as sets of formulas:
view Formula from TRIV to FORMULA is
    sort Elt to Formula .
endv

--- Sets of formulas
fmod Context is
    --- Inclusion + Renaming
    pr SET{Formula} * (sort Set{Formula} to Context ,
                       sort NESet{Formula} to NEContext).
endfm
```

System G3cp (Equational part) (II)

```
fmod Sequent is
 pr Context .
 --- Sequents and list of sequents
 sorts Sequent SSequent .
 subsort Sequent < SSequent .

 --- constructors
 op proved : -> Sequent [ctor] .
 op _|--_ : Context Context -> Sequent [ctor prec 122] .

 --- List of unproved sequents
 op _||_ : SSequent SSequent -> SSequent
   [ctor assoc id: proved] .
endfm
```

System G3cp (Rules)

```
mod G3cp is
 pr Sequent .
 var P : Prop . vars C C' : Context . vars F G : Formula .

 rl [I] : P, C |-- P, C'
          ------------
       => proved .

 rl [AndL] : F /\ G , C |-- C'
             -----------------
           => F, G , C |-- C' .

 rl [AndR] :        C |-- F /\ G, C'
             -------------------------------
         => (C |-- F, C') || (C |-- G , C') .
...
```

System G3cp (Some test)

```
search [1] 'a /\ 'b |-- 'b /\ 'a =>* proved
Solution 1 (state 10)
show path 10 . --- Showing the proof
state 0, Sequent: 'a /\ 'b |-- 'b /\ 'a
===[ rl C, F /\ G |-- C' => F, C, G |-- C' [label AndL] .]===>
state 1, Sequent: 'a, 'b |-- 'b /\ 'a
===[ rl C |-- C', F /\ G => (C |-- C', F) || ..) [label AndR]
state 3, SSequent: ('a, 'b |-- 'b) || ('a, 'b |-- 'a)
===[ rl P, C |-- P, C' => proved [label I] . ]===>
state 6, Sequent: 'a, 'b |-- 'a
===[ rl P, C |-- P, C' => proved [label I] . ]===>
state 10, Sequent: proved
--- However... this one did not finish in 2 hours...
search [1]
  empty |--('a\/ ('b -> 'c)/\ ~ ('d/\ 'e/\ 'f)\/~ ('x<->'y))
       <-> ('a\/ ('b -> 'c)/\ ~ ('d/\ 'e/\ 'f)\/~ ('x<->'y))
 =>* proved .
```

System G3cp (Test)

- The `search` command uses a breath-first-search strategy (thus guaranteeing completeness).
- In the system G3cp, all the rules are invertible (and the strategy is irrelevant).
- ... then... we can do better!

```
--- just rewrite
rew empty |-- ('a \/ ('b -> 'c) /\ ~ ('d /\ 'e /\ 'f) \/ ....
rewrites: 293 in 0ms cpu (0ms real) (20928571 rewrites/second)
result Sequent: proved

rew 'a /\ 'c |-- 'b .
result Sequent: 'a, 'c |-- 'b
```

- The same theory $\mathcal{R}$ can be executed with many different strategies.
- Strategies, being a particular way of applying correct rules, can never affect correctness (but completeness).

Outline

System G3ip

Consider the system G3ip for intuitionistic logic:

- We have invertible rules ($\wedge_L$, $\wedge_R$, $\rightarrow_R$, $\vee_L$).
- but also some non invertible rules:

$$\frac{\Gamma \vdash F_i}{\Gamma \vdash F_1 \vee F_2} \vee_R$$

- We may also have loops (Dyckhoff's LJT solves this problem):

$$\frac{\Gamma, A \rightarrow B \vdash A \quad \Gamma, B \vdash G}{\Gamma, A \rightarrow B \vdash G} \rightarrow_L$$

- So we have to think better how to encode the system.

System G3ip

One possible solution....

- The invertible rules will be specified as equations.
- However, the system with list of Sequents/goals is not longer confluent.
- We shall replace the list of goals with a set of goals. And then, we recover confluence.
- With sets, the `search` command will be able to avoid loops (due to implication-left) for free.

Reduction strategy: Equations vs Rules

Maude uses the following strategy:

- A term t is normalized to t_c (using the set of equations E).
- A rule that matches t_c is applied and the term rewritten to t'.
- The resulting term t' is again normalized.
- This strategy is complete if the executability conditions (for equations) are met and the theory is ground coherent:

$$\mathcal{R} \vdash t \rightarrow^1 t' \text{ then } \exists t'', \mathcal{R} \vdash can_{E/B}(t) \rightarrow^1 t'' \text{ and } [t'] = [t'']$$

- The strategy above meets our intuition for G3ip:
  - Apply eagerly the invertible rules;
  - after that, apply a non-invertible rule (decide).

System G3ip (Equations)

```
--- Formulas as in G3cp.
--- op _|--_ : Context Formula -> Sequent [ctor prec 122] .

mod G3ip is
 pr SET{Sequent} .
 var P : Prop . var C : Context . vars F G H : Formula .
 --- Equational rules
 eq  P, C |-- P = proved.
     -------------
  = proved [label I].

 eq  F /\ G , C |-- H
     -----------------
  = F, G , C |-- H [label AndL] .
```

System G3ip (Rules)

```
rl [orR1] : C |-- F \/ G
            ----------------
          => C |-- F .

rl [orR2] : C |-- F \/ G
            ----------------
          => C |-- G .

rl [ImpL] :          C, F -> G |-- H
            ---------------------------------
          => (C, F -> G |-- F) , (C, G |-- H) .
```

System G3ip

Many things to improve/think about this little example:

- The choice of sets of sequents avoids loops:

```
search [1] 'p -> 'q |-- 'p =>* empty .
No solution.
states: 2  rewrites: 3 in 0ms cpu (0ms real) ...
```

- But we lose control of the branches of the proof. Maybe a tree structure instead of sets can help.
- Due to the breath-first-search strategy, solving subgoals is not completely independent.... but it should.
- So... better strategies/heuristics are needed.

Outline

SAT

- Tinelli's separation of concerns: (correct) inference system and (sophisticated) heuristics.
- The inference system (RWL specification) may have different (more or less efficient) executions.
- A clear distinction between deterministic and non-deterministic rules is a good starting point.
- In the following rules, in $\Delta \vdash \Gamma$, Δ is a set of literals and Γ a set of clauses.
- Γ is satisfiable iff from $\varnothing \vdash \Gamma$ we can derive $\Delta \vdash \varnothing$ (where Δ is the satisfying assignment).

Davis-Putnam-Logemann-Loveland SAT Procedure

Equational Rules

```
--- \/ is already commutative
op _\/_ : Clause Clause -> Clause [assoc comm id: ([])] .
op _,_ :ClauseSet ClauseSet->ClauseSet [assoc comm id: null] .
```

subsume: $\dfrac{\Delta \vdash \Gamma, l \vee C}{\Delta \vdash \Gamma}$ if $l \in \Delta$

```
eq [subsume] : l,CTX |- CS,(l \/ C) = l,CTX |- CS .
```

resolve: $\dfrac{\Delta \vdash \Gamma, l \vee C}{\Delta \vdash \Gamma, C}$ if $\neg l \in \Delta$

```
eq [res1] : p,CTX |- CS,( ~(p) \/ C) = p,CTX |- CS,C .
eq [res2] :  ~(p),CTX |- CS,(p \/ C) =  ~(p),CTX |- CS,C .
```

Davis-Putnam-Logemann-Loveland SAT Procedure

Equational Rules

close: $\dfrac{\Delta \vdash \Gamma, \Box}{\varnothing \vdash \Box}$ if $\Delta \neq \varnothing$ or $\Gamma \neq \varnothing$.

```
eq [close1] : CTX |- C,CS,[] = null |- [] .
eq [close2] : CTX,l |- CS,[] = null |- [] .
```

assert: $\dfrac{\Delta \vdash \Gamma, l}{\Delta, l \vdash \Gamma}$ if $l \notin \Delta, \neg l \notin \Delta$

```
ceq [assert] : CTX |- CS,l = CTX,l |- CS
      if (l in CTX) == false and ( ~(l) in CTX) == false .
```

Davis-Putnam-Logemann-Loveland SAT Procedure

Non-deterministic rules

split: $\dfrac{\Delta \vdash \Gamma, l \vee C}{\Delta, l \vdash \Gamma \quad \Delta, \neg l \vdash \Gamma, C}$ if $l \notin \Delta, \neg l \notin \Delta, C \neq \square$

```
crl [split1] : CTX |- CS,(l \/ C) => l,CTX |- CS
 if (l in CTX) == false
    and (~(l) in CTX) == false
    and C=/=[] .

crl [split2] : CTX |- CS,(l \/ C) =>  ~(l),CTX |- CS,C
  if (l in CTX) == false
    and (~(l) in CTX) == false
    and C =/= [] .
```

How can I implement the heuristics ?

Reflection

- A logic is reflective if it can faithfully represent its metalevel at the object level.
- RW can faithfully represent its own theories and their deductions via a finite rewrite theory $\mathcal{U}$ (the universal theory).
- For any finite $\mathcal{R}$ (including $\mathcal{U}$) we have

$$\begin{array}{ll} \mathcal{R} \vdash t \to t' & \text{iff} \\ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle & \text{iff} \\ \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \cdots & \end{array}$$

Reflection allows e.g., defining strategies, transformation tools, proving meta-logical properties, add/delete rules and reduce/normalize terms on the fly, etc.

```
Term: '_|-_['F:Formula,'_/\_['G1:Formula,'G2:Formula]]
```

How can I implement the heuristics ?

Strategy language

- Mechanism to control the nondeterministic behavior of $\mathcal{R}$.
- Strategies can be seen as proof tactics or solving heuristics.
- A strategy language SL defines, modularly and separately, strategy modules SM. Hence, what we get is a theory transformation:

$$(\mathcal{R}, SM) \mapsto SL(\mathcal{R}, SM)$$

where t in $\mathcal{R}$ becomes $s@t$ where s is a strategy expression in SM.

Building blocks

- Rule (labels) and matching with possibly extra conditions.
- Sequential composition, disjunctive choice, iteration, conditionals, etc.

Outline

An incomplete list of tools...

- SALC, Labeled Sequent Calculus for Description Logic (`https://github.com/arademaker/SALC`).
- Deep inference `http://ozan-k.com/software.html`
- Linear Logic and Simply Dependent Multimodal Linear Systems `https://github.com/carlosolarte/`
- Interactive theorem prover and a proof checker for Matching Logic `https://github.com/xc93/matching-logic-prover/tree/master/prover`

We managed to implement the proof checker in ~100 Maude statements that fit just well in two pages (Xiaohong Chen)

An incomplete list of tools...

- Structural Proof Theory in Maude (using symbolic techniques)
- A tool for proving inductive properties of software
  `http://imi.kyushu-u.ac.jp/~daniel/citp.html`
- Circ: Prover for coinduction and structural induction
  `http://fsl.cs.illinois.edu/index.php/Special:CircOnline`
- Constructor-based Inductive Theorem Prover (CITP)
  `https://github.com/ittutu/CITP.git`
- The Maude Resolution Theorem Prover
  `http://maude.cs.uiuc.edu/tools/rtp/`
- Maude Inductive Theorem Prover (ITP)
  `http://maude.cs.uiuc.edu/tools/itp/`
- Maude-NRL Protocol Analyzer
- Maude Coherence and Termination tools.

Concluding Remarks

Rewriting logic as a powerful (meta-) logical framework:

- Almost zero representational distance.
- A clear distinction between deterministic computations and (non-deterministic) rules.

Reasoning techniques and tools

- Reduce, rewrite, search, model-checkers, etc.
- Meta-level module (reflective computation)
- Strategy language.

Maude is pure fun ;-)

Don't miss CIMPA'2020

Logic@Natal: School in Mathematical Logic and Applications

- Course 1: "Set Theory as an Axiomatic Theory", Valeria de Paiva.
- C. 2: "Universal Algebra, Equational Logic and Term Rewriting", José Meseguer.
- C. 3: "Introduction to Proof Theory and Applications", Dale Miller
- C. 4: "Proofs in Theories", Gilles Dowek
- C. 5: "General methods in proof theory for Structural and Modal Logics", Sonia Marin
- C. 6: "Rewriting Logic", Carolyn Talcott
- C. 7: "Narrowing-based Analysis", Santiago Escobar

`https://www.cimpa.info/`

November - December (to be defined soon) 2020 in Natal.

Thanks!