#3
EXAMPLES OF AUTOLOG TYPE METALOGIC

8/13/2022 (corrected)

§1 Type terms and term types

Consider the following autolog metalogic inference rule:

```
T:type => list(T):type.
```

Both of the literal expressions of the rule are a "typing term", 'T:type' expressing a judgement of type and the term 'list(T)' is a "term type", an algebraic term expressing a type. These patterns continue throughout this lecture note.

The grammar for autolog is described in Chapter 3 of
[1] https://
skolemmachines.org/autolog/docs/AutoLog_Design.pdf

Almost every autolog grammatical category is a "term": literals, functions, operators, judgements, ...

For the examples given in the following sections I mostly use notations very similar to those used in the following articles
[2] https://
en.wikipedia.org/wiki/Intuitionistic_type_theory
[3] https://
plato.stanford.edu/entries/type-theory-intuitionistic/

Ref[3] has a very nice table displaying dual symbolic operators for intuitionistic logic and intuitionistic type theory. Here is a version of that table:

logic	type	
⊥	∅	
⊤	1	
A∨B	A+B	
A∧B	A×B	
A⊃B	A→B	
∃x:A.B	Σx:A.B	
∀x:A.B	Πx:A.B	
=	= :	(added)

In Ref[3,§2.4] the author writes an '=' between the logic entries and the corresponding type entry. This might be somewhat misleading in this lecture however. I will attempt to only use the logic operators in contexts for logic and the type operators in contexts for types, especially in typing contexts (typing term)

Lterm : Tterm

where Lterm is a logic term and Tterm is a type term. The examples in the following sections will employ visually similar symbolism, respecting the logic/type distinction just made. (*However, I use → instead of ⊃ for logic terms.) The primary theme for this lecture will be the autolog interplay between term logic and term type computed by autolog inference programming. The autolog language itself is still under development, so the examples serve as a motivation of the language design changes and refinements.

---

§2 ⊥ ∅ , ⊤ 1, 2

See also Lecture Notes #1.

The terms P→⊥, P:∅ and ¬P might be characterized using several rules or modulators, including the following:

rule	modulator
P→⊥ => P:∅.	P→⊥ = P:∅.
P:∅ => P→⊥.	P:∅ = P→⊥.
P→⊥ => ¬P	P→⊥ = ¬P.
¬P => P→⊥.	¬P = P→⊥.
P:0 => ¬P.	P:∅ = ¬P.
¬P => P:∅.	¬P = P:∅.
A:T, A→⊥ => T=∅.	

These are ⊥-logic, ∅-type dual correspondences.

The terms ⊤→P and P:1 might be characterized as follows:

rule	modulator
⊤→P => P:1.	⊤→P = P:1.
P:1 => ⊤→P.	P:1 = ⊤→P.
⊤→P => P.	⊤→P => P.
P => ⊤→P.	P = ⊤→P.

The 2 type, intended to express two distinct outcomes, might be employed to express a restricted law of excluded middle (LEM), as follows:

```
    rule                  modulator
  P:2 => P | ¬P.        P:2 = Pv¬P.
```

Notice that the modulator cannot use P|¬P as a term because '|' is not a term operator for autolog. '|' is only a separator for rule consequents. However, as in lecture #1 one could also employ the rule

```
PvQ => P | Q.
```

to unfold Pv¬P into consequent P|¬P.

EXERCISE C. Write a small autolog program/theory where some predicates require or "enjoy" LEM and some do not. Give an answer (a) where some of the rules/modulators of this section are employed, and an answer (b) where none of the rules or modulators above are employed.

The rules/modulators in this § might lead to unintended inferences for some theories when used for term contexts other than rule literals.

---

§3 ∧ x

This section illustrates some examples involving the autolog inferential interplay between ∧ logic and x type.

EXAMPLE 1. The following specification defines a pair.

```
S:type, T:type,
   X:S, Y:T => pair(X,Y):S×T.              //1
```

In a rule like this we might refer to pair(X,Y) as the "constructor" term for the "type" term T×S, and the rule describes this "construction". The '×' is a defined operator, whose context defines it as an infix autolog operator (functor) of arity 2. Autolog will display the internal form as T×S, without quotes. (The × operator can be selected from the code menu of the autolog editor, "cross product".)

A "deconstructor"rule, might be formulated conversely

```
pair(X,Y):SxT => S:type, T:type, X:S, Y:T.
```

or, employing logic ∧ dual to x

```
pair(X,Y):SxT => (S:type) ∧ (T:type), X:S ∧ Y:T.
```

As in Lecture #1, we could also employ the rule

```
P∧Q => P, Q.
```

to unfold embedded logic terms into autolog rule literals.

The constructor and deconstructor rules form a definition for the type. Autolog does not reserve the names 'type' nor 'pair' for any fixed usage other than what the programmmar specifies by the rules.

Consider this problem (using //1)

```
true => int:type, float:type,
            2:int, 2.0:float.        //data
pair(2,2.0):int x float => goal.     //goal rule
```

An easy autolog verification tree for goal is

```
                true
                  |                      //data ...
               int:type
                  |
               float:type
                  |
                2:int
                  |
               2.0:float
                  |                      //1
      pair(2,2.0):int x float
                  |                      //goal rule
                 goal
```

Using terminology from the literature we could say that this derivation does establish that pair(2,2.0) "inhabits" the type int x float. We will reconsider versions of "inhabiting" in the sequel

{We could just as well have used notation pair(S,T) for the type rather than SxT, but I wanted to illustrate the ∧,x duality.}

For a quick references regarding dependent types, see [4]
   https://en.wikipedia.org/wiki/Dependent_type

For a dependent pair type consider the following

EXAMPLE 2. The following specification defines an ordered pair for a relation R on type T.

```
     T:type, R:T→T→T, X:T, Y:T, R(X,Y)      //2
               => ord_pair(R,X,Y):ord_pair(R,T,T).
```

EXERCISE A. Invent an autolog program deploying //2 that supports the following goal.

```
   ord_pair(lesseq,1,3):ord_pair(lesseq,int,int) => goal.
```

The next example illustrates some possible inference relationships between ∃ logic expressions and SxT types. Lecture #2 has other examples for ∃ logic.

EXAMPLE 3. This example employs some ∃-logic for deriving witness for the pair of EXAMPLE 1.

```
   true => int;type, ∃(X:int).  // a- there is an integer
   ∃(X:S) => (X:S).             // b- autolog unfold
   S:type, T:type,              // c- pair constructor
      X:S, Y:T => pair(X,Y):SxT.
   A:int x int => goal.         // d- A = answer
```

The following autolog proof supplies an inhabitant (answer) for goal question.

```
                true
                  |         a- generate Skolem constant
               ∃(a:int)
                  |         b- unfold ∃
                a:int
                  |         c- pair constructor rule
         pair(a,a):int x int
                  |         d- Q=pair(a,a) is answer
                 goal
```

As an aside, in autolog one can define a pair (X,Y) using intput ''(X,Y). (The functor name is '' which does not show in toString() display). So, an alternate formulation for a constructor/deconstructor pair might look something like this (displayed without quotes ''):

```
S:type, T:type, X:S, Y:T => (X,Y):SxT.
(X,Y):SxT => (S:type) ∧ T:type, X:S ∧ Y:T.
```

and a judgement modulator equation might look like this

```
(X,Y):SxT = (X:S) ∧ (Y:T).
```

EXERCISE D. Formulate an autolog lemma which justifies the x type modulator equation (RxS)xT = Rx(SxT). (Hint, see EXAMPLE 4 below, re type operator +.)

---

§4 v +

A simple specification of type + logic would be the modulator equation

```
X:S+T = (X:S)v(X:T).
```

Autolog constructor rules might look like this

```
S:type, T:type => S+T:type.      // -a
S:type, T:type, X:S => X:S+T.    // -b
S:type, T:type, X:T => X:S+T.    // -c
```

and deconstructor

```
     X:S+T => X:S | X:T.              // -d
```

S+T represents "type cases" or "alternatives".

EXAMPLE 4. The following problem explores + type logic for three types R, S, T. Assume that rules a, b, c and d are included, in addition to

```
      true => R:type, S:type, T:type,
              z:(R+S)+T.
      z:R+(S+T) => goal.
```

The following is a proof tree:

```
                  true
                   |
         R:type - S:type - T:type
                   |
               z:(R+S)+T
               /   d    \
          z:(R+S)        z:T
         /  -d-  \        |     -a
       z:R       z:S     S+T:type
        |    a    |       |     -c
     S+T:type   S+T:type   z:S+T
        |    b    |  -c    |
     z:R+(S+T) z:R+(S+T) z:R+(S+T)
        |         |         |
       goal      goal      goal
```

This problem is a kind of "lemma". It shows that the autolog rule

```
              X:(R+S)+T => X:R+(S+T).
```

is a derived rule that can be included along with rules a, b, c, and d.

EXERCISE E. Prove a lemma showing that

$$X:R+(S+T) \Rightarrow X:(R+S)+T.$$

is a derived rule.

Thus, the modulator

$$X:R+(S+T) = X:(R+S)+T.$$

could be used for judgements. Also, the modulator

$$R+(S+T) = (R+S)+T.$$

might be employed for type term. (The issue of context is being ignored for now.)

---

§5 ∀ Π

Using Π as a dependent type constructor is illustrated first. Consider functions of a type t such that

```
F:int->int x int, X:int => Y:int, F(X)=(X,Y).
```

Such functions F(X)=(X,?) use the input argument X to assign some pair (X,?) as output. The particular function g(X)=(X,4), all X, is an instance of such a function.

EXAMPLE 5. The $\Pi(N:int)(N \times int)$ can represent the dependent type t, characterized via ∀ using the rule

$$F:int \to int \times int,\ \forall(X:int)(\exists(Y:int)(F(X)=(X,Y))) \Rightarrow F:\Pi(N:int)(N \times int).$$

EXERCISE F. Employing the autolog rule just above, write an autolog rule specifying the function g(X)=(X,4) mentioned before example 5, and then prove the goal

$$g:\Pi(N:int)(N \times int) \Rightarrow goal.$$

Notice that ∀ logic is employed to establish a witness for this kind of dependent function Π type.

Another Π type representation is related to S→T types, and which has a succinct modulator formulation:

$$S \to T = \Pi(X:S)(X:T).$$

The modulator applies to type term contexts only. Expressed as an autolog rule, we would have something like

$$S:type,\ T:type,\ \forall(X:S)(X:T) \Rightarrow \forall(X:S)(X:T):S \to T.$$

Generally, derived ∀ logic expressions serve as witness evidence to related Π types. An example of such a withness judgement expression would be

$$\forall(X:T)(P(X)) : \Pi(X:T)(P(X))$$

so long as context for a rule gives indexicality to the terms, such as T:type ∧ P:T→prop (indicating that P(X) would be used as a literal in a rule).

This approach uses a logic expression in a judgement for a related type expression L:T to indicate that there is some relevant derivation of L to bear witness to the type T.

It should be stressed that autolog (at present) does not confine its ∀∃ΠΣ expressions as to argument profile. For example,

∀(X)((X:S) '⊃' (X:T)) = ∀(X:S)(X:T).

parses and looks lite it expresses the intentions used earlier in the notes. (Each of ∀∃ΠΣ is only restricted to be a name for a functor expression.) But such a modulator would most likely require other rules of modulators to be explicitly provided in order to carry out the programmer's theory design.

----

§6 ∃ Σ

The following example illustrates an autolog rule pattern where a (possibly non-discrete)∃-logic term inhabits a Σ-type term.

EXAMPLE 5. Explore unfoldings for the the first rule, which illustrates a kind of "∃:Σ" duality:

```
    T:type, P:T→prop, ∃(X:T)(P(X))
          => ∃(X:T)(P(X)) : Σ(X:T)(P(X)).       //- a
```

```
true => t:type, 0:t, s:t→t, e:t→prop, e(0).  //- b
X:t, e(X) => e(s(s(X))).                      //- c
X:T, E:T→2, E(X) => ∃(X:T)(E(X)).             //- d
∃(X:t)(e(X)) : Σ(X:t)(e(X)) => goal.          //- e
```

Notice that in the //-a rule the leading literal P:T→prop, if satisfied first, would supply values for P and T, so that the next literal ∃(X:T)(P(X)) would be predicative, and also the consequent judgement. This is a common pattern mentioned in previous lecture notes. (A judgement satisfied can predicate following expressions in a rule.)

EXERCISE F.

(i) Construct a proof tree for Example 4.
(Hint see Lecture notes #2)

(ii) What "witnesses/inhabits" the type in the goal?

(iii) Can one use autolog to prove that the inhabited type in the goal is not finite?
(Since autolog is not yet fully specified, attempt to guess at a reasonable answer.)

The next example ilustrates how a theory with v-case habitation is derivation-inhabited, according to autolog.

EXAMPLE 6. This example ilustrates how a Σ-type with v-case witnesses is derivation-inhabited, according to autolog:

```
T:type, P:T→prop, ∃(X:T)(P(X))
       => ∃(X:T)(P(X)) : Σ(X:T)(P(X)).      //- a
true => r:type, p:r→prop,                   //- b
         a:r, b:r, p(a)vp(b).
PvQ => P | Q.                               //- c
```

```
    X:T, E:T→prop, E(X) => ∃(X:T)(E(X)).        //- d
    ∃(X:t)(e(X)) : Σ(X:t)(e(X)) => goal.        //- e
```

and here is a derivation of the goal ...

```
                     true
                       |                         -b
                 r:type  p:r→prop
                       |                         ..
                    a:r  b:r
                       |                         ..
                   p(a)vp(b)
                    /     \                      -c
                 p(a)     p(b)
                  |          |                   -d
           ∃(a:e)(p(a))   ∃(b:r)(p(b))
                 |              |                -a
     ∃(a:e)(p(a)):∃(a:e)(p(a))  |
                 |     ∃(b:e)(p(b)):∃(b:e)(p(b))
                 |              |                -e
                goal           goal
```

Rule c in this example does not type it's variables. One might say that the typing is "implicit" via the v. At the current time, no autolog process "checks" that kind of typing.

---

§7 Inductive types

The example given here is a reworking of Example 12(p.18) of

[4]https://www.SkolemMachines.ORG/reports/colog/colog.pdf

which was formulated for coherent logic with functions. The link has a discussion relating the colog formulation to a coq problem problem specification. This was one of problems motivating the design for autolog as an extension of colog.

EXAMPLE 7. This problem specifies an inductive list type and an inductive length function whose specs are intertwined.

```
length(cons(s(0),cons(s(s(0)),nil)))=s(s(0)):nat
                => goal.  // show length([1,2])=2

T:type => nil:list(T).
T:type, X:T, L:list(T) => cons(X,L):list(T).

true => nat:type.
X:nat => s(X):nat.  // "X+1"

true => length(nil)=0:nat.
L:list(T), X:T =>
          length(cons(X,L))=s(length(L)):nat.

X:T => X=X:T.
X=Y:nat => Y=X:nat.
X=Y:nat, Y=Z:nat => X=Z:nat.

X=Y:nat => s(X)=s(Y):nat.
X=Y:nat => length(X)=length(Y):nat.
```

This is a notational reformulation of a problem (x12alt.co) computed by colog14I.

See §8 below for more re A=B:T notation.

---

§8 = , =:

A judgement like 5=2+3:nat is supposed to convey at least the term typing information specified by the following modulator:

$$(X:T)\wedge(Y:T)\wedge(X=Y) = (X=Y):T.$$

It is unlikely that a modulator regimen (as envisioned for autolog) would support unique term reduction derived via equality reasoning as practiced in algebraic logic.

For example, 5=2+3 might recursively derive via

```
                  s(s((s({s(s(0))}))))
                    --3--   --2--
```

or

```
                  s((s({s(s(s(0)))})))
                    --2--   --3--
```

using technically distinct derivations. The result of the reductions are identical, but the derivation steps in the reductions are not identical.

Another design issue affecting autolog is that implicitly typed equality modulations are convenient but may not impose strict type indexing, as illustrated by the following example.

EXAMPLE 8. The following problem uses no explicit type judgements for its logic terms.

```
true => ¬(p(a)vq(a)).    //- a
¬(AvB)=¬A∧¬B.            //- b
P∧Q => P,Q.              //- c
¬q(Z) => goal.           //- d
```

A simple derivation tree looks like this

```
                    true
                      |             -a
                ¬(p(a)vq(a))
                      |             -b
                 ¬p(a)∧¬q(a)
                      |             -c
                    ¬p(a)
                      |             -c
                    ¬q(a)
                      |             -d
                     goal (Z=a)
```

The modulator at b applies implicitly to the derivation term implicitly (the match of the Aterm ¬(AvB) of the modulator.) This sort of typeing by context is convenient. There is at present no explicit type check mechanism for autolog that allows this implicit typeing.

Exercise G. Suggest a typing regimen where type judgment controls the index-applicability for example G.

---

§9 Universes

Autolog, at present, imposes no restrictions on type universes. I have used a lax "--:type" notation to create indexical terms for type names. Other type names chosen for the examples in this note were chosen for intuitive appeal.

However, much more elaborate notations could be employed to name and specify universes according to principles of indexicality.

At present, I prefer to allow universe programming specs that might lead to contradictory and/or paradoxical theories; such things are also interesting.

More on this later ...

----

§10 Algebraic logic type algebras ... ?

There are prospects for extending the "logic:type" program design formalisms in this lecture note. There are at least two possibilities, each of which has been employed in this note: One is the coherent-form rule mechanisms, others are the rather tighter mechanisms using modulator equations for logic terms, type terms, and judgement terms. Various examples have been given in the previous sections of this note.

In the near future, this section or another lecture note will be devoted to the issue of "algebraic logic type algebras" in a more systematic manner. Does this suggest "symmetry" or "duality"?

~~~~~~
~~~~~~

§11 propositions as types (hott)

Here is a definition from the preprint SB
https://github.com/UniMath/SymmetryBook

"Let P be a type. The property that P has at most one element may be expressed by saying that any two elements are equal. Hence it is encoded by Π(a,b:P, a=b). We shall call such a type a proposition, and its elements will be called proofs."

Here is one partial autolog formulation of the defining condition for a proposition:

«
```
P:prop => P:type.
P:prop, A:P, B:P => A=B.
```
»

to which we add some extension:

«
```
true => rs:prop.
P:prop, X:P => P.  // W
true => p.
true => q.
p => r:rs.
q => s:rs.
rs => goal.
```
»

Now there are TWO DISTINCT autolog proofs of the goal

```
true          true
  |             |
rs:prop       rs:prop
  |             |
  p             q
  |             |
 r:rs          s:rs
  |- 1          |- 2
  rs            rs
  |            |
 goal          goal
```

The first (1) is uses r:rs as a witness for rs in W, and the second (2) is using s:rs as a withness for rs in W.

So NOT all autolog proofs of proposition rs are identical, one using r (from p) and the other using s (from q) as a witness.

This approach is like using either r or s as an indexing for the same proposition rs. The witnesses r an s play equivalent roles in different proofs, and r=s is also deducible. This also shows how autolog indexing can be employed using a type statement X:T, and that either X or T might be the indexing parameter.

So, in an autolog machine proof a witnessed fact W:P (bound) would not literally mean W is bound to an autolog proof. Thus "...will be called proofs" is a metaphor used in an otherwise formal mathematical context. (Coq may also reveal different verifications, depending on loaded library.)

Here is another definition from SB:

"Let X be a type. If for any x:X and any y:X the identity type x=y is a proposition, then we shall say that X is a set. The reason for doing so is that the most relevant thing about a set is which elements it has; distinct identifications of equal elements are not relevant."

This definition may be awkward to unwind as an autolog program, so I'll leave it alone for now ...