#1
EXAMPLES OF
AUTOLOG METALOGIC
2/6/2019

§1 Heyting term algebra, modulators and rules

All of the examples in this lecture are intended to illustrate a metalogic based upon Heyting algebraic logic terms embedded into autolog code. It is possible to employ other algebraic logics in a similar manner (a topic for subsequent lecture notes).

For quick reference to background information regarding Heyting algebra, see

https://en.wikipedia.org/wiki/Heyting_algebra

In this lecture note, we will use the following symbolic notation for the intended Heyting operators:

meet	∧
join	∨
less or equal	≤
residual	→
negation	¬
least/bottom	⊥
greatest/top	⊤

These notes were written using the autolog editor, in order to capture symbolic notations faithfully.

Most of the examples describe how one can write codes that express some intended metalogic either as term algebra or a literals in autolog. Small programs are als given so as to illustate autolog computations.

There are two styles for coding algebraic logic axioms. One style employs coherent form rules and the other employs equality modulators (implemented as rewrite).

As a first example, let us carefully consider the difference between the autolog modulator

```
     A∧B=B∧A.      // term modulator
```

and the autolog rule

```
     A∧B => B∧A.  // inference rule
```

Both express similar intentions but result is different behaviors for the inference engine. Both forms are "autolog inferences". To illustrate the difference suppose that we have only the one fact

```
     true => (a∧b)vc.
```

and the start of an inference tree

```
               true
                 |
              (a∧b)vc
                 |
                 ?
```

If the term modulator is part of our theory then we also have the tree term inference

```
               true
                 |
              (a∧b)vc
                 |
              (b∧a)vc
```

abtained by matching the left side of the modulator in the tree term (a∧b)vc, substituting a∧b by b∧a and asserting the tree term (b∧a)vc. The modulator can act at the term level, the rule can only infer at literal level.

On the other hand, the rule would not apply to terms inside a tree term, so the last inference is not possible possible. Obviously, modulators are more active in such situations. If the term modulator is included in our program, then the inference rule is redundant.

A modulator equality

```
     L=R.
```

substitutes instances of L (the left Lterm) in a fact tree term by the corresponding instance of R (the right Rterm), and not vice versa.

A match regimen for a modulator can be illustrated using a simple example. Suppose that our theory is

```
   true => a=a+c.
   a=b.           // M (rewrite a as b)
```

Here is a simulation of branch action for this theory

```
              true
               |
              a=a+c    0
           ----|----
              b=a+c    1
               |
              a=b+c    1
           ----|-----            |
              b=b+c    2
               |x
              b=b+c    2  ignored, not new
```

In this simulation, modulator M is used on each of the individual instances of a in the fact at stage 0, producing modulants at stage 1. In stage 2, M modulates remaining instances of a in the proof terms of stage 1. Any repeated fact is ignored (not actually added to the branch). An incremental saturating modulation regimen suc
as this would, over successive stages, assert a complete branch set of modulants.

Generally speaking, both modulators and definite rules use a level-saturation methodology (described more fully in the Autolog design document).

At present, the only way to index modulators is via the Lterm (operators or constants). Thus, for example, if we wan t to include the modulator

```
  AvB=BvA.
```

it should be the case the modulator makes sense for

ANY instance AvB that might occur on a proof tree branch. This means that variables A and B are "virtually indexed" by occurring in the Aterm of the modulator, rather than some "esplicit index" such as A:T and B:T for some T. We leave this as an open design issue at this time.

---

§2 Folding and unfolding metalogic rules

An "unfolding" rule replaces a term algebra expression by literals. Unfolding rules for ∧ and v could be

```
      ⊥ => false.
     A∧B => A, B.
     AvB => A | B.
```

A "folding" rule replaces literal terms by an appropriate term algebra expression. Unfolding rules for ∧ and v could be

```
     true => ⊤.
     A, B => A∧B.
     A, B:prop => AvB.
```

Note that a literal like 'B:prop' (for example) is needed to fix a referent/index for B in the consequent of the second rule. A rule like the following would not be what is really intended

```
     A => AvB.
```

because an application of this rule would result in a Skolem constant replacing 'B' in the consequent, which would entail "some" B rather than "any" B. However, the literal term 'B:prop' employed above is just an example, and in practice some other indexing for B might be appropriate.

{The folding and unfolding rules given here as examples. Other versions are possible, and sometimes necessary, as in the case of the typing above. Sometimes it is wise to type all term variables as an indexing ploy, to limit the reference of term variables and for efficiency of autolog computations.}

For →, we might have

$$A\to B,\ A \Rightarrow B.$$

and/or, as a modulator,

$$A\to B \wedge A = B.$$

For ¬, we have

$$\neg A,\ A \Rightarrow \text{false}.$$

Notice that this rule is preferable to

$$A,\ \neg A \Rightarrow \text{false}.$$

It is usual for autolog to satisfy antecedent literal terms from left-to-right. ¬A is indexed via its operator ¬, and a literal match (using an index table) provides a value for A, which either does or does not match the second literal of the preferred form of the rule. In the alternate rule, much more work may be required to first achieve a match for A and then locate ¬A.

Alternatively, we might employ modulators

```
     ¬A ∧ A = ⊥.
     ⊥ = false.
```

For these examples, the choice of rule/modulator would be a programming style decision. Additional axioms could be

```
     P => ¬¬P.  // better:
        P:prop, P => ¬¬P.
     A∧¬B => ¬(A→B).
     ¬(A→B) => ¬¬A ∧ B.
```

The following modulator might be specified for some autolog programs, but would be considered as unsound as a substitution rule for P's not in an heyting algebra logic context.

```
       P=¬¬P.     // NO
```

To illustrate the style of reasoning with metalogic terms, consider the following problem.

EXAMPLE 1.

```
     // rule has types, impredicativity, indexicality
     P:T→prop, Q:T→prop, X:T, P(X) => P(X)vQ(X).  // #1
     //  coherent unfolding for v
     PvQ => P|Q.                                  // #2
     true => p:int→prop, q:int→prop, a:int.    // data
     p(a) => goal.
     q(a) => false.
```

which has a proof tree

```
                true
                 |-   data
             p:int→prop
                 |
             q:int→prop
                 |
               a:int
                 |-   #1
            p(a) v q(a)
             /  #2   \
          p(a)       q(a)
            |         |
           goal     false
```

----

§3 "ex falso (sequitur) quodlibet"

"From contradiction every statement follows" --
the principle of explosion -- but not for autolog
inference generally. To specify EFQ explicitly for the
metalogic under consideration, a rule like

```
     ⊥ => P.
```

is NOT going to work since this only says, in effect, that
some literal P can be inferred from bottom, not that all
can be inferred. (That P is not indexical in the rule is
also a problem).

We might attempt to program EFQ as follows

```
     ⊥, P:prop => P.
```

via a qualifying type judgement in the antecedent of the rule. This approach also indexes P. To enforce EFQ, rules would need to index all literals and include a similar inference for each indexed literal possible. One might only index certain chosen literals (smaller explosion). For example, consider this problem.

EXAMPLE 2.

```
   true => ⊥, 1:int,
           p(2),
           q:int→prop .
   ⊥, P:int→prop: X:int => P(X).    // Explode these.
   q(A) => goal.       // via explosion specification
```

Consider the following proof tree

```
          true
            |
            ⊥
            |
          1:int
            |
        q:int→prop
            |
           q(1)
            |
           goal
```

Show that

```
     p(1) => goal.
```

cannot be proved, because 'p' does not fit the explosion profile. The example (and this section) is intended to illustrate EFQ re autolog, nor to promote imposing EFQ, nor to promote avoiding EFQ. Either promotion would require more motivations.

Another well known form of argument for explosion works by employing a rule like

```
     P => PvQ.
```

where one gets to impose that Q is "anything". However, the coherent form rule only says PvQ for some Q (not all). In order to get irrelevant derivations, we could try something like the following, deriving that unicorns exist. But again, not everything follows automatically.

EXAMPLE 3.

```
     true => p, ¬p.
     true => unicorns_exist:relevent.
     P, Q:relevent => PvQ.  // need to index Q.
     ¬P ∧ (PvQ) => Q.
     unicorns_exist => goal.

                 true
                  |
                  p
                  |
                 ¬p
                  |
          unicorns:relevent
                  |
              p v unicorns_exist
```

```
         |
   unicorns_exist
         |
        goal
```

I left out the rule

```
        ¬P,P => false.
```

(which is indexical) because false in a derivation frame would have stopped the expansion of the branch, and that also would have stopped derivation of the irrelevant.

----

§4 Tree frames for metalogic

An intuitionistic Kripke-like frame can be defined for the tree structures computed for a metalogic program by a Skolem machine.

Suppose that our autolog theory includes the following axioms/rules:

```
     A∧B => A, B.
     A∨B => A | B.
     A→B, A => B.
     ⊥ => false.
     A→⊥ => ¬A.  // or  modulator
                 //   {A→⊥ = ¬A.}
```

DEFINITION: A "branch (or frame) set" is a set of branch terms all contained in one contiguous Skolem tree branch. A branch frame "forces" a term provided that term can be deduced on a tail for any branch containing the branch se

The following statements 1-4 are consequences of this definition of forces, and resemble the Kripke conditions for intuitionistic logic frames:

1- If P is a term, branch set W forces P, and $W \leq U$ (subset) for branch set U, then U forces P. (monotonicity)

2- If $A \wedge B$ is term and branch set W forces $A \wedge B$, then W forces A and W forces B.

3- If $A \vee B$ is a term and branch set W forces $A \wedge B$, then W forces A or W forces B.

4- If $A \rightarrow B$ is a term, branch set W forces A, then W forces B.

DEFINITION: If no branch set $U \geq W$ forces A, then W "supports" $\neg A$.

Notice that in EXAMPLE 1, the initial branch set (up to the application of rule #1) forces $p(a) \vee q(a)$, but that no lower specific branch term is forced.

EXERCISE 1. Formulate a reasonable definition of the notion of a term algebra corresponding to an autolog problem. (Hint: "Herbrand universe".) Give an example where neither P nor ¬P is forced, so ¬P is supported, for a term P in the relevant term algebra of the problem.

EXERCISE 2. Express the rules (1-4) as autolog rules using a predicate forces(W,P).