Coherent Logic *via* colog

colog primer

John Fisher
jrfisher@cpp.edu

2009 – 2014

This colog primer describes how to express coherent first-order logic formulas using colog code. Colog code is the machine code for a Skolem Machine [3].

1 Coherent logic theories

A coherent logic well-formed formula has a general form which can be expressed as follows.

$$\forall \hat{x}(a_1 \wedge \ldots \wedge a_m \rightarrow \exists \hat{y_1}(b_{11} \wedge \ldots \wedge b_{1k_1}) \vee \ldots \vee \exists \hat{y_n}(b_{i_n1} \wedge \ldots \wedge b_{i_nk_{i_n}})) \quad (1)$$

where $m, n >= 0$, each a_i and b_{jk} are atomic propositions, and any or all of the quantifiers may be vacant. If present, the universal variables $\hat{x} = \{x_1, \ldots, x_s\}$ have scope which is the entire formula, but the existential variables in any $\hat{y_j}$, if present, only have scope restricted to the particular disjunct. It is assumed that the formula is closed and that none of the universal variable appear among any of the existential variables.

When $m = 0$, formula (1) is conventionally written as

$$\forall\hat{x}(\top \to \exists\hat{y_1}(b_{11} \land \ldots \land b_{1k_1}) \lor \ldots \lor \exists\hat{y_n}(b_{i_n1} \land \ldots \land b_{i_nk_{i_n}})) \tag{2}$$

or also

$$\forall\hat{x}(\exists\hat{y_1}(b_{11} \land \ldots \land b_{1k_1}) \lor \ldots \lor \exists\hat{y_n}(b_{i_n1} \land \ldots \land b_{i_nk_{i_n}})) \tag{3}$$

On the other hand, if $n = 0$ then formula (1) is conventionally written as

$$\forall\hat{x}(a_1 \land \ldots \land a_m \to \bot) \tag{4}$$

A coherent logic *theory* is a finite sequence of coherent logic formulas, each one designated as either an *axiom* or as a *conjecture*.

Coherent logic is *super constructive*: Nothing can be concluded that has not been imposed by the axioms or conjectures of the coherent theory and whose computation cannot be justified by a terminating calculation of a Skolem Machine [3].

2 Translation to colog

This section offers several coherent logic theories and translations to colog code form. In the coherent logic theories an axiom is given terminated with period at the end, and a conjecture (query) with a question mark ? at the end.

Our intention in this section is to give examples covering the various formulas as axiom or query. The reader is advised to carefully check that each of the coherent logic formulas given as examples in this section matches one of the forms (1),(2),(3), or (4) from the first section, and how.

The proof tree figures were automatically generated by the colog GUI prover available at the website [1].

Example 1. This simple example illustrates how constants are captured into the colog domain ...

$p(a)$.

$\forall x\ q(x)$.

$\exists x\ q(x)$?

using a special predicate `dom`. The colog code ...

```
true => p(a), dom(a).

dom(X) => q(X).

q(X) => goal.
```

The colog deduction tree in Fig.1 shows how `dom` works to carry a constant (the `a` which appears in the first axiom) into a universal assertion (the second axiom).

$$\begin{tikzcd}
\mathtt{true}_0 \arrow[d, dash] \\
\mathtt{p(a)}_1 \arrow[d, dash] \\
\mathtt{dom(a)}_2 \arrow[d, dash] \\
\mathtt{q(a)}_3 \arrow[d, dash] \\
\mathtt{goal}_4
\end{tikzcd}$$

Figure 1: Example 1 tree

Example 2. This example illustrates how an existential quantifier can inject a new domain element.

$p(a)$.

$\forall x(p(x) \to \exists y\ q(x,y))$.

$\forall z\ r(z)$.

$\exists z(q(a,z) \wedge r(z))$?

colog code:

```
true => p(a), dom(a).

p(X) => dom(Y), q(X,Y). % N.B.

dom(Z) => r(Z).

q(a,Z), r(Z) => goal.
```

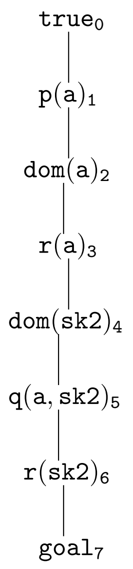

Figure 2: Example 2 tree

The thing to notice is how the `dom(Y)` in the second rule gives evidence for the existential variable and how the resulting constant can then be used to match the existential variable `Z` in the third colog rule.

The colog programmer can use any quantification predicate, but `dom` is recommended: The idea is that `dom(X)` in the antecedent represents a universal quantification (already belongs to domain) and `dom(X)` in the consequent represents existential quantification (will be added to domain).

Example 3. Existential variables may have common names, but still have limited scope – limited to adjacent conjunct.

$\exists y\ p(y) \vee \exists y\ q(y).$

$\forall z(p(z) \rightarrow r(z)).$

$\forall z(q(z) \rightarrow s(z)).$

$\exists w\ s(w) \vee \exists w\ r(w)?$

…colog code:

```
true => p(Y) | q(Y).

p(Z) => r(Z).
```

```
q(Z) => s(Z).

s(W) => goal.  r(W) => goal.  % two goals
```

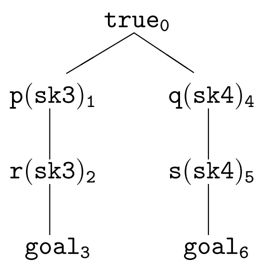

Figure 3: Example 3 tree

Notice that different Skolem constants are generate in the branches.

Example 4. Here is a simple example showing how to formulate a conjecture having coherent form (1) with **both** antecedent and consequent.

$\forall x\ (p(x) \to q(x) \vee r(x))$.

$\forall x(q(x) \to \exists u\ s(u))$.

$\forall x(r(x) \to \exists v\ s(v))$.

$\forall x\ (p(x) \to \exists w\ s(w))$? % *implication query*

The colog translation uses assumption of antecedent to effect the query:

```
p(X) => q(X) | r(X).

q(X) => r(U).

r(X) => s(V).

true => p(@X). % assumption of antecedent

s(W) => goal.
```

The *hypothetical* variable `@X` is actually a colog constant, meaning "suppose `@X` is an *arbitrary* general value such that `p(@X)`". The colog programmer can choose their own unique names for hypothetical variables; there is no obligation to use the `@` notation. Notice that the use of the hypothetical variable imposes a nonempty domain for the problem.

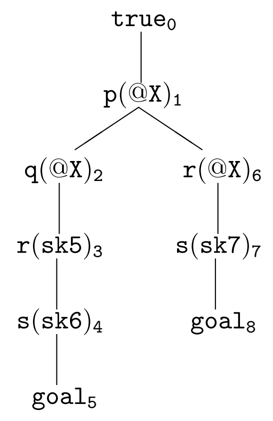

Figure 4: Example 4 tree

Fig. 4 illustrates a colog deduction tree for this problem.

The reader should construct a small example that shows that one needs unique names for different hypothetical variables.

Additional examples of translations from logical coherent formulas to colog are given below, for negation and equality.

3 Negation

Falsity axioms form (4) can disprove queries.

Example 5.

$p \to q \vee r.$

$q \to false.$

$r \to false.$

$p \to false$?

colog code:

```
p => q | r.

q => false.

r => false.

true => p.  % assumption of antecedent
```

```
false => goal.
```

Notice that the proof tree cannot actually use the last colog rule. The proof tree directly deduces that `p` must be false.

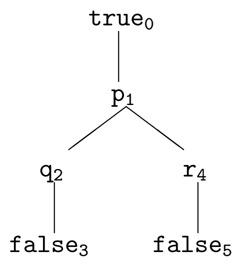

Figure 5: Example 5 tree

Example 6. Here is a rework of Example 5, this time using contraries of the predicates.

$p \to q \vee r.$

$\neg q.$

$\neg r.$

$p \to \mathit{false}$?

$(\neg p \to \mathit{goal}$?   % *alternately*)

colog code:

```
p => q | r.

true => ~q.

true => ~r.

q, ~q => false.  % consistency axiom

r, ~r => false.  % consistency axiom

true => p.  % assumption of antecedent

false => goal.  % superfluous again
```

We add *consistency* axioms for the negations. colog allows the use of tilde ~ for contraries, but the tilde is part of the predicate name and not an operator. (By way of contrast, minus - is a unary term operator, and would not usually be used for predicates.) No meaning is attached to ~p, but the programmer can write axioms to assert intended logical relationships. For example, we would have to explicitly use an axiom like

$$\neg\neg p \rightarrow p. \tag{5}$$

to say that double negation collapses (for p).

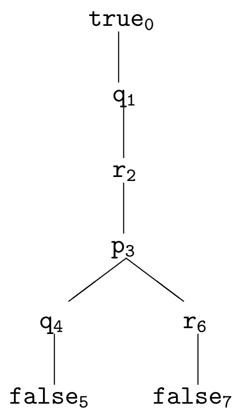

Figure 6: Example 6 tree

4 Equality

The colog programmer can use '=' for equality: the colog reader will parse '=' as an infix binary predicate. (If one uses 'eq' say for equality, that will have to be a prefix predicate.) The programmer needs to provide the relevant axioms for equality. (It is possible to automate the generation of equality axioms for colog theories.)

Example 7. Using = as equality, translate the following problem to colog.

$p(a, b)$.

$f(c) = a$.

$c = d$.

$f(f(c)) = b.$

$p(f(d), f(f(c)))$?

colog code:

```
true => p(a,b), f(c)=a, c=d, f(f(c))=b.  % rule 1

true => dom(a), dom(b), dom(c), dom(d).  % rule 2

p(f(d),f(f(c))) => goal.  % rule 3

 % equality axioms
dom(X) => X=X. % rule 4
X=Y => Y=X. % rule 5
X=Y, Y=Z => X=Z. % rule 6

% function substitutivity
X=Y => f(X)=f(Y). % rule 7
% predicate substitutivity
X=Y, U=V, p(X,U) => p(Y,V). % rule 8
```

The proof tree is linear having 36 nodes. Here is the proof extracted from the deduction (automatically, by a colog prover).

```
 LEAF 36.
@35, rule3:  p(f(d),f(f(c))) => goal
@34, rule8:  f(c)=f(d), b=f(f(c)), p(f(c),b) => p(f(d),f(f(c)))
@29, rule7:  c=d => f(c)=f(d)
@18, rule8:  a=f(c), b=b, p(a,b) => p(f(c),b)
@14, rule5:  f(f(c))=b => b=f(f(c))
@12, rule5:  f(c)=a => a=f(c)
@8, rule1:  true => p(a,b), f(c)=a, c=d, f(f(c))=b
@5, rule4:  dom(b) => b=b
@0, rule2:  true => dom(a), dom(b), dom(c), dom(d)
```

The extracted proof reads from bottom of the proof tree to its top or root. The reader should load the colog code and inspect the tree. (Notice that rule 7, being complex, is a queued colog rule.) The reader should run the colog prover on the colog code for Example 7 and inspect the proof tree.

This example was constructed so as illustrate the need to supply both the equality axioms (reflexivity, symmetry, transitivity) and to enable substitution of equals for both terms and predicates. Example 13 below shows a substitution axiom for the case of a binary infix operation in the theory.

The colog programmer would not be obligued to use '=' as the equality predicate so long as the appropriate equality axioms and substitutivity axioms are supplied. However, Example 8 is interesting in this regard: note that only reflexivity of '=' is explicitly given as an axiom.

. . .

Example 8. If 'eq' is a reflexive relation which substitutes for equality '=' then 'eq' *is* equality.

$\forall x\ eq(x,x).\quad eq\ is\ reflexive$

$\forall x \forall y \forall u \forall v (eq(x,u) \wedge eq(y,v) \wedge x = y \to u = v).\ \ \%\ eq\ substitutesfor\ =$

$\forall x\ x = x.\quad \%\ =\ is\ reflexive$

$\forall x \forall y (eq(x,y) \to x = y)$?

colog code:

```
dom(X) => eq(X,X).

eq(X,U), eq(Y,V), X=Y => U=V.

dom(X) => X=X.

true => eq(@X,@Y), dom(@X), dom(@Y).

@X=@Y => goal.
```

See Fig.7 for the deduction tree. Here is the extracted proof followed by the deduction tree . . .

```
 LEAF 10.
@9, rule5:  @X=@Y => goal
@8, rule2:  eq(@X,@X), eq(@X,@Y), @X=@X => @X=@Y
@5, rule3:  dom(@X) => @X=@X
@3, rule1:  dom(@X) => eq(@X,@X)
@0, rule4:  true => eq(@X,@Y), dom(@X), dom(@Y)
```

The next section outlines the syntax currently allowed for our colog language implementation.

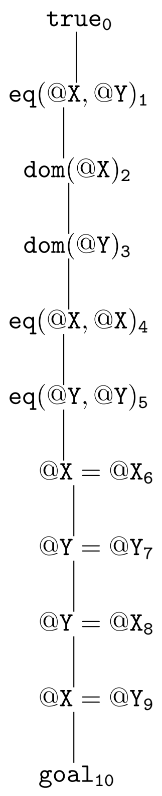

Figure 7: Example 8 tree

5 Succinct colog syntax

A colog **function name** begins with lowercase alphabetic followed by alphabetic, numeric, and underscore ('_') in any mixture. A colog **predicate name** has the same definition except that the name may be preceded by arbitrarily many keyboard tildes (~), explained below.

5.1 symbolic term syntax

- `X` variable starts with Uppercase alpha
- `c` constant starts with lowercase alpha
  `@V`, V a variable is a special constant, universal witness (examples 4, 8)
- `f(t1,t2,...)` function name starts with lowercase alpha,

`t1`, `t2`, ... are terms

- `(t1` β `t2)`, parens required, binary infix operator symbols β:

  ```
  *  #   +  /  \  -   ^   &   |   ;   ->   =>
  ```

- μ`t` unary prefix operator symbols μ:

  ```
  -   ?   !
  ```

- fixed or float numbers, e.g., 233 '21.789'

The binary infix '-' requires a space to follow, and the unary prefix '-' requires no space, e.g., as in a colog code axiom expressed like this ...

```
dom(X) => (X - -Y) = (X + Y).
```

Numbers are symbolic at present, no evaluation. Alternately, a function or constant name can be `'quoted'` or `"quoted"`, any symbols within quotes.

5.2 symbolic predicates

- `p` predicate name p starts with lowercase alpha
- `p(t1, t2, ...)` p starts with lowercase alpha (or quoted, as for functions)
  `t1`, `t2`, ... are terms
- `t1` ι `t2` *no* parens, infix predicate symbol ι:

  ```
  =  !=  <  <=  >  >=
  ```

A predicate name can begin with a keyboard tilde character ~, which is read as part of the predicate name, and not as a prefix operator. The preferred method to represent contraries in colog is to use the tilde and then include axioms regarding the intended logic. See Example 6 in Section 3. This pseudo negation is *inherently intuitionistic*, unless the colog programmer adds axioms for special consideration, e.g.,

```
p(X), ~p(X) => false.  % universal consistency rule

~p(X), ~~p(X) => false.  % a consistency rule for non-set logic
```

```
p(X) => ~~p(X). % universal property of contraries

~~p(X) => p(X). % set-logic axiom for predicate p

dom(X) => p(X) | ~p(X). % set-logic axiom for predicate p
```

Enforcing set-logic for contraries requires axioms for each predicate in the colog theory. Such axioms must be included manually by the colog programmer (or by some code wizard ...).

5.3 colog conjunction, disjunction

conjunction
`a1, a2, ...` finite sequence of atoms

disjunction
`c1 | c2 | ...` finite sequence of conjunctions

5.4 colog rule forms

`true =>` δ. *consequent* δ is a colog disjunction

α `=>` δ. *antecedent* α is a colog conjunction, δ a colog disjunction

α `=> goal.` α is a colog conjunction

α `=> false.` α is a colog conjunction

And finally, a **colog theory** is a finite sequence of colog rules.

At present the colog reader ignores comments in the code. Comments are single lines started with % or //. Multi-line comments have form /* ...*/.

6 Coherent approximations

The methods for translating coherent logic into colog exemplified in Section 2 can all be automated quite easily.

There are often ways to specify a colog problem as a translation of a first-order logic problem even when the source theory itself is not explicitly coherent. This section explores some possibilities. Check back here later for additional examples.

6.1 *included* middle

Example 9. How might we express a denial of excluded middle in colog? Specifically, suppose that p/1 is not a set predicate, and then prove that it is not.

$\forall x\ (p(x) \vee \neg p(x)) \rightarrow \perp.$

$\forall x\ (p(x) \vee \neg p(x)) \rightarrow \perp?$

The idea is to give a colog formulation of the FOL problem and to prove it with a Skolem Machine. There is minimal logical content to this problem since it asks to prove what is supposed as an axiom.
. . . possible colog code:

```
true => dom(@X), p_middle(@X). % a 'middle' instance for the
axiom

p_middle(@X), p(@X) => false.  % outside p

p_middle(@X), ~p(@X) => false.  % outside ~p

dom(X) => p(X) | ~p(X). % supposition for query, show false

false => goal.  % unusable
```

Fig.9 shows a colog proof tree for our colog problem formulation.
Our coherent representation uses a trick suggested by Heyting algebras: Represent propositions by open sets on the real line and use interior-of-complement as the logical negation of a proposition. If we can deny the middle of a proposition then there is an instance of that middle, `p_middle(@X)`. Notice that the colog constant `@X` is introduced in an axiom as an existential instance! (Reminder: One can use any new symbol instead of our `@X` colog notation.)

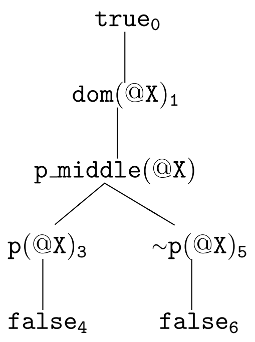

Figure 8: Example 9 tree

6.2 *Coq via* colog

The intention of the examples in this subsection is to represent a few Coq[2] problems *closely* as colog problems, by way of example.

Example 10. Consider the following *Coq problem script*:

```
Module Example10.
Inductive bool : Type :=
  | true : bool
  | false : bool.
Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.
Lemma lem1 : andb true false = false.
   Proof. reflexivity. Qed.
Lemma lem2 : andb false true = false.
   Proof. reflexivity. Qed.
Theorem andb_eq_orb :
  forall (x y : bool),
     (andb x y = orb x y) -> x = y.
  Proof.
     intros x y H.
     destruct x. destruct y.
     reflexivity.
     rewrite <- lem1. rewrite H. reflexivity.
     destruct y.
     rewrite <- lem2. rewrite H. reflexivity.
     reflexivity.
  Qed.
```

via colog ...

```
   \% Frame the Theorem for a colog proof
     true => bool(@X), bool(@Y),
             andb(@X,@Y) = orb(@X,@Y).
     @X=@Y => goal.
   \% Inductive: destruct a bool
     bool(Z) => Z=true | Z=false.
   \% consistency
     andb(A,B)=true, orb(A,B)=false => false.
     andb(A,B)=false, orb(A,B)=true => false.
   \% Definition andb
     bool(Y) => andb(true,Y)=Y.
     bool(Y) => andb(false,Y)=false.
   \% Definition orb
     bool(Y) => orb(true,Y)=true.
     bool(Y) => orb(false,Y)=Y.
   \% equality
     bool(B) => B=B.
     A=B => B=A.
     A=B, B=C => A=C.
   \% substitution of =
     A=B,C=D => orb(A,C)=orb(B,D).
     A=B,C=D => andb(A,C)=andb(B,D).
```

It is instructive to compare the resulting colog proof with the Coq proof. This is left as an exercise for the reader. A Coq proof often requires a careful choice of tactics, even though multiple helpful tactics might apply. Colog is considerably more nondeterministic: Any successfully saturated colog tree suffices for proving.

The colog theory here illustrates how reflexivity is force for all things within the scope of theory (bool). More generally, we might have more than one type in the theory. For example, if we had had bool and nat within the same theory we could have expressed reflexivity for = like this

```
\% nat type needs equality
nat(Z) => dom(Z).
\% bool type needs equality
bool(Z) => dom(Z).
\% and one reflexivity axiom
dom(Z) => Z=Z.
```

Example 11. Consider the following Coq problem involving proof by induction:

```
Module Example11.
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O => m
    | S n' => S (plus n' m)
  end.
Theorem plus_0_r : forall n:nat, plus n O = n.
Proof.
  intros n. induction n as [| n'].
  (* Case n = 0 *)  reflexivity.
  (* Case n = S n' *)
    simpl. rewrite -> IHn'. reflexivity.
Qed.
```

We want a simple colog formulation for proving the theorem by induction.

```
\% Theorem
true => nat(@N), plus(@N,0)=@N.  % induction assumption
plus(0,0)=0 => basegoal.
basegoal, plus(s(@N),0)=s(@N) => goal.  % to prove
\% nat type
true => nat(0).    % use 0 symbol
nat(N) => nat(s(N)).
nat(N) => dom(N).
\% Definition plus
nat(X), nat(Y) => nat(plus(X,Y)).
nat(Y) => plus(0,Y)=Y.
nat(X), nat(Y) => plus(s(X),Y)=s(plus(X,Y)).
\% equality
dom(Z) => Z=Z.
X=Y => Y=X.
X=Y, Y=Z => X=Z.
\% substitution
A=B, C=D => plus(A,C)=plus(B,D).
A=B => s(A)=s(B).
```

The reader should work both the Coq problem and the colog problem and compare the results. See the extracted colog proof to compare with the Coq proof.

Example 12. Here is a simple Coq problem involving polymorphic lists:

```
Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X -> list X -> list X.
Fixpoint length (X:Type) (l:list X) : nat :=
  match l with
  | nil      => 0
  | cons h t => S (length X t)
  end.
Check (cons nat 1 (cons nat 2 (nil nat))).
  (* =>
     cons nat 1 (cons nat 2 (nil nat))
          : list nat
  *)
```

The following colog approximation expresses the polymorphism using *type predicates*:

```
% Check this construction
  type(Type, cons(1,cons(2,nil))) => goal.
% polymorphic list
  type(T) => type(list(T),nil).
  type(T), type(T,X), type(list(T),L) => type(list(T),cons(X,L)).
  type(T), type(T,X) => dom(X). \% type(dom,X)
% data
  true => type(nat),
          type(nat,1),
          type(nat,2),
          type(nat,3).
```

Running the colog problem shows that the colog proof binds `Type` in the colog goal rule to `list(nat)` (@13). We formulated the colog problem to emulate the `Check` problem for Coq.

```
> java -jar colog13B.jar file=x12.co log=true

colog: file=x12.co, d=300, w=1000, c=-1, PROOF,
#inferences=11, #facts=14,   #branches=1, time=0ms

LEAF 14.
@13, rule1: type(list(nat),cons(1,cons(2,nil))) => goal
@12, rule3: type(nat), type(nat,1), type(list(nat),cons(2,nil))
   => type(list(nat),cons(1,cons(2,nil)))
@9, rule3: type(nat), type(nat,2), type(list(nat),nil) =>
    type(list(nat),cons(2,nil))
@7, rule2: type(nat) => type(list(nat),nil)
@0, rule5: true =>
    type(nat), type(nat,1), type(nat,2), type(nat,3)
```

The reader is encouraged to formulate more examples of Coq problems with colog approximations. Coq provides excellent motivation for coherent logic formulations, and possibly there are *coherent* Coq tactics that could be designed. The examples 10, 11 and 12 suggest that coherent logic might concern itself with datatypes in a systematic manner.

6.3 *Abstract algebra via* colog

Algebra problems are excellent test cases for problem formulation and for studying effective search heuristics for provers. The primary obstacle is the management of *term complexity.* (This example illustrates colog's `*` symbolic operator used as the operation for our monoid.)

Example 13. Here is the problem: Show that left and right inverses in a monoid are equal.

```
% data
  true => dom(e),
          dom(x), dom(y), dom(z), % hypothesis
          (y*x)=e,  % left inverse for x
          (x*z)=e.  % right inverse for x
% conjecture
  y=z => goal.
% closure for *
  dom(X), dom(Y) => dom((X*Y)).
% associativity of *
  dom(X), dom(Y), dom(Z) => ((X*Y)*Z)=(X*(Y*Z)).
% e is for *
  dom(X) => (X*e)=X, (e*X)=X.
% = axioms
  dom(X) => X=X.
  X=Y => Y=X.
  X=Y, Y=Z => X=Z.
% substitutivity for =
  A=B, C=D => (A*C)=(B*D).
```

The primary colog search heuristic is complexity, which is illustrated with the following command-line execution of colog on this input theory (`x13.co`).

```
> java -jar colog13B.jar file=x13.co depth=6500 width=1
  complexity=2 log=true

colog: file=x13.co, d=6500, w=1, c=2, PROOF, #inferences=1384,
#facts=1407, #branches=1, time=434ms

LEAF 1407.
@1406, rule1: y=z => goal
@1405, rule7: z=y => y=z
@1404, rule8: z=((y*x)*z), ((y*x)*z)=y => z=y
@1341, rule7: y=((y*x)*z) => ((y*x)*z)=y
@1340, rule8: y=(y*(x*z)), (y*(x*z))=((y*x)*z) => y=((y*x)*z)
@1319, rule7: ((y*x)*z)=(y*(x*z)) => (y*(x*z))=((y*x)*z)
@1318, rule4: dom(y), dom(x), dom(z) => ((y*x)*z)=(y*(x*z))
@650, rule8: y=(y*e), (y*e)=(y*(x*z)) => y=(y*(x*z))
@649, rule7: (y*(x*z))=(y*e) => (y*e)=(y*(x*z))
@648, rule9: y=y, (x*z)=e => (y*(x*z))=(y*e)
```

```
@93, rule8: z=(e*z), (e*z)=((y*x)*z) => z=((y*x)*z)
@92, rule7: ((y*x)*z)=(e*z) => (e*z)=((y*x)*z)
@91, rule9: (y*x)=e, z=z => ((y*x)*z)=(e*z)
@87, rule7: (e*z)=z => z=(e*z)
@84, rule5: dom(z) => (z*e)=z, (e*z)=z
@66, rule7: (y*e)=y => y=(y*e)
@64, rule5: dom(y) => (y*e)=y, (e*y)=y
@9, rule6: dom(z) => z=z
@8, rule6: dom(y) => y=y
@0, rule2: true => dom(e), dom(x), dom(y), dom(z), (y*x)=e, (x*z)=e
```

Without the complexity switch (or cut on the GUI toolbar) the search would include many other facts (unnecessary for proof). The reader can load the theory into the GUI and run the proof search. Which rules are not used? Try using complexity=3, 4, . . .

The general problem of managing operator/function complexity remains an issue of primary concern for colog design.

6.4 *Function extensionality*

Example 14. The colog functions `f` and `g` have equal definitions, assuming commutivity for `+`.

```
f=g => goal.

nat(X) => f(X)=(X+1).
nat(X) => g(X)=(1+X).

nat(X), nat(Y) => (X+Y)=(Y+X).
true => nat (1).

f(@Nat)=g(@Nat) => f=g.  % if arbitrarily = for values then =
true => nat(@Nat).

nat(Z) => dom(Z).

% equality axioms
dom(X) => X=X.
X=Y => Y=X.
X=Y, Y=Z => X=Z.
A=B, C=D => (A+C)=(B+D). % substitutivity for +
```

The theory proves easily using a colog prover (load and run).

6.5 *Russell's paradox redux*

Russell's class of all classes that do not belong to themselves

$$R = \{w | w \notin w\} \tag{6}$$

has as a classical logic consequence the following paradoxical formula.

$$R \in R \leftrightarrow R \notin R \tag{7}$$

Let us look at a colog rendition, where `dom` refers to the class domain for this example
Example 15.

```
true => dom(r),                % Russell's class R
        dom(A), ~in(A,A).    % some class not belonging to self

dom(X), ~in(X,X) => in(X,r).   %  define R (6) coherently
dom(X), in(X,r) => ~in(X,X).

dom(X) => ~in(X,X) | in(X,X).  % excluding middle for in
                               % comment out to compute
                               % intuitionistic model

~in(X,X), in(X,X) => false.    % consistency re in
```

The coherent refutation for this theory is shown in Fig.9

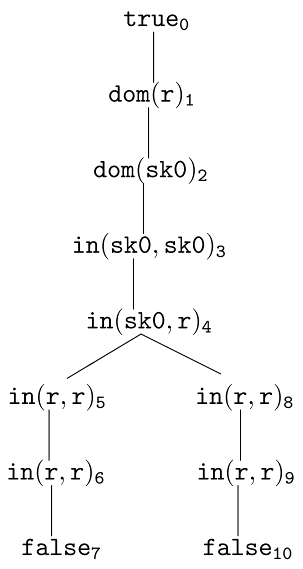

Figure 9: Example 15 refutation

However, if we *delete the excluded middle rule* we can compute the coherent tree model for the pared theory, as shown in Fig. 10.

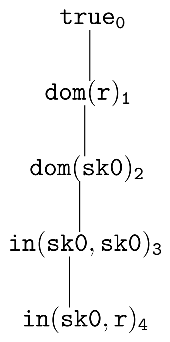

Figure 10: model without excluded middle rule

We can coherently prove the paradoxical equivalence (7) without the excluded-middle rule (assume either implicand, derive other one as goal), but we cannot produce the explicit contradiction in the intuitionistic version.

References

[1] *SkolemMachines website:* `SkolemMachines.Org`

[2] The Coq Proof Assistant website `http://coq.inria.fr/`

[3] John Fisher and Marc Bezem, Skolem Machines, *Fundamenta Informaticae*, 91 (1) 2009, pp.79-103.
A version of this paper with minor corrections is linked at the SkolemMachines website: `SkolemMachines.pdf`.
Note that *colog* is referred to as *geolog* in this paper.