AUTOLOG DESIGN NOTEBOOK

MACHINA EX LOGICA

Skolem machine experiments with indexical inference

© John R. Fisher

fisher.r.john@gmail.com

Last update: November 22, 2023

Table of Contents

Table of Contents 2

Foreword ⁕ 3

1 Coherent logic extended, impredicativity ⁕ 5

2 Impredicativity Management via Indexicality ⁕ 8

3 An ANTLR4 grammar for Autolog ⁕ 11

3.1 AUTOLOG.g4 Antlr source code ⁕ 11

3.2 generating the parser for AUTOLOG.g4 ⁕ 16

3.3 Compiling autolog programs ⁕ 17

3.4 Using the Autolog Editor and Viewer GUI prototypes ⁕ 18

3.5 About Autolog “types” ⁕ 22

3.6 About Autolog rewrite “modulators” ⁕ 23

4 Autocode examples motivating design requirements ⁕ 26

4.1 Type checking ⁕ 26

4.2 Type terms and term complexity ⁕ 26

4.3 Consistency rules ⁕ 27

4.4 Types of equality, equality types and extensionality ⁕ 28

4.5 Kinds of logical negation ⁕ 29

4.6 Universes ⁕ 30

4.7 Formulating axioms for sets ⁕ 31

4.8 Auto lemmas ⁕ 33

4.9 Coherent modal impredicativity ⁕ 34

4.10 Currying and indexicality ⁕ 34

4.11 Parametric functors and data-type concepts ⁕ 35

4.12 Logic theories and meta-interpreters in Autolog ⁕ 39

4.13 Online Modulator and metalogic examples ⁕ 40

5 Autolog Implementation design aspects ⁕ 43

5.1 Branch fact indexing and tableing ⁕ 43

5.2 Fairness ⁕ 50

5.3 Distributive choice algorithm ⁕ 51

5.4 Rule predicativity and indexicality definitions ⁕ 55

5.5 Static and dynamic equality modulators ⁕ 60

5.6 Term profiles and branch fact entanglements ⁕ 69

5.7 Staged concurrent inference actions ⁕ 72

6 Autolog24 specific design proposals ⁕ 76

6.1 Indexicality related definitions ⁕ 76

6.2 match and substitute methods - MatchTest tool and testing ⁕ 82

6.3 Rule literal entanglement methods and TangleTool testing ⁕ 85

6.4 Rewrite entanglement methods and TangleTool testing ⁕ 93

6.5 ActiveRewrite completeness and soundness issues ⁕ 99

6.6 Active Inference Design Scenarios ⁕ 108

6.7 sam.machine.engine.ActiveMod concurrency testing tool ⁕ 109

6.8 sam.machine.active.ActiveTree code notes ⁕ 112

6.9 sam.machine.active.ActiveTheory code notes ⁕ 112

6.10 sam.machine.active.ActiveRule code notes ⁕ 112

6.11 sam.machine.active.ActiveRewrite code notes ⁕ 112

6.12 sam.gui.TreePane code notes ⁕ 112

7 Linked References ⁕ 113

Foreword ⁂

The AUTOLOG DESIGN NOTEBOOK is a *requirements and design documen*t for an extended coherent logic inference programming system and Skolem Machine implementation called ***Autolog***. The Autolog language is an extended machine language for a Skolem Machine originally designed to represent predicative coherent logic rules. The language parsers and translators for Autolog are implemented using ANTLR4 tools. The Autolog language grammar is specified in the file AUTOLOG.g4.

The Autolog Design Notebook uses examples to motivate the language design aspects and to discusses how the new language features are intended to be computed by the Autolog computation engine.

The Autolog programs or theories have been variously characterized as sets of axioms or rules having some general coherent logic formula as its abstract form. Recently, I prefer to think of the theories as a collection of *inference methods* which can apply to branches in the working tree of a Skolem Machine. The methods apply generally to any branch, but can produce different extensions to different branches, depending on current facts on the relevant branch. So, the method metaphor really seems to be closest to the machine (action) aspect of Autolog, and the same metaphor is appropriate for equality modulation implemented as a rewrite action.

The parent machine language for Autolog was called Colog, last posted as a component of the system `colog14I`.

The Autolog Design Notebook builds on technical documentation provided in the Colog language document (colog.pdf) and the Skolem abstract machine document (SAM.pdf), available online.
In particular, various important basic predicative indexing specifications are described in detail in the SAM document. These documents augment some of the presentation in the Autolog Design Notebook:

```
https://skolemmachines.org/reports/colog/colog.pdf
https://skolemmachines.org/reports/SAM/SAM.pdf
```

Autolog has several significant extensions (language and computation) to Colog, including:

types
Autolog implements language features for term typing that coordinate well with general *indexing* algorithms. Examples are provided to motivate a style of Autolog programming using judgements of the form `e1:e2` as a term indexing feature. We could say that expression e1 being indexed by expression e2, whenever e2 is sufficiently bound.

impredicativity
Autolog implements impredicativity or implicit reference to predicates and functions variables, as in `P(F(X))`, where `P` is an implicit predicate, `F` an implicit function, and `X` is a variable term argument for `F`. Variable reference to predicate, function or term variable symbols appearing in the antecedent of an Autolog rule have implicit universal quantification over the entire rule, and free variable appearances of predicate, function or term variable symbols in the consequent of an Autolog rule represent an existential quantification of variables with intended scope over the conjunctive component (containing the implicit reference) of a consequent of an Autolog rule. The implementation of a variable term argument is an internal indexed Variable machine object, and the implementation of a impredicative predicate or function is a special form for an internal `Functor` machine object.

parametric functors

Autolog Functors are either named or parametric (but not both). A named `Functor` has a string `name` attribute and a null fnctr attribute. A parametric `Functor` has a non-null `Term` as its `fnctr` attribute and a null `name` attribute. Expressions like `p(X,Y)` or `f((g(Y))` are stored as named functors, whereas expressions like `p(a)(b,c)`, `p(a)(b,F(c))` or `P(X)(a)(b,F(c))` are stored as parametric `Functors`. This notational liberalism allows one to write Autolog programs involving impredicativity and a mathaematical *functorial* flavor familiar to category theory (for example). This notational extension requires new term matching methods. For example, a parametric literal rule factor `P(F(X))` can match a bound branch fact expression `p(f(a))` via substitutions `P/p`, `F/f` and `X/a`.

Named Functors are always predicative (via the bound name), but onlyparametric Functors whose `fnctr` part is variable-free at input are considered to be predicative, and otherwise the parametric Functor is impredicative (`fnctr` attribute unbound in whole or part at input).

indexicality

The design specification of the new Autolog language (and subsequent changes thereto) must be subject to reasonable methods for matching literal terms in a rule or equality modulator in order to effect the general working inference mechanisms of a Skolem Abstract Machine. Current thinking suggests that some form of indexicality be imposed on the inference methods of an Autolog program. Impredicative functors can be made indexical, using type judgements or other term bindings of implicit predicate or function symbols in Autolog rules and modulators.

DUOPs

DUOPs are *dynamic user defined operators*. Various methods for alowing user-defined operators are currently under study. The goal here is to allow a wide variety of unicode symbols as mathematical or logical symbols to be used in Autolog inference methods (rules or rewrite modulators).

equality/rewrite modulation

Autolog employs a Skolem Machine for extended coherent logic computations. The plan is to augment Autolog with appropriate renditions of equality modulation in order to enable saturation-like capabilities for equality substitutions. It is planned to have (at least) two mechanisms available for equality reasoning: `Rule` equality methods (and dynamic rewrites) and direct equation modulation of the search tree facts using static `Rewrite` methods. Both kinds of equality modulators require some new methods of operation for the underlying machine (but very similar to the application of coherent rule methods.

subprograms

Designing mechanisms for Autolog subprograms, or lemmas), is an important issue. This is a a kind of *coherent cut* concept which requires more theoretical analysis.

This autolog design notebook was started near the beginning of 2018. This document changes often as system development proceeds. The pages version was started in 2020 in order to have a simple way to incorporate unicode text into the design document for easy extraction (copy/paste) into the `AutologEditor`.

logic profile analysis

An enhanced code profiling tool, `sam.machine.lang.AutoLogAnalyzer`, is described. This logic profiler is used to convert internal logical language components into more efficient `sam.machine.search` active components.

Java **implementation code website** : https://skolemmachines.org/autolog/README.html

1 Coherent logic extended, impredicativity ⋇

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

$$\forall \hat{x}(a_1 \wedge \ldots \wedge a_m) \rightarrow \exists \hat{y}_1(b_{11} \wedge \ldots \wedge b_{1k_1}) \vee \ldots \vee \exists \hat{y}_n(b_{i_n1} \wedge \ldots \wedge b_{i_nk_{i_n}})) \quad (1.1)$$

where `m,n ≥ 0`, each a_i and b_{jk} are literal predicate expressions, 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 in the consequent of formula. It is assumed that the formula is closed and that none of the universal variable appear among any of the existential variables. Any predicate or function symbols appearing in any of the literals is assumed to be *named*, using conventional alphabetic or symbolic symbols for names.

When `m=0`, formula (1.1) is conventionally written as

$$\forall \hat{x}(\top \rightarrow \exists \hat{y}_1(b_{11} \wedge \ldots \wedge b_{1k_1}) \vee \ldots \vee \exists \hat{y}_n(b_{i_n1} \wedge \ldots \wedge b_{i_nk_{i_n}})) \quad (1.2)$$

or also

$$\forall \hat{x}(\exists \hat{y}_1(b_{11} \wedge \ldots \wedge b_{1k_1}) \vee \ldots \vee \exists \hat{y}_n(b_{i_n1} \wedge \ldots \wedge b_{i_nk_{i_n}})) \quad (1.3)$$

On the other hand, when `n=0` the formula (1.1) is conventionally written as

$$\forall \hat{x}(a_1 \wedge \ldots \wedge a_m \rightarrow \bot) \quad (1.4)$$

Named function expressions are allowed as arguments for predicates in the coherent logic formulas above. The *named* aspect of this coherent logic is the crucial aspect of its *predicativity.* For example, consider the rule

$$(\forall x)(\forall y)[p(x, f(y)) \rightarrow (\exists z)q(x, z) \vee (\exists z)r(y, f(z))] \quad (1.5)$$

Am autocode version of this rule would be

```
«
      p(X,f(Y)) => q(X,Z) | r(Y,f(Z)).                        %(1.5a)
»
```

All of the functor expressions use named functors, with input names ‘`p`’, ‘`f`’, ‘`q`’ and ‘`r`’.

Impredicative coherent logic allows the quantifications to range over predicates and functions. An important example would be impredicative definitions for extensional equality.

$$(\forall p)(\forall x)(\forall x)(x = y \wedge p(x) \rightarrow p(y)) \quad (1.6)$$
$$(\forall f)(\forall x)(\forall x)(x = y \rightarrow f(x) = f(y)) \quad (1.7)$$

Autolog versions of (1.5) and (1.6) could be

```
«
      X=Y, P(X) => P(Y).                                      %(1.6a)
      X=Y => F(X)=F(Y).                                       %(1.7a)
»
```

If the reader copies a selection including `autocode` «...» from this document and pastes it into the `AutologEditor`, the editor will parse and compile the `autocode` to the `AutologViewer`.

Formulas (1.5) and (1.6) could be made more ***indexical*** via the following rules, using typed quantifications.

$$(\forall p : t \rightarrow prop)(\forall x : t)(\forall y : t)[x = y : t) \wedge p(x) \rightarrow p(y)] \quad (1.8)$$
$$(\forall f : t \rightarrow s)(\forall x : t)(\forall x : t)[x = y : t \rightarrow f(x) = f(y) : t] \quad (1.9)$$

Autolog versions of (1.7) and (1.8) could be

```
«
  T:type, P:T→prop, X=Y:T, P(X) => P(Y).                %(1.8a)
  T:type, S:type,F:T→S, X=Y:T: => F(X)=F(Y):S.          %(1.9a)
»
```

The leading antecedent predicative literals (if matched first) effectively ***index*** the impredicative occurrences of P and F in the rest of the antecedent of the rule, and so also th econsequent of the rule. (We did not use typing quantifiers t and s for 1.7 or 1.8, so we leave that invention to the reader. Variables can be used to represent types also.)

An example of impredication in the consequent of a rule could be the following coherent rules expressing the existence of a function having at least two arguments for a particular type t.

$$\top \rightarrow t : type \quad (1.10)$$
$$\top \rightarrow (\exists f)f : t \rightarrow t \rightarrow t \quad (1.11)$$

Autocode versions could look like this

```
«
   true => t:type.                                      (1.10a)
   true => F:t→t→t.                                     (1.11a)
»
```

Where F is an impredicative function (variable).

Autolog also allows more generalized functor form (all compiled as Functor objects). The more general terminology is parametric functor forms. Impredicative function and predicate forms are a subtype of parametric functors.

For a concrete simple example, consider the functional device used to prove Peirce's representation theorem for algebraic groups: every algebraic group is isomorphic to the group of left-multiplications of itself (or right). Here, we are only going to look at a way to represent the functional device as a parametric functor, and not give a full outline for the algebraic group theory, nor a proof of Peirce's theorem.

$$(\forall g : group)(\forall x \in g)[left(x) : g \rightarrow g] \quad (1.11)$$
$$(\forall x, y \in g)[left(x)(y) = x \circ y] \quad (1.12)$$

Autocode version of these formulas might look like the following.

```
«
   G:group, X∈G => left(X):G→G.                         %(1.11a)
   X∈G, left(X):G→G, Y∈G => left(X)(Y)=X∘Y.             %(1.12a)
```

»

An Autolog machine language Functor object represent either named or parametric functions and predicates, both of which are objects. A `Functor` object has both a `name` atribute and a `fnctr` attribute. Named functors have a `String name` and a `null fnctr` attribute. Impredicative (variable) functors have a `Variable fnctr` attribute and a `null name` attribute. General parametric functors have a `Functor` for their `fnctr` attribute. We will revisit the details of these internal code structures several times in this Notebook. The details are especially important for the matching algorithms that match `Variable` and `Functor` expressions against Autolog branch facts (`Functors`) in a proof search.

These explicit examples do not exhaust the possibilities of extended coherent logic expressions containing impredication and parametric function and predicate `Functors`. But we have enough here to work with in order to explain how it will be that a Skolem Abstract Machine would compute Autolog's extended coherent form logic.

Here are some references that the student might consult now, while pondering this and the next chapter:

https://plato.stanford.edu/archives/fall2014/entries/type-theory/

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

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

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

https://plato.stanford.edu/entries/type-theory-intuitionistic/

As mentioned above, if the reader copies a selection including autocode «…» from this document and pastes it into the AutologEditor, the editor will parse and compile the autocode to the AutologViewer. Also, reviewing the analyzed structure of the code (AutoLogAnalyzer) in the Viewer should confirm the indexicality patterns of the inputs.

In the next chapter we give a short introduction to impredicativity management via various patterns of indexicality.

2 Impredicativity Management via Indexicality ⁂

This chapter considers naive aspects of impredicativity management for the Skolem abstract machine. Later chapters and sections will dive into the implementation technicalities. The primary tool for impredicativity management is the use of \emph{indexicality} to bind variables in predicates and functions.

A very simple example illustrates the essential requirements for using an impredicative rule.

```
«                                                              %(2.1)
   % 1 prove successor of 0 is int
     s(0):int => goal.
   % 2 successor function
     true => 0:int, s:int→int.
   % 3 impredicative rule
     F:T→S, X:T => F(X):S.
»
```

Note that rule 3 has both a function quantification in the antecedent and a fact assertion for the matching function in the consequent. A proof for the simple theory is Ilustrated in Figure 2.1.

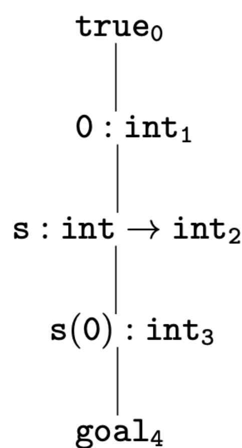


Figure 2.1: Skolem machine proof tree

At tree node `0`, rule 2 asserts facts shown as node 1 and node 2 on the branch. At node 2 rule 3 matches its antecedent against facts for node 1 and node 2 `F/s`, `X/0`, `T/int` and asserts its bound consequent as node 3 on the branch, and then rule 1 applies to give the proof.

There are several other issues to consider, which we will illustrate using more examples . Consider the following impredicative Autolog rule, corresponding to formula $(\forall p)(\forall x)(p(x) \rightarrow p(f(x))$:

```
«                                                        %(2.2)
   P(X) => P(f(X)).
»
```

One difficulty with such a autocode rule is that its antecedent would not have its predicate functor P bound at the time when branch matches are sought. A decision must be made regarding whether or not to insist that {\tt P}must be bound as a term variable in another literal factor of the antecedent, such as

```
«                                                        %(2.3)
   P:T→prop, P(X) => P(f(X)).
»
```

Note that if the first literal factor is matched to a fact on branch (where P occurs as a term variable), then the second factor `P(X)` has a bound predicate functor and branch fact matching by functor is used to find possible matches on branch.

A predicate literal expression or function expression of the form `F<args>` either has a name specified at input for its F part or the F part is itself a Functor expression. If the F part is a Functor with an occurrence of a variable, then the F<args> expression is said to be ***impredicative***. If the F part is a name (`String`) then the `F<args>` expression is said to be to ***predicative***. The `F<args>` expression is said to be ***indexical*** provided that any variables in the F part would be bound (named) by matching a rule factor to the left of the actual input code occurrence of F. It follows that if F<args> is predicative then it is indexical.

Notice that *if* the previous rule had been input as

```
«                                                       %(2.4)
   P(X), P:int→prop => P(f(X)).
»
```

then the `P(X)` in the antecedent would not be indexical, assuming that the literals are matched to in a left-to-right manner.

Continuing with this discussion, now consider the rule

```
«                                                       %(2.5)
   P:int→prop, P(X), F:int→int => P(F(X)).
»
```

It is clear that the antecedent literal `F:int-int` provides a binding context if it can be matched on branch, and then that binding can be used to decide whether to assert the consequent to branch.

Now consider another variation of the preceding rule

```
«                                                       %(2.6)
   P:int→prop, P(X) => F:int→int, P(F(X)).
»
```

where the function F is existentially scoped in the consequent. Again note that a term binding of the function is required before the resulting proposition is sufficiently bound to match branch facts, using colog implementation methods.

Assuming that the consequent type judgement is matched first, then the function in the last literal factor is sufficiently bound in order to decide whetherthe proposition is new, and if so assert its bound value to branch.The new value will involve a new Skolem function, arising viaa new Skolem *eigenvariable* in the existential literal `F:int→int`. In order to make this discussion a little more concrete, consider the following theory, and then the proof tree in Figure 2.2.

```
«                                                       %(2.7)
   true => 0:int, q:int→ prop, q(0).
   P:int→prop, P(X) => F:int→int, P(F(X)).
   F:int→int, q(F(X)) => goal.
»
```

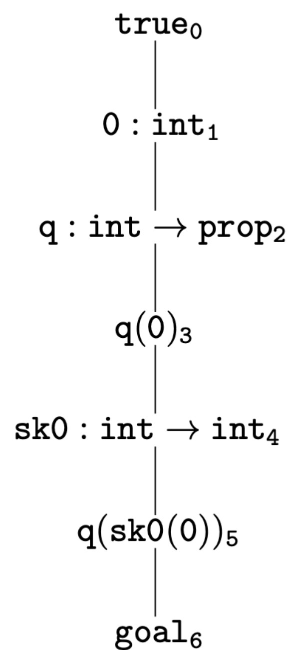


Figure 2.2: Proof tree with Skolem function

Exercise: For each of the following antecedent literals, describe a method that might make finding a matching fact on the current search branch more efficient, such as *tabeling* and also conjecture what specific indexing method might be used, base on the degree of indexicality of the literal factor:

```
«                                                    %(2.8)
   q => goal.
   Q => goal.
   q(a) => goal.
   q(X) => goal.
   Q(a) => goal.
   Q(X) => goal.
   q(a)(b) => goal.
   q(a)(X) => goal.
   q(X)(b) => goal.
   Q(a)(b) => goal.
   Q(a)(X) => goal.
   Q(X)(b) => goal.
   Q(X)(Y) => goal.
»
```

Most of the relevant implementation issues are touched on by the brief examples in this introductory chapter. A more detailed analysis of matching methods employs what we call *literal factor analysis*, which is covered as an implementation detail in Chapter 5. Factor analysis describes how indexicality (or partial indexicality) influences the retrieval of branch facts that could match rule factors.

3 An ANTLR4 grammar for Autolog ⁎

In this chapter we consider the ANTLR4 grammar specifying Autolog expressions which allow parametric functors and impredicativity. We then consider the parse-tree walker that generates the compiled Autolog machine code structures. It may well be that, in order to implement certain structures for impredicativity and other language features, we may have to modify the current grammar in some regard. As indicated by the examples, the grammar also needs type judgements to constrain impredicativity, in accordance with the way that the abstract Skolem machine will intend to compute impredicativity. The AUTOLOG.g4 grammar is a kind of "meta-coherent-logic": The intention is that the AUTOLOG.g4 grammer be able to read and compile all sources written in accordance with the previous Colog.g grammar from colog14l, as well as new language features. The antler website is https://www.antlr.org.

We want the antlr grammar to read user codes left-to-right like intuitive natural language textual presentations of algeba and logic, so that symbolic term bindings occur as would be the case in most math and logic texts, with parens imposed for restricting term bindings. DUOPs are "meta" ops having less tight parse binding because they are lower among the antler term grammar rules. The higher term rules parse more tightly because they get to bind before lower rules can try, unless parens hold terms together.

3.1 AUTOLOG.g4 Antlr source code ⁎

```
/**
  *  AUTOLOG.g4    "parametric coherent logic"
  *  AutoLog programming language grammar,
  *  coherent logic forms, Type terms/term types, impredicativity
  *  functorials (parametric functors), modulators.
  *  @version 10/5/2020 (# term_collection)
  */
grammar AUTOLOG ;

@header{
package sam.antlr.autolog ;
}

theory : inference* ; // inference = rule or modulator

inference :
   conjunction INFER disjunction '.'  # inference_rule
   |  term EQ term '.'                # inference_rewrite
   ;
disjunction :
   conjunction (('|' | ';') conjunction)*
   ;
conjunction :
   literal (',' literal)*
   ;
literal : // always translate as Functor
   // impredicative prop
```

```
   VARIABLE                  # literal_impprop
   // terms other than impprop
   | term                    # literal_term
   ;
term :    // earlier case forces tighter binding
   NUMBER                   # term_number
   | VARIABLE               # term_variable
   | name                   # term_name
   | term '(' terms ')'     # term_functor // ALLOWS PARAMETRIC
   | POSSIBLE term          # possible_term
   | NECESSARY term         # necessary_term
   | GEN term               # gen_term
   | NEG  term              # term_neg
   | LNEG term              # term_lneg
   | MINUS term             # term_minus
   | SHARP term             # term_sharp
   | term STAR term         # term_star
   | term CIRCLE term       # term_circle
   | term CCIRCLE term      # term_ccircle
   | term AND term          # term_and
   | term CCFLEX term       # term_ccflex
   | term MEET term         # term_meet
   | term INTERSECT term    # term_intersect
   | term JOIN term         # term_join
   | term UNION term        # term_union
   | term PLUS term         # term_plus
   |<assoc=right>          // right associative
     term IF term           # term_if
   |<assoc=right>          // right associative
     term ARROW term        # term_arrow
   |<assoc=right>          // right associative
     term BSLASH term       # term_bslash
   | term FI term           # term_fi
   | term LARROW term       # term_larrow
   | term FSLASH term       # term_fslash
   | term MEMBER term       # term_member
   | term EQ term           # term_eq
   | term IFF term          # term_iff
   | term EQUIV term        # term_equiv
   | term NEQ term          # term_neq
   | term LESS term         # term_less
   | term LESSEQ term       # term_lesseq
   | term GREATER term      # term_greater
   | term GREATEREQ term    # term_greatereq
```

```
   | '(' t=term ')'         # term_paren
   | term ':' term          # term_judgement
   /////  DUOPs dynamic user operators (copy/pasted)
   | DUOP '(' terms ')'     # term_functional_DUOP
   | DUOP term              # term_prefix_DUOP   // pre << post << in
   | term DUOP term         # term_infix_DUOP
   /////  end DUOPs
   | '{' term '|' term '}'  # term_collection  // Functor type 4
   | '(' term ',' term ')'  # term_pair        // Functor type 5
   ;
terms :
   t=term (',' t=term )*
   ;
name :
   ALL|SOME|LAMBDA|PI|SUM|TOP|BOTTOM|POSSIBLE|NECESSARY
   |GEN|NEG|LNEG|EQ|IFF|EQUIV|NEQ|STAR|CIRCLE|CCIRCLE|AND
   |CCFLEX|MEET|INTERSECT|JOIN|UNION|MEMBER|EMPTY|PLUS|MINUS
   |IF|BSLASH|FI|FSLASH|ARROW|LARROW|SHARP|LESS|LESSEQ
   |GREATER|GREATEREQ|LOWER|UPPER|NUMERIC|QUOTED|DUOP|LNAME
   ;
LNAME :  // lexed general name
   LOWER (LOWER | UPPER | '_' | '-' | NUMERIC )*
   ;
NUMBER :
   NUMERIC+
   | NUMERIC+ '.' NUMERIC+
   ;
VARIABLE : // similar to Prolog
   (UPPER | '_' | '?')  (LOWER | UPPER | '_' | NUMERIC)* /
   ;
ALL :     '∀' ; // logic forall
SOME :    '∃' ; // logic forsome
LAMBDA :  'λ' ; // Greek lambda
PI :      'Π' ; // Greek Pi
SUM :     'Σ' ; // Greek Sigma
TOP :     '⊤' ; // top, truth
BOTTOM :  '⊥' ; // bottom, falsum
POSSIBLE : '◇'  ; // modal,  cta (click edit button to add)
NECESSARY : '□' ; // modal cta
GEN :     '@'  ;
NEG :     '~'  ;
LNEG :    '¬'  ; // logic negation, option l
EQ :      '='  ;
IFF :     '↔' ; // cta
```

```
EQUIV :    '≈' ; // option x
NEQ :      '≠'  ; // not equal, option =
STAR :     '*'  ;
CIRCLE :   '◦'  ;  // cta
CCIRCLE :  '•'  ;  // cta
AND :      '&'  ;
CCFLEX :   '^'  ;  // shft+6, e.g. 2^3 (=8)
MEET :     '∧'  ;  // cta
INTERSECT : '∩'  ;  // cta
JOIN :     'v'  ;  // cta
UNION :    '∪'  ;  // cta
MEMBER :   '∈'  ;  // cta
EMPTY :    '∅'  ;  // cta
PLUS :     '+'  ;
MINUS :    '-'  ;
IF :       '→'  ;  // cta
BSLASH:    '\\' ;  // backslash
FI :       '←'  ;  // cta
FSLASH:    '/' ;
ARROW :    '->' ;
LARROW :   '<-' ;
SHARP :    '#'  ;
LESS  :    '<'  ;
LESSEQ :   '≤'  ;  // option <
GREATER :  '>'  ;
GREATEREQ: '≥'  ;  // option >
LOWER :    'a'..'z'    ;
UPPER :    'A'..'Z'    ;
NUMERIC :  '0'..'9'    ;
QUOTED :    // embed double quoted "data"
   // e.g., "This is a comment."
   '\u{0022}' .*? '\u{0022}'  // "any" charseq
   ;
DUOP :  // Dynamic User Operators
        // A- copy these from table and paste in editor
    // arrows
          '↠'|'↰'|'⇀'|'⇐'|'⇠'|'⇰'
     |'↑'|'↡'|'↱'|'⇁'|'⇑'|'⇡'|'⇱'
     |    '↢'|'↲'|'⇂'|'⇒'|'⇢'|'⇲'
     |'↓'|'↣'|'↳'|'⇃'|'⇓'|'⇣'|'⇳'
     |    '↤'|'↴'|'⇄'|'⇔'|'⇤'|'⇴'
     |'↕'|'↥'|'↵'|'⇅'|'⇕'|'⇥'|'⇵'
     |'↖'|'↦'|'↶'|'⇆'|'⇖'|'⇦'|'⇶'
     |'↗'|'↧'|'↷'|'⇇'|'⇗'|'⇧'|'⇷'
```

```
      |'↘'|'↨'|'↸'|'⇈'|'⇘'|'⇨'|'⇸'
      |'↙'|'↩'|'↹'|'⇉'|'⇙'|'⇩'|'⇹'
      |'↚'|'↪'|'↺'|'⇊'|'⇚'|'⇪'|'⇺'
      |'↛'|'↫'|'↻'|'⇋'|'⇛'|'⇫'|'⇻'
      |'↜'|'↬'|'↼'|'⇌'|'⇜'|'⇬'|'⇼'
      |'↝'|'↭'|'↽'|'⇍'|'⇝'|'⇭'|'⇽'
      |'↞'|'↮'|'↾'|'⇎'|'⇞'|'⇮'|'⇾'
      |'↟'|'↯'|'↿'|'⇏'|'⇟'|'⇯'|'⇿'
     // greek
      |'α'|'β'|'γ'|'δ'|'ε'|'ζ'|'η'|'θ'|'ι'|'κ'|    'μ'|'ν'
      |'Α'|'Β'|'Γ'|'Δ'|'Ε'|'Ζ'|'Η'|'Θ'|'Ι'|'Κ'|'Λ'|'Μ'|'Ν'
      |'ξ'|'ο'|'π'|'ρ'|'ς'|'σ'|'τ'|'υ'|'φ'|'χ'|'ψ'|'ω'
      |'Ξ'|'Ο'|    'Ρ'|    'Τ'|'Υ'|'Φ'|'Χ'|'Ψ'|'Ω'
     // math ops - 1st is KB opt - (e.g. subtraction)
   |'−'|'∐'|'∠'|'∰'|'≀'|'≐'|    '≰'|'⊀'|'⊐'|'⊠'|'⊰'|'⋀'|'⋐'|'⋠'|'⋰'
   |'∁'|'∑'|'∡'|'∱'|'≁'|'≑'|'≡'|'≱'|'⊁'|'⊑'|'⊡'|'⊱'|'⋁'|'⋑'|'⋡'|'⋱'
   |'∂'|'−'|'∢'|'∲'|'≈'|'≒'|'≢'|'≲'|'⊂'|'⊒'|'⊢'|'⊲'|'⋂'|'⋒'|'⋢'|'⋲'
       |'∓'|'∣'|'∳'|'≃'|'≓'|'≣'|'≳'|'⊃'|'⊓'|'⊣'|'⊳'|'⋃'|'⋓'|'⋣'|'⋳'
   |'∄'|'∔'|'∤'|'∴'|'≠'|'≔'|    '≴'|'⊄'|'⊔'|    '⊴'|'⋄'|'⋔'|'⋤'|'⋴'
   |    '∕'|'∥'|'∵'|'≅'|'≕'|    '≵'|'⊅'|'⊕'|    '⊵'|'⋅'|'⋕'|'⋥'|'⋵'
   |'∆'|'∖'|'∦'|'∶'|'≆'|'≖'|'≦'|'≶'|'⊆'|'⊖'|'⊦'|'⊶'|'⋆'|'⋖'|'⋦'|'⋶'
   |'∇'|'∗'|    '∷'|'≇'|'≗'|'≧'|'≷'|'⊇'|'⊗'|'⊧'|'⊷'|'⋇'|'⋗'|'⋧'|'⋷'
   |            '∸'|    '≘'|'≨'|'≸'|'⊈'|'⊘'|'⊨'|'⊸'|'⋈'|'⋘'|'⋨'|'⋸'
   |'∉'|        '∹'|'≉'|'≙'|'≩'|'≹'|'⊉'|'⊙'|'⊩'|'⊹'|'⋉'|'⋙'|'⋩'|'⋹'
   |'∊'|'√'|    '∺'|'≊'|'≚'|'≪'|'≺'|'⊊'|'⊚'|'⊪'|'⊺'|'⋊'|'⋚'|'⋪'|'⋺'
   |'∋'|'∛'|'∫'|'∻'|'≋'|'≛'|'≫'|'≻'|'⊋'|'⊛'|'⊫'|'⊻'|'⋋'|'⋛'|'⋫'|'⋻'
   |'∌'|'∜'|'∬'|'∼'|'≌'|'≜'|'≬'|'≼'|'⊌'|'⊜'|'⊬'|'⊼'|'⋌'|'⋜'|'⋬'|'⋼'
   |'∍'|'∝'|'∭'|'∽'|'≍'|'≝'|'≭'|'≽'|'⊍'|'⊝'|'⊭'|'⊽'|'⋍'|'⋝'|'⋭'|'⋽'
   |'∎'|'∞'|'∮'|'∾'|'≎'|'≞'|'≮'|'≾'|'⊎'|'⊞'|'⊮'|'⊾'|'⋎'|'⋞'|'⋮'|'⋾'
   |'∏'|'∟'|'∯'|'∿'|'≏'|'≟'|'≯'|'≿'|'⊏'|'⊟'|'⊯'|'⊿'|'⋏'|'⋟'|'⋯'|'⋿'
    // B- single quoted 'anything'  e.g.,  X 'has' Y
    | '\'' .*? '\''             // 'any' char seq
    ;
WS : [ \t\n\r]+ -> skip ; // ignore all whitespace
SL_COMMENT : ('%'|'//') .*? ('\n'|'\r') -> skip ;
ML_COMMENT : ('/*' | '(*') .*? ('*/' | '*)' )  -> skip ;
INFER   : '=>' ;
/*
  - name cannot be lex
  - DUOPs are "meta" ops having less tight parse binding because
    they try to parse later in the grammar rule order.  The higher rules
    parse more tightly because they get to bind before lower rules can try,
    unless parens hold terms together.
  - Note that parametric functors MUST have TYPE FUNCTOR.FUNCTIONAL
```

```
      = (byte)0.
    - We are abandoning parse for postfix ops, for now.
 */
```

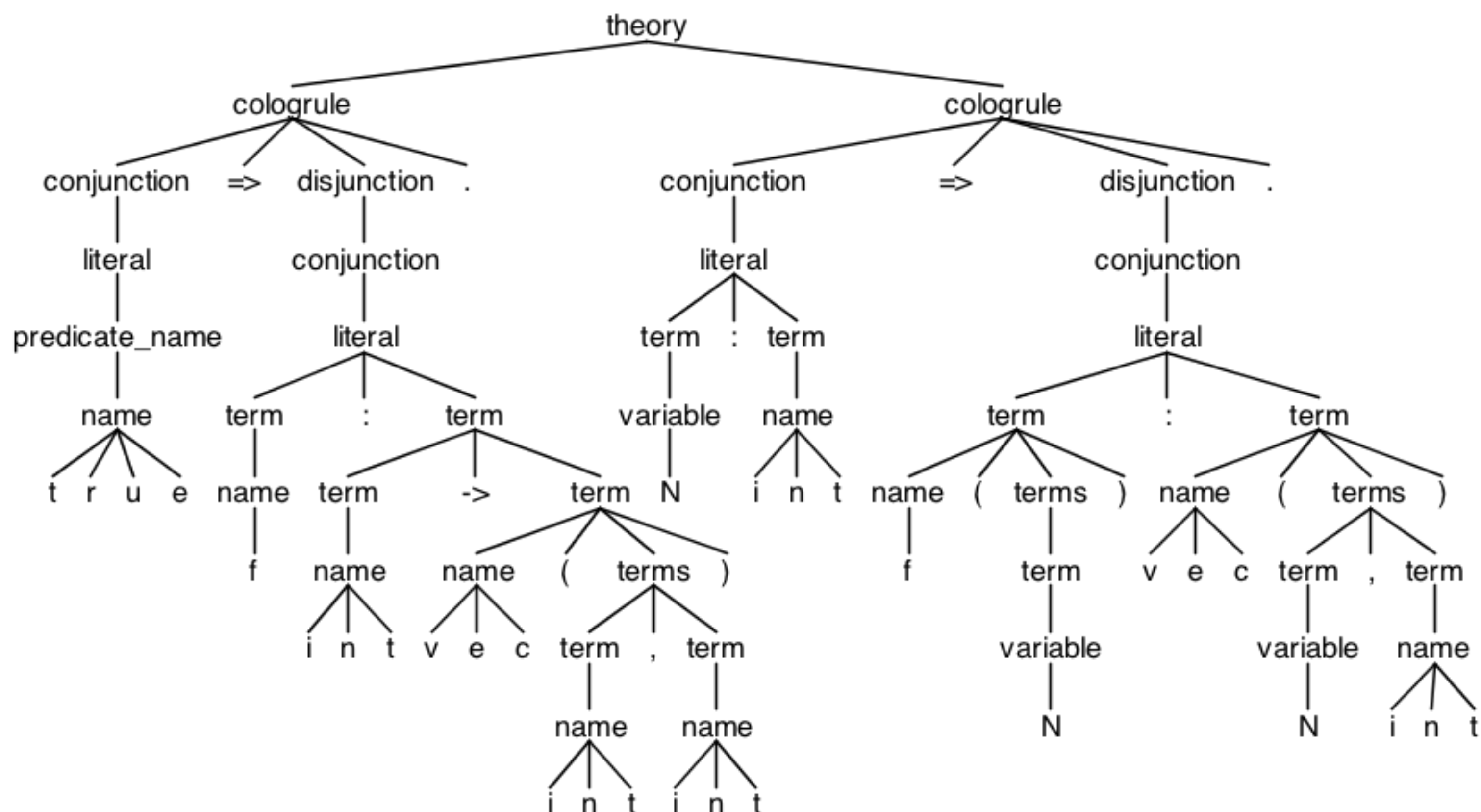
3.2 generating the parser for AUTOLOG.g4 ※

The source code files associated with Autolog are linked at the autolog development page:

```
https://skolemmachines.ORG/autolog/README.html
```

Following the instructions on the development page, here is a sample command line interaction with our grammar:

```
sam/antlr/autolog> source aliases
     ...
sam/antlr/autolog> antlr4       # creates parser
sam/antlr/autolog> compile      # compiles the parser
sam/antlr/autolog> test         # testkit, sample user input
    true => f:int→vec(int,int).   % f has type "loose" product
    N:int => f(N):vec(N,int).     % f is dependent
 ^D
```

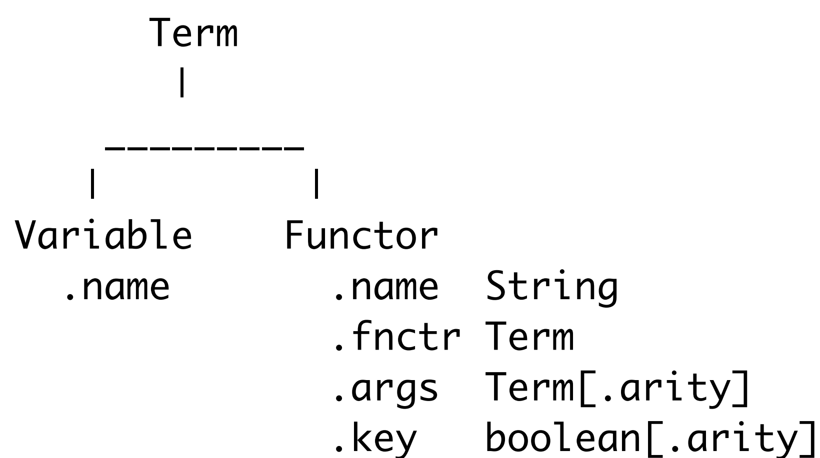


The snapshot shows the parse tree displayed after the ^D. It is recommended that the reader use this graphical test device to parse and inspect a variety of theory inputs, including those from the Notebook. Many aspects of the syntactic design adequacy of the grammar can be probed using the inspection tool.

3.3 Compiling autolog programs ※

A user *program* is a sequence of rules and modulators. The program is compiled into internal form using component `sam.antlr.Compiler`, which calls `sam.antlr.Reader` in order to translate parsed expressions into run-time Java component structures — inference methods.The `Reader` has tree-walker methods which construct `sam.machine.lang` components which are used to build Rules and Modulators which comprise a `sam.machine.lang.Theory` generated by the input. In turn, The `sam.machine.lang` components are used to construct the active `sam.machine.search` components which perform the actual inference computations at run time.

Of particular interest for later discussion of indexing methods and inference actions in Chapter 5 are the `sam.machine.lang` components `Term`, `Variable` and `Functor`.



For example, consider how functor terms are translated using the `#term_functor` grammar rule in the AUTOLOG.g4 grammar. The following is the specific code for this translation lifted from `Reader.java`:

```
/////----- # term_functor
//  term: ...  |  term   '(' terms ')'  # term_functor
//  "t(t1,...,tn)" will parse to stack=[tn,...,t2,t], with tn on top.
public void exitTerm_functor(AUTOLOGParser.Term_functorContext ctx)  {
   int n = ctx.terms().term().size() ;  // how many terms
   Term[] arguments = new Term[n] ;
   // pop args from stack
   for (int i = n-1 ; 0 <= i ; i--)
      arguments[i] = stack.pop() ;  // reverse
   Term term = stack.pop() ;
   Functor f ;
   // term is Variable, e.g., P(...)
   if (term instanceof Variable)
      f = new Functor("",term,arguments,(byte)0) ;
   // term is name or built-in
   else if (term instanceof Functor && ((Functor)term).args==null)
      f = new Functor(((Functor)term).name,null,arguments,(byte)0) ;
   // term is functorial
   else
      f = new Functor("",term,arguments,(byte)0) ;
   stack.push(f) ;
}
```

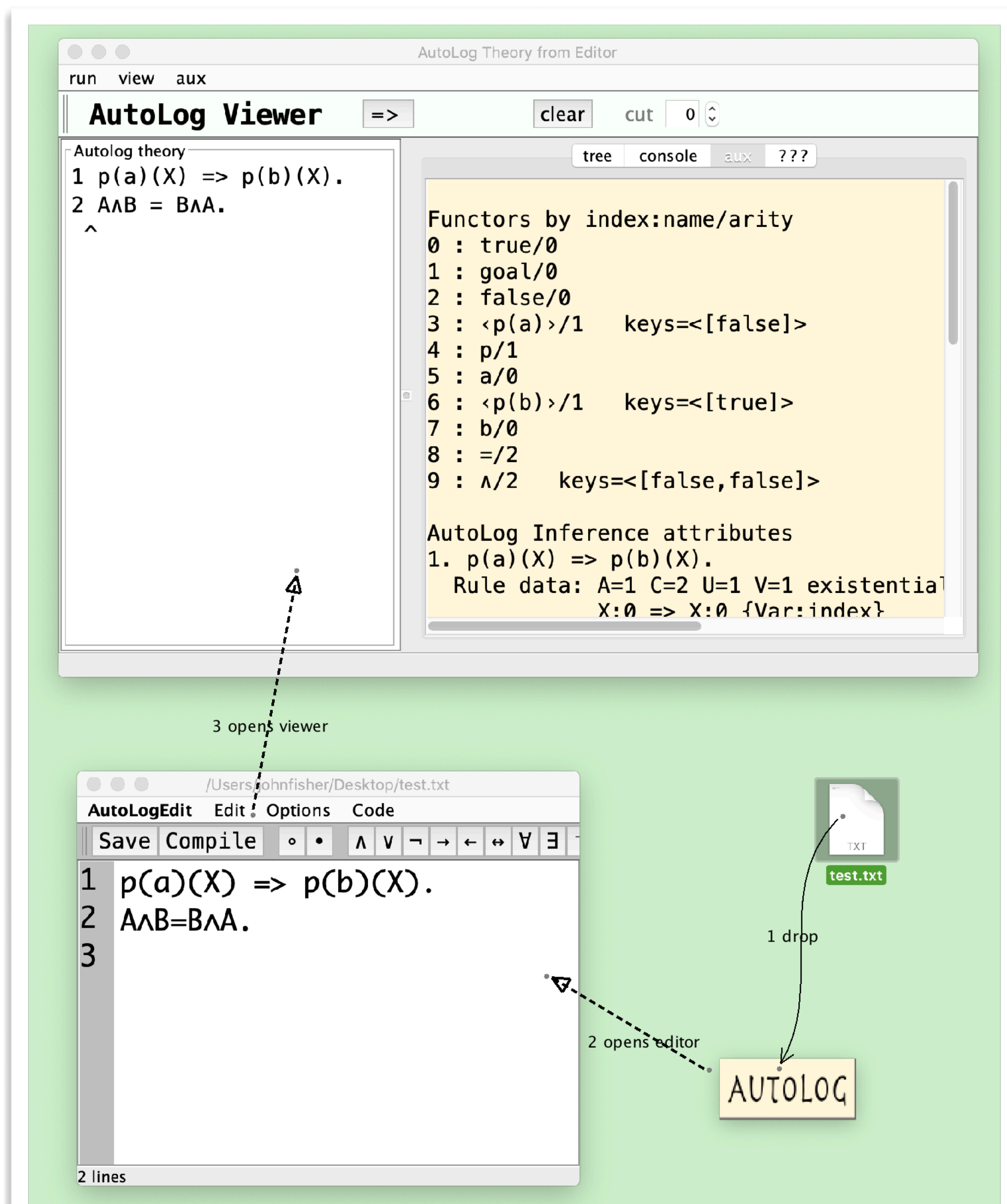
This sample walker code for term translation to `sam.machine.lang.Functor` components illustrates some typical translation patterns.

The next section discusses autolog GUI prototypes which are very convenient for development testing.

3.4 Using the Autolog Editor and Viewer GUI prototypes ※

Testing the (evolving) AUTOLOG grammmar and the compiled source codes is supported by two essential GUI components, the Autolog Editor and the Autolog Viewer. Initial test cycles use the methods outlined in §3.2 (antlr parser generation) and §3.2 (Reader and compiler construction of Skolem machine codes). Enhanced development tools are included in the GUI components as code analysis tools, primarily the `sam.machine.lang.AutoLogAnalyzer`. Later versions of the viewer will include the graphical display of the search engine results and the inference control components (similar to colog14l).

The following screenshot show the appearances of the editor and viewer along with some notations regarding user interactions.



The particular scenario depicted in the screenshot is (1) drag and drop a source file onto the autolog desktop ikon. This causes the desktop app to (2) open the file , using the Reader to convert the source code into `sam.machine.lang` components, which are then displayed in the open editor. (The Reader translation is only the first stage of compilation …) If the Compile button on the editor toolbar is clicked (3), the machine components are further compiled into `sam.machine.search` components in and displayed in the viewer. The machine components from the editor are first analyzed before creating search components and the result of the analysis is displayed in the **aux** tab of the right pane.

The Reader "compiles" the logic of the text, using the recogition grammar into machine recognition logic components. The autolog viewer "compiles" the machine recognition components into machine active inference components (to be “interpreted” by the JVM). The informal term "translates" also works as a description of what the autolog "compiling" accomplishes.

The **Code** menu on the editor window has two menu items concerning autolog coding. The “code info” selection shows the information shown below in screenshots (HTML, which displays in editor pop-up).

Autolog coding information

1- Builtin Autolog operators

The following builtin symbolic operators can be used by the programmar. The operators are listed in order of precedence: An operator listed before another binds more tightly in contexts without parens. Otherwise use parens. There is no fixed prior meaning assumed for the operators. The programmer's inference rules or rewite modulators determine *intended semantics*.

```
name (functor)
    ⊤          editor (22A4)
    ⊥          editor (22A5)
    ∀          editor (2200)
    ∃          editor (2203)
    λ          editor (3BB)
    ∅          editor (2205)
    Π          editor (3A0)
    Σ          editor (3A3)
unary prefix operator
    @
    -
    ¬  option l editor (AC)
    ~
    #
    ◇           editor (25C7)
    □           editor (25A1)
binary infix operator
    =
    ≠   option =
    ≈   option x
    ≤   option <
    ≥   option >
    ∧           editor (2227)
    ∨           editor (2228)
    ->
    →           editor (2192)
    \
    <-
    ←          editor (2190)
    /
    ↔           editor (2194)
    +
    *
    &
    ^
    ∘           editor (2218)
    ∙           editor (2219)
    ∩           editor (2229)
    ∪           editor (222A)
    ∈           editor (2208)
```

Symbols marked "editor (**unicode**)" are available to paste from the Autolog editor toolbar. The **name**s can be used as functors with arity determined by source context. The rest of the symbols are keyboard symbols. The right arrows are right associative. The keyboard - is intended for use as a prefix "minus". For subtraction the programmer might use – (option -) as an infix subtraction operator, as in 3–5=-2. (See DUOPs below.)

2- Parametric functor terms and predicates

A Parametric functor term appears as a function application which itself is applied to an argument. Examples could be like the

```
p(a)(b)
(f*g)(X)=f(X)*g(X)
F(G∘H)=F(G)∘F(H)
F(G∘H)(X)=F(G)(X)∘F(H)(X)
(F(G)∘F(H))(X)=F(G)(X)∘F(H)(X)
```

The third expression is not functorial but the fourth and fifth expressions are. The last three expressions are impredicative (variable ops). Parametric functors must be expressed using applied function expressions such as the examples (no prefix or infix versions can be parsed).

3- Dynamic user specified operators (DUOPs)

The Autolog programmer can designate functional(...), prefix unary, infix binary operators using unicode symbols pasted into source code. The unicode symbols are copied from a convenient table source. For example, we could use op symbol α in various ways,

```
α(X,Y) => goal.   // functional predicate
αX=X.             // prefix op in rewrite
XαY => YαX.       // infix predicate in rule
XαY=YαX,          // infix op in rewrite
```

It is of course not recommended to use operators in more than one intended arity/position context. The examples merely emphasize that the dynamic occurrence of the op in source code designates its intended position and arity usage. The built-in ops of the first section above have more precedence (bind tighter) than DUOPs (but all ops bind tighter using parens).

One finds lists of math/logic unicode symbols at various online locations, for example
`https://en.wikipedia.org/wiki/Mathematical_operators_and_symbols_in_Unicode`
Test such a found symbol by copying it from the reference and then paste it in the source code. To test whether the symbol renders in a reasonable manner, compile the source and view the display of the operator in the AutoLog Viewer. The Autolog editor displays a handy list of symbols (**Code** menu, ⇑⌘I).

Operator words are possible, as in

```
mary' likes 'henry
```

which then displays in compiled form in the same mannner (with the single quotes).

4- compile

Use the **Compile** menuitem under the **Code** menu, or use the **Compile** button on the toolbar. The current code in the Autolog editor is translated and displayed in the Autolog Viewer. Any errors are displayed in the AutoLog Viewer **console**.

If the input document contains autolog code lines surrounded by parens « ... » (keyboard option | and shift-option |, respectively), then the compile command will first extract the embedded autolog code befor compiling it. All text outside of these parens is treated as commentary, and the text «inside the parens» is presumed to be autolog code.

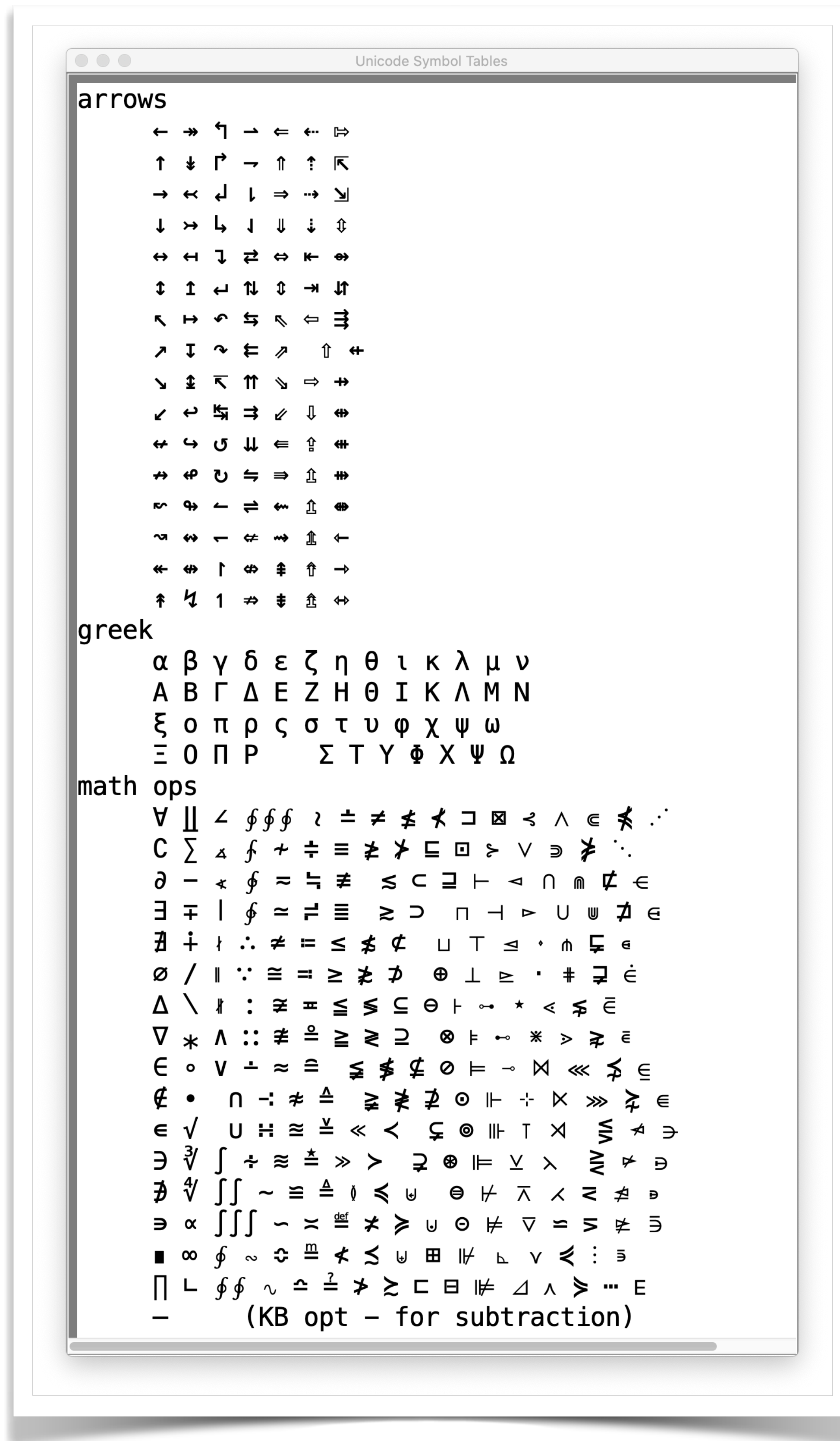
9/4/2020

The problem of accurate embedding of unicode into an edit tool seems annoying and deficient, usually requiring work-arounds or cumbersome approximations, such as the following autolog device.

To view the "code info" (current version) in your browser use the weblink

https://skolemmachines.org/autolog/help/Info.html

Another item under the editor Code menu is “unicode symbols” and a pop-up showing the following appears. The programmer can copy symbols from the symbol tool and paste into the source code of the editor.



The link to the plain text source (for download, not viewing this unicode) is

https://skolemmachines.org/autolog/help/symbols.txt

3.5 About Autolog “types” ⁎

Notice that "type" is not a grammatical category defined in grammar `AUTOLOG.g4`. Rather, coherent types are inherently expressed via the association that is make by a judgement literal. In particular, regarding a judgement J having form `term1:term2`, the following conditions hold

1. J itself appear as a term in any Autolog rule (modified 9/24/2017)
2. Either `term1` or `term2` might be intended as a type
3. J can appear only at the rule literal level
4. J cannot appear negated
5. J provides an
   indexical association (`:`) between `term1` and `term2` for matching
   similar grounded judgement expressions (facts) on a branch:
   a. If J appears in the antecedent of a rule
      then J must match a judgement fact already on branch
      in order for the rule to be satisfied.
   b. If J appears in a disjunct of the consequent of a
      rule then J might be asserted to the branch case as a new fact
      on that branch

Item 1 means that judgements obout judgements are NOW syntactically allowed (judgements can be arguments for predicates generally). Item 2 means that any term can be associated with a type (term), including a type (term) associated with a type (term). In particular,types might be typed.Item 3 means that judgements are potential facts on branches and never arguments within facts.Item 4 means that judgements are never contrary to each other via negation, but they can be declared explicitly contrary, `X:int, X:bool => false`. Item 5 means that the semantics of judgements only involves whether or not an instance of a judgement might or might not appear on a branch of a Skolem Machine.

If an Autolog theory involves an intended type theory, the needed aspects of the type theory must be expressed by explicit rules (or modulators) of the Autolog theory.

The new Autolog grammar can be used in a predicative manner with tight distinctionbetween arguments and predicates and no explicit type contexts (and no judgements), as with the earlier colog versions.

The `@` operator is used to designate general arguments associated with universal quantification (as with previous colog grammars). As an example, the theory problem

$$(\forall X : t)p(X) .$$
$$(\forall X : t)[p(X) \rightarrow q(X)] .$$
$$(\forall X : t)q(X)?$$

could be translated to Autolog code as follows:

«
```
true => @t:t, p(@t).
X:t, p(X) => q(X).
q(@t) => goal.
```
»

`@t` represents an arbitrary element of type `t`. Previous quantified logic translators (to coherent logic) have employed this mechanism, and a new translators for quantified impredicative typed logic will do similar. Exercise: draw a proof tree for the translation.

3.6 About Autolog rewrite “modulators” ⁂

For a telling comparative example, let us consider the difference between the autolog modulator

«
```
A∧B=B∧A.      // inference 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 expressions*. To illustrate the difference suppose that we have only the one fact

```
true => (a∧b)∨c.
```

and the start of an inference tree

```
   true
     |
 (a∧b)∨c
     |
     ?
```

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

```
   true
     |
 (a∧b)∨c
     |
 (b∧a)∨c
```

obtained by matching the *left* side of the modulator in the tree term (a∧b)∨c, substituting a∧b by b∧a and asserting the tree term (b∧a)∨c. 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 tree inference is not possible. In order to use the rule to the same effect, one needs additional rules expressing equality of terms under substitution, which was the approach used for colog14I.

Obviously, modulators are more *active* than rules in such situations. If the present term modulator is included in our program, then the inference rule is redundant, since the modulator can of course act at the top fact level also.

An autolog modulator equality

L=***R***.

In intended to substitute instances of ***L*** (the left term) in a fact tree term by the corresponding instance of ***R*** (the right term), and not vice versa. In fact the modulator is implemented using a

`sam.machine.lang.Rewrite` component having `Term` attributes `.Lterm` and `.Rterm`. The corresponding active staged inference search component is `sam.machine.engine.ActiveRewrite.`

A match regimen for a modulator can be illustrated using another 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 such as this would, over successive stages, assert a complete branch set of modulants of facts from previous action stages.

Generally speaking, both modulators and definite rules use a level-saturation methodology (described more fully in the Autolog Chapter 5).

Strictly speaking, modulators should express equality terms ***L***=***R*** for which ***L*** and ***R*** would be interchangeable under any intended interpretation. BUT the modulator ***L***=***R*** only specifies that ***R*** be substituted for instances of ***L*** in branch fact expressions. In order to express symmetric substitution requires a different input expression for the modulator. Currently, the only way to do that is something like the following modification of the previous program

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

Of course this introduces much possible circularity of inference, which would be blocked by inference checks on repeated facts, the latter being quite a bit of extra computation in practice.

The idea of *paired* Rewrite expressions (e.g., a=b paired with b=a) might be employed to avert circularity: If fact F_2 resulted from earlier fact F_1 via one of a pair of rewrite instances, then block the reverse rewrite, rather than do the rewrite and then check for a repeated result. This idea is under investigation as a implementation design topic belonging in §5.9 (eventually). Avoiding circularity of inference is a very important implementation design topic having several open aspects, including the following aspect:

At present, the only way to index rewrites would be via the `Lterm` (using its operators or constants). Thus, for example, if we want to include the modulator

«

```
AvB=BvA.
```

»

(again under threat from cirularity) it should be the case the modulator makes sense for ANY instance A∨B that might occur on a proof tree branch.This means that variables A and B are "virtually indexed" by occurring in the Lterm of the modulator, rather than some "explicit index" such as that afforded by typing `A:T` and `B:T` for some T. We leave this as an open design issue at this time.

This section was drafted early in development for motivational impetus. And indeed the first passes at rewrite methods turned out to be inadequate.The discussion in this section has extensive conceptual and detailed design refinements in §5.5-5.8.

4 Autocode examples motivating design requirements ⋇

This chapter provides some ideas for using the expressiveness of the Autolog grammar, employing sample rules and modulators as program inference methods. The examples in this chapter are rather brief. It is important to keep in mind that the following sections are examples for expressions which are allowed by `AUTOLOG.g4`, and not necessarily examples of inference methods for good and effective theories. When AutoLog is implemented, one can better test how such axioms behave under the pressures of automated proof search.

The code examples in this chapter serve as motivation for how to implement and employ predicativity, general expression indexing, and indexicality. The implementation details are in Chapter .

4.1 Type checking ⋇

As is, the grammar allows one to write and parse illformed rules with regards to some *intentions* for typing. For example, if(i) below is intended then (ii) would be illformed:

```
«
   f:int->int->int, X:int, Y:int => f(X,Y):int.  % (i)
   f:int->int->int, X:int => f(X):int.           % (ii)
»
```

the problem being that f was intended as a function of 2 int arguments in the antecedent and then used as a function of 1 argument in the consequent of `(ii)`.

However, type terms are not necessarily given any meaning by the grammar itself. They are merely symbolic expressions. Using `(i)` can be said to impose the intended meaning, while `(ii)` would not. It is only when the terms are used in accordance with some convention for type meaning that one can expect well-formed typed Autolog rule expressions. This is a reasonable argument against insisting on type-checking: If you meant `(i)` then don't use `(ii)`.

The current thinking is to implement type checking as an (future) option for the new Autolog language `Reader` and to base the type checking on a set of rules abstractly specified in some manner. It would then be possible to load rules in accordance with a selectable (or no) type-checker.

Another issue is notation. If we had written `(ii)` like this

```
«
   f:int→int→int, X:int => f(X):int.
»
```

using → rather than ->, then the the other rules or modulators in the theory intending the same kind of typing should also systematically use → rather than ->.

Possibly, an interesting idea would be to design and implement type checkers whose input is autolog code and the checker inspects the code with regard to specifiable constraints on type terms.

4.2 Type terms and term complexity ⋇

Current Colog provers (e.g., colog14l) strategies which allow for blocking the application of *complex* rules. Complex rules are those having deeper term nesting in the arguments of consequent literals than in the antecedent literals. This would mean, for example, that the recursive rule in the theory

```
«
   true => p(0).
```

```
        p(X) => p(f(X)).
    »
```
could be blocked (cut) after a chosen number of applications. The cut strategies has been useful to achieve proofs from algebraic theories where term complexities can could rapidly grow if not limited.

However, using type-terms, consider the following similar recursive rule
```
    «
        P:T→prop, X:T, P(X) => P(f(X)).
    »
```
This rule would not be considered to be complex using the current algorithm: T→prop has depth 1 as does `f(X)`. Some modification of the complexity algorithm would be needed, perhaps to ignore type judgements.

4.3 Consistency rules ⁕

An interesting looking impredicative rule is
```
    «
        P:prop, P, ¬P => false.
    »
```
Note that the second literal factor is impredicative but indexical (indexed by a binding of the first factor), but the Third factor is predicative (functor = ¬) and factual (argument P bound).

For such a rule to trap inconsistent facts on the current branch would require that asserted be prop-typed, as in
```
    «
        X:T, p(X) => Y:T, q(X,Y), q(X,Y):prop.
    »
```
This might be a little burdensome to ensure. Alternately, the first literal factor of the consistency rule
```
    «
        ¬P, P => false.
    »
```
is predicative (functor=¬) and applies via the predicate index ¬ for branch facts without resorting to the prop-typing. Notice that the '¬' narrows the possible choices when the first factor is matched. Notice that if the current branch contains bot facts p(a)(b) and ¬p(a)(b) then the consistency rule applies with P replaced by `p(a)(b)`.

An alternate rule version
```
    «
        P, ¬P => false.
    »
```
does not narrow branch choices if the initial factor in the antecedent is matched first.

The `Autolog.g4` grammar does not require that '¬' be used for logical negation. One could use predicative rules such as
```
    «
        p(X), not_p(X) => false.
    »
```
where the intended negation is absorbed in the term name ‘not_p’. But such a rule would cover the case for consistency of a single predicate p, whereas the previous impredicative rule may cover all such consistency restrictions for facts on branch. Another (of many) general versions might be the following rule, which would impose a dynamic user operator (DUOP) for prefix negation:

«
```
P, 'not'P => false.
```
»
Which would display as "`P, notP => false.`" In the Autolog Viewer. Of course, it would be better for the programmer to use one notation for an intended logical negation or suffer the complications of making sure that different notations work together to effect the same meaning under interpretation.

4.4 Types of equality, equality types and extensionality ⁂

The Autolog grammar allows types of equality for contexts using various expressions …
«
```
A=B:T => A:T, B:T, A=B.
A:T, B:T, A=B => A=B:T.
```
»
As an exercise, let the reader write axioms for symmetric, reflexive,
and transitive properties of equality a) without types and b) for typed '='.

One can still write Autolog theories ignoring typing contexts or one can mix typed with untyped equality. Of course, it may not be a good idea to ignore typing if one intends to write typed coherent form theories.

Extensionality for terms can be expressed with some convenience, as with the
following rules:
«
```
F:T->S, X=Y:T => F(X)=F(Y):S.
P:T->prop, X=Y:T, P(X) => P(Y).
```
»
Extensionality for types and constructed types might be similarly expressed
«
```
T:type, S:type, T=S:type => pair(T,T)=pair(S,S).
X:T, T=S:type => X:S.
```
»

Note that the grammar allows to use equality in a type role, such as in the following example.
«
```
A:T, B:T, equiv(A,B) => P:A=B.
A:T, B:T, P:A=B => A=B:T, A=B.
```
»

One might say that `A=B:T` expresses a *type of equality* whereas `P:A=B` expresses a *witnessed equality type (or a proof)* P of A=B, and A=B expresses *a propositional fact*. We reiterate that the grammar allows many kinds of rule expressions that do not have a proscribed "meaning", but the rules that the logic programmer might write formal rules that reflect some intended interpretation.

It would be a good idea for the reader to experiment writing additional rules with equality, expressing whatever intended meanings may be of interest, expressed symbolically. This approach affords some inductive generation of equal terms (earlier branch equality proof terms lead to later branch equality terms). In this regard, one might find that the '=' operator restricts the expressiveness of equality axioms. Of use might be equality terms '`eq(-)`', which might have various formulations. For a preliminary example, consider
«

```
(* For a type T, eq(T) is an equality
   on T x T whose application is a type. *)
type(T), X:T, Y:T =>
eq(T):T->T→type,
eq(T)(X,Y):type.
% expand on this example
```
»

The expression eq(T)(X,Y) a parametric functor term sanctioned by grammar AUTOLOG.g4.

canonicity

It is easy to write Autolog rules that introduce terms whose computations get "stuck" because there may be missing additional rules which would "reduce" the terms. With regard to types, consider the following small example formulated with regard to a kind of "univalence axiom".

«
```
true => 0:boole, 1:boole.
% X:boole => X=0 | X=1.
true => p:boole→boole, p(0)=1, p(1)=0.
true => iso(boole,boole,p). % note
% "univalence axiom"
iso(S,T,P) => P:S=T.
% "compute or forcing axiom"
X:S, P:X=Y, P:X→Y, X:S => P(X):T, transport(P,X,P(X)).
%  compute p(0) via p:boole=boole
transport(p,0,A), A=1 => goal.
```
»

The problem here is that an isomorphism which transforms into an inhabited equality via a univalence axiom needs a term which computes the inhabitants of equal types based upon the isomorphism which gave rise to the equality.Otherwise the subsequent computations using the equality get "stuck" in the Skolem Machine. In general, this is a *canonicity problem* for types enhanced by axioms. In this example, the device to unstick the transfer was to let the isomorphism inhabit the equality using an additional forcing axiom. Although quite simple in the example, designing forcing axioms requires considerable effort. (I do not address automatic generation at this time.)

4.5 Kinds of logical negation ⁂

The Autolog.g4 grammar specifies ¬, ~ or - as builtin prefix operators, and we used ¬ as logical negation in §4.3 above. (We will restrict - as a symbol for an *algebraic additive inverse*.) Let us use ¬ as constructive logical negation (as in 4.3) and ~ as a kind of classical negation. How might this usage for symbols be reflected in autocode type expressions? First of all, the following autorule is conveniently motivated by considering Heyting interpretations of intuitionistic logic :

«
```
P:prop, ¬P => ~P.
```
»

If one considers Heyting topological interpretation for intuitionistic negation, this autocode rule is motivated by the constructed definition of ***¬S*** , for an open subset ***S*** of the real line: ***¬S = int(~S)***, where ***~S*** is the set complement of ***S*** and ***int(~S)*** is the interior (largest open *subset*) of ***S***.

Here are some more autocode rules , for constructive negation ¬ and classical negation ~

```
«
   P:prop => ¬P:prop.
   P:prop => ~P:prop.
   P:prop, ¬P => ~P.
   P:prop => P | ~P.  % LEM only re ~
   P:prop, P, ¬P => false.
   P:prop, P, ~P => false.
   P:prop, P, => ¬¬P.

   P:prop, P, => ~~P.    % a
   P:prop, ~~P, => P.    % b
»
```

The last two rules (a and b) might be replaced by the following modulator

```
«
   ~~P=P.   % M
»
```

The implementation of such a modulator involves several interesting design issues. We leave the implementation details for later, §5.8.

Note first that the last two rule methods are intended to work only at the *current fact level*, where the rule methods match current facts and infer new facts. The modulator M is supposed to work at any term level in any current branch fact. This rewrite amounts to a simplification or *reduction* of the fact.

For example, suppose that the branch of our Skolem machine has fact a∧~~b=c as a current fact. The modulator M applies and asserts fact a∧b=c to the current branch. This action is the Left-to-Right rewrite action associated with the modulator. This rewrite amounts to an *expansion* of the fact.

For a symmetrical example, suppose that the branch of our Skolem machine has fact a∧b=c as a current fact. The modulator M applies and assert fact a∧~~b=c to the current branch. This action is the Right-to-Left rewrite action associated with the modulator.

Now this replacing Left by Right by Left by Right … will be blocked by the basic mechanism which avoids asserting any fact already on the current branch. However, modulators can be blocked from the cycling using advanced indexing methods which will be explain in §5.8.

It is altogether possible to employ the rule methods a and b to achieve the same sub term replacement results, BUT additional rules regarding replacement rules for extensionality of equalities would be needed and included in our theory: That was the approach used with colog14l, for coherent predicative logic.

4.6 Universes ⋇

No effort has been made (so far) to devise any special universe mechanism that must be imposed on the new Autolog system.

Here are some sample rules that suggest how free one would be to specify type restrictions and type relations.

```
«
   % express that dom is a "universe"
```

```
    X:T =>  X:dom.  % T populated
    % express that inhabitation insures a type
    _:T =>  type(T).  % T populated
»
```
Similar examples can specify hierarchies of types. One question might be how to express that a type is a subtype of dom (say) without the type needing be populated (?) .

Types can be dependent.

```
«
    true => f:int->vec(int,int).           % type loose product
    N:int => f(N):vec(N,int).              % dependent
    true => g:int→pair(int,vec(int,int)).  % type loose sum
    N:int => g(N):pair(N,vec(N,int)).
»
```
Of interest here might be the issue of *naming* the dependent types.(?)

Notice the the judgement `g(N):pair(N,vec(N,int))` has variable `N` bound by the antecedent of the rule. The current grammar would allow to write rules that would have an existential variable in a consequent of the rule, but we cannot think of any good reason to employ that, nor do we know whether that possibility might spoil the decision question regarding what the intended interpretation of an impredicative type term would be.

There is however, an implicit type base and hierarchy determined by any given Autolog theory. The tree analysis and procedural semantics characterized in §3 and §4 of reference

https://skolemmachines.org/reports/SkolemMachines.pdf

still applies, in a general manner, to the extended theories described in this note. Several *structures* might be associated with an impredicative Autolog theory, depending upon the possible trees that can be generated. This issue requires further study.

In this Notebook, functor-form terms are used as type designators in judgements primarily because current programming language practice mostly uses that convention.That convention allows to specify the logic of terms via term arguments using coherent rules in coherent theories, again in an analogous manner to programming languages.

4.7 Formulating axioms for sets ⁎

Coherent set types can be specified with axioms in several ways. The following sample theory explores axioms for set membership in an unconventional manner. Rather than expressing set membership via a predicate term, e.g. X∊S, this formulation uses a judgement term X:S.

```
«
    % 1 meta --
       S:set(T), X:S => X:T.
          (* If S is a set of T and X
             belongs to S then X is a T *)
```

```
% 2 constructors --
  S:set(T), X:T => X:add(X,S).
     (* If S is a set of T and X
        is a T, then X belongs to
        the add(X,S) *)

  S:set(T), Y:T, X:S => X:add(Y,S).
     (* If S is a set of T, Y is a
        is a T, and X is in S then X belongs to
        add(Y,S)  *)

   % 3 consistency --
     S:set(T), X:S, empty(S) => false.
        (* If S is a set of T and X belongs to
        S then it is contrary that S be empty.
        This is an example of contrary not
        involving explicit negation. *)

% 4 equi-extent axioms for '='
   A=B, X:A => X:B.
   {X:A => X:B ?}, {X:A => X:B ?} => A=B.

% 5 declarations --
   true => s:set(int),
           int(1), int(2), int(3).
      (* some data, for example *)

   true => empty(nil).

% 6 query --
   2:add(3,add(2,add(1,nil))) => goal.
      (* a possible query: 2 belongs to the set *)
```

»

Membership types, such as expressed by the first axiom, use a judgement literal for membership, rather than a predicate literal, as in the formulation

«

```
% 1 meta --
   S:set(T), belongs(X,S) => X:T.
      (* If S is a set of T and X
         belongs to S then X is a T *)
```

»

Notice that judgements cannot be negated (see the grammar), but literal terms (predicates) can. A membership type resembles a dependent type: membership can depend on an element having been added to the set. An example of another way to specify a membership type is, a \emph{proposition-built set}.   For example, if P is a proposition defined for argument of type Tthen one can build the set $\{X:T | P(X)\}$. Adding the set builder notation to the grammar might be an interesting thingto do. Here is a formulation using the grammar as it is. The prop-built set is expressed as a term.

«

```
% 7 prop-built set --
```

```
        X:T, P:T->prop, P(X) => X:setOf(T,P).
        X:setOf(T,P) => X:T, P(X).
»
```

The second of the axioms under \%4 illustrates an issue of major importance for coherent logic: How to express logical equivalence using coherent axioms. This issue is continued in the next section, on Auto Lemmas. That issue aside (for the moment), the axioms express that if two coherent types are co inhabited then they are equal, which is an interesting logical conceptin its own right.

Exercise: Express Russel's membership paradox Russel's using the membership-types style of axioms from above.

collection terms

The `AUTOLOG.g4` grammar has a *collection term builder* that provides an alternate expression style for sets, collections and conglomerates. Some examples are illustrated as follows (3 rules and 4 rewrites).

```
«
   P:T→prop => {Z:T | P(Z)}:set(T).

   X∈{Z:T | p(Z)} => X:T, p(X).

   P:T→prop, Q:T→prop,
       X∈{Z:T | P(Z) v Q(Z)} => X:T, P(X) | X:T, Q(X).

   A∪B={X | (X∈A) v (X∈B)}.

   A∩B={X | (X∈A) ∧ (X∈B)}.

   ~A={X | ¬(X∈A)}.

   A\B = {X | (X∈A) ∧ ¬(X∈B)}.
»
```

Looking ahead to §4.11 on autolog meta-interpreters, various meta-interpreters involving the autolog collection builder, autolog rule meta-logic using parametric terms, and algebraic logic modulation would make good autolog programming exercises. (The examples illustrate here exemplify notational possibilities, but do not constitute a sufficient theory of anything.)

4.8 Auto lemmas ⁂

This section outlines the most difficult aspect of extending the Skolem Machine to compute for the new Autolog language. Getting this right will take careful specification and experimentation.

The issue intimately involves how to constructively represent logical equivalence as precondition for a coherent conclusion.

To continue the example from the previous section, reconsider the proposed axiom (and notice that the current Autolog grammar does not parse this.)

```
«
   {X:A => X:B ?}, {X:A => X:B ?} => A=B.
»
```

The *intention* of this is that if one could separately prove the two braced lemmas {… ?}, then one can conclude A=B.

One mode for lemma presumption would be that, whenever a lemma appears in an axiom, the context for the lemma is all of the other axioms which themselves do not reference lemmas is the antecedent. (This raises large difficulties involving dependence if more than one axiom of a coherent theory needs lemmas.) For now, assume that only one axiom in the theory needs lemmas.

So, assume that the Autolog theory (sequence of axioms) has the form T = N ∪ L where N is axioms requiring no lemmas and L is an axiom needing lemmas.

For the example at hand the needed lemma has the form {A => C?}which we presume to mean that N ∪ {true => A} ∪ {C => goal} should prove when *separately computed* by a Skolem Machine.

This preliminary formulation of coherent lemmas obviously needs more study. The idea of separate computation, in particular, requires careful specification, even though some ideas about it may seem fairly clear. There may be some use for *AutoLog scripts*: a meta language that expresses AutoLog theories and *problems*, which are automatically checked by a Skolem Machine. The motivation for this approach is the *verification methodology* of Coq.

4.9 Coherent modal impredicativity ⋇

An additional intended feature of the AUTOLOG.g4 grammar is to allow impredicativity suitable for extended coherent-form modal logic axioms.

For example, expressing some necessity and modal axioms, we might have write

```
«
   possible(P) => ~necessary(~P).
   P => necessary(P).
   necessary(P→Q), necessary(P) => necessary(Q).
»
```

or using Autolog built-in (prefix) symbols

```
«
   ◇P => ~□~P.
   P => □P.
   □(P→Q), □P => □Q.
»
```

Note that the antecedent of the second rule is completely impredicative.

4.10 Currying and indexicality ⋇

Autolog indexical profile specifications can involve *currying* technics (and type forms) for argument arrangements of parametric functors. (see https://en.wikipedia.org/wiki/Currying)

```
«
     P:T→(S→prop), X:T, Y:S => P(X)(Y):prop.
     P:(TxS)→prop, X:T, Y:S => P(X,Y):prop.
```

```
    F:T→(R→S), X:T, Y:R => F(X)(Y):S.
    F:(T×R)→S, X:T, Y:R => F(X,Y):S.

    true => f:int→int→int,   f(X)(Y)=X^Y.    // f(X)(Y):int
    true => f:(int×int)→int, f(X,Y)=X^Y.     // f(X,Y):int
```
»

Notice (2 middle rules) that the type notation `F:T→(R→S)` is associated with term-form notation `F(X)(Y):S` and that the type notation `F:(T×R)→S` is associated with the term-formnotation `F(X,Y):S`. These given specifications observe a logic/type convention, but this particular association is not an inherent Autolog system requirement that would be enforced by the Autolog parser or compiler. **It is an association specified by the programmer's rules.**

4.11 Parametric functors and data-type concepts ※

The `AUTOLOG.g4` grammar now parses functor expressions where the function name is instead a functor expression possibly with parameters (and the latter in a recursive manner). This allows the expression of several interesting mathematical constructs.

As an example, consider impredicative axioms regarding the category product of functors

«
```
    F:A->B, G:B->C => F∘G:A->C.
    F:A->B, G:B->C, X:A  => F∘G:A->C, (F∘G)(X):C.
```
»

There are many similar rules that can be expressed regarding category theory. Category commutativity diagrams are expressed using expressions formed with the extended coherent-form logic language `AUTOLOG.g4`.

As would be expected, the addition of functorial/parametric terms adds complications to the implementation of the abstract machine.

Another aspect of parametric functors is the issue of *concepts*, and predicates expressing concepts. As an example, consider the "concept of a list", and the corresponding type:

«
```
    L:list => list(L).
    list(L) => L:list.
```
»

The concept version is taken to be the predicte version of the type. If we were to express this for the "list of integer" concept, we might have

«
```
    L:list(int) => list(int)(L).
    list(int)(L) => L:list(int).
```
»

and this requires expressions containing parametric functors. Many interesting questions regarding axiomatic (algebraic) types have versions regarding the corresponding concepts.

The abstract theory of axiomatic categories, types and concepts awaits more exploration. There are some intriguing analogies between logical types and concepts enabled by impredicativity, e.g., coherent equivalences between concepts and types

«

```
X:P => P(X).
P(X) => X:P.
```
»

Next, consider two simple list versions. First a non-parametric version borrowed from the colog14(I) library.

«
```
// data & query
true => list(add(a,add(b,add(c,add(d,add(e,nil)))))).
hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),e) => goal.
% rules
dom(X), list(R) => list(add(X,L)). // add constructor
list(add(X,R)) =>
   hasItem(add(X,R),X), // has added item
   list(R),             // rest is a list
   tail(add(X,R),R).    // tail of add(X,R) is R
list(L), tail(L,R), hasItem(R,I) => hasItem(L,I).
// If tail of list has item, so does the list
```
»

The following `colog14I` proof was extracted from the search tree displayed on the next page

```
LEAF 27.
@26, rule2: hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),e) => goal
@25, rule4: list(add(a,add(b,add(c,add(d,add(e,nil)))))), tail(add(a,add(b,add(c,add(d,add(e,nil))))),add(b,add(c,add(d,add(e,nil))))),
hasItem(add(b,add(c,add(d,add(e,nil)))),e) => hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),e)
@24, rule4: list(add(b,add(c,add(d,add(e,nil))))), tail(add(b,add(c,add(d,add(e,nil)))),add(c,add(d,add(e,nil)))),
hasItem(add(c,add(d,add(e,nil))),e) => hasItem(add(b,add(c,add(d,add(e,nil)))),e)
@22, rule4: list(add(c,add(d,add(e,nil)))), tail(add(c,add(d,add(e,nil))),add(d,add(e,nil))), hasItem(add(d,add(e,nil)),e) =>
hasItem(add(c,add(d,add(e,nil))),e)
@19, rule4: list(add(d,add(e,nil))), tail(add(d,add(e,nil)),add(e,nil)), hasItem(add(e,nil),e) => hasItem(add(d,add(e,nil)),e)
@13, rule3: list(add(e,nil)) => hasItem(add(e,nil),e), list(nil), tail(add(e,nil),nil)
@10, rule3: list(add(d,add(e,nil))) => hasItem(add(d,add(e,nil)),d), list(add(e,nil)), tail(add(d,add(e,nil)),add(e,nil))
@7, rule3: list(add(c,add(d,add(e,nil)))) => hasItem(add(c,add(d,add(e,nil))),c), list(add(d,add(e,nil))),
tail(add(c,add(d,add(e,nil))),add(d,add(e,nil)))
@4, rule3: list(add(b,add(c,add(d,add(e,nil))))) => hasItem(add(b,add(c,add(d,add(e,nil)))),b), list(add(c,add(d,add(e,nil)))),
tail(add(b,add(c,add(d,add(e,nil)))),add(c,add(d,add(e,nil))))
@1, rule3: list(add(a,add(b,add(c,add(d,add(e,nil)))))) => hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),a),
list(add(b,add(c,add(d,add(e,nil))))), tail(add(a,add(b,add(c,add(d,add(e,nil))))),add(b,add(c,add(d,add(e,nil)))))
@0, rule1: true => list(add(a,add(b,add(c,add(d,add(e,nil))))))
```

```
true 0
list(add(a,add(b,add(c,add(d,add(e,nil)))))) 1
hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),a) 2
list(add(b,add(c,add(d,add(e,nil))))) 3
tail(add(a,add(b,add(c,add(d,add(e,nil))))),add(b,add(c,add(d,add(e,nil))))) 4
hasItem(add(b,add(c,add(d,add(e,nil)))),b) 5
list(add(c,add(d,add(e,nil)))) 6
tail(add(b,add(c,add(d,add(e,nil)))),add(c,add(d,add(e,nil)))) 7
hasItem(add(c,add(d,add(e,nil))),c) 8
list(add(d,add(e,nil))) 9
tail(add(c,add(d,add(e,nil))),add(d,add(e,nil))) 10
hasItem(add(d,add(e,nil)),d) 11
list(add(e,nil)) 12
tail(add(d,add(e,nil)),add(e,nil)) 13
hasItem(add(e,nil),e) 14
list(nil) 15
tail(add(e,nil),nil) 16
hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),b) 17
hasItem(add(b,add(c,add(d,add(e,nil)))),c) 18
hasItem(add(c,add(d,add(e,nil))),d) 19
hasItem(add(d,add(e,nil)),e) 20
hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),c) 21
hasItem(add(b,add(c,add(d,add(e,nil)))),d) 22
hasItem(add(c,add(d,add(e,nil))),e) 23
hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),d) 24
hasItem(add(b,add(c,add(d,add(e,nil)))),e) 25
hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),e) 26
goal 27
```

A second example, specified as an autolog program might be the following.

```
«
   // data & query
   true => add(a,add(b,add(c,add(d,add(e,nil))))):list(T).
   hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),e) => goal.
   % rules
   X:T, R:list(T) => add(X,R):list(T).
   add(X,R):list(T) =>
      hasItem(add(X,R),X), // add(X,..) hasItem X
      R:list(T),           // R is a list of T
      tail(add(X,R),R).    // tail of add(X,R) is R
   L:list(T), tail(L,R), hasItem(R,I) => hasItem(L,I).
   // If tail of list has item, so does the list
»
```

or, using parametric functors …

```
«
   // data & query
   true => add(a,add(b,add(c,add(d,add(e,nil))))):list(T).
   hasItem(add(a,add(b,add(c,add(d,add(e,nil))))),e) => goal.
   % rules
   X:T, list(T)(R) => list(T)(add(X,R)).
   list(T)(add(X,R)) =>
      hasItem(add(X,R),X), // add(X,..) hasItem X
      list(T)(R),          // R is a list of T
      tail(add(X,R),R).    // tail of add(X,R) is R
   list(T)(L), tail(L,R), hasItem(R,I) => hasItem(L,I).
   // If tail of list has item, so does the list
»
```

pair terms (and sequences)

The AUTOLOG.g4 grammar now has a *pair term builder* that can be used to construct pair data types for autolog programs. Consider, for example the following autolog rules.

```
«
   A:T, B:T => (A,B):pair(T).    // type
   true => 1:nat, 2:nat.
   (1,2):pair(nat) => goal.

   A:T, B:T => pair(T)(A,B).                // WARNING
   A:T, B:T, pair(T)(A,B) => (A,B):pair(T).  // AMELIORATION
»
```

The WARNING rule is not advisable because `pair(T)(A,B)` will be parsed so that `(A,B)` constitutes an internal argument array for the functor `pair(T)`. However the AMELIORATION rule would in effect convert the argument list to a pair term. (That is conceptually awkward …)

The pair constructor for terms can be employed to define sequences, as in following sample code.

```
«
   A:T => (A,nil):seq(T).
   A:T, S:seq(T) => (A,S):seq(T).

   true => a:t, b:t, c:t.
   (a,(b,(c,nil))):seq(T) => goal.
»
```

Why would the parametric version of `seq` cause confusion (as it was for pair)?

4.12 Logic theories and meta-interpreters in Autolog ⁎

The new Autolog language allows many possibilities for expressing meta-logic rules, generally understood. For example, using logic terms as literals

```
«
   P, P→Q => Q.
»
```

The way that this rule is written, the first literal factor in not indexical. But if one reorders the antecedent literals for satisfaction against branch facts using the *second* literal literal term first,

```
«
   P→Q, P  => Q.
»
```

Then the literal P is indexed matching the the predicatice first literal (predicate functor →). Either order could be used (with different search/match profiles), but the second version would tend to be more efficient.

For another example, meta-logical equivalence has nice extended coherent-form renderings, such as

```
«
   P↔Q => P→Q, Q→P.
»
```

or (when type terms would be useful):

```
«
  P:prop, Q:prop, P↔Q => P→Q, Q→P.
»
```

Using such rules as axioms for coherent algebraic logic is currently under study. An especially interesting idea is to formulate equational algebraic logic and then use equalities to modulate coherent meta-logic literals via rewrite mechanisms.

Formulating rules governing quantifiers is also interesting. Various approaches can be employed. For example, consider

```
«
   true => ∀(X:T,p(X)).
```

```
    ∀(X:T,P(X)), Z:T => P(Z).
»
```

The first rule is an approximation to a coherent axiom for $(\forall x : t)p(x)$.The second rule is an instruction for the procedural semantics for applying the axiom.The reader is invited to provide the details regarding how the two rules will work together to provide correct inferences -- a somewhat subtle issue.

Similarly, consider also

```
«
    true => ∃(X:T,p(X)).
    ∃(X:T,p(X)) =>, Z:T, P(Z).
»
```

A related approach regarding *natural types* is illustrated with the following natural language example. Reasonable parses for the two sentences

Every applicant brought a portfolio.
Every applicant brought the same portfolio.

Judge the first sentence to be an $\forall\exists$ context, and the second to be an $\exists\forall$ natural language context. The *logical semantics* for the two sentences call for distinct logic translations. As assertions, a translator might provide the following extended coherent forms (LaTeX equations):

$$\top \rightarrow \texttt{brought1} : \forall(\exists) \wedge \texttt{brought1}(\forall(\texttt{salesagent}), \exists(\texttt{portfolio})) \, .$$
$$\texttt{P} : \forall(\exists) \wedge \texttt{P}(\forall(\texttt{G}), \exists(\texttt{K})) \wedge \texttt{X} : \texttt{G} \rightarrow \texttt{Y} : \texttt{K}, \texttt{P}(\texttt{X}, \texttt{Y}) \, .$$

An expression such as $\texttt{brought1} : \forall(\exists)$ indicates a natural type for the predicate, which governs the rule behavior for the translation.

Now, contrast the translation for the other verb type, $\exists\forall$,

$$\top \rightarrow \texttt{brought2} : \exists(\forall) \wedge \texttt{brought2}(\forall(\texttt{salesagent}) \wedge \exists(\texttt{portfolio})) \, .$$
$$\texttt{P} : \exists(\forall) \wedge \texttt{P}(\forall(\texttt{G}), \exists(\texttt{K})) \wedge \texttt{X} : \texttt{G} \rightarrow \texttt{Y} : \texttt{K}, \texttt{P}(\texttt{X}, \texttt{Y}) \, .$$

$\texttt{brought1}$ and $\texttt{brought2}$ are variations on the stem verb. We do not here suggest a uniform mechanism for how to handle dictionary forms for natural types. As an exercise, translate the equations into Autolog expressions using the Autolog Editor and check that they compile.

4.13 Online Modulator and metalogic examples ⨳

An online collection of **Lecture Notes, Autolog Meta Programming**
Is available at

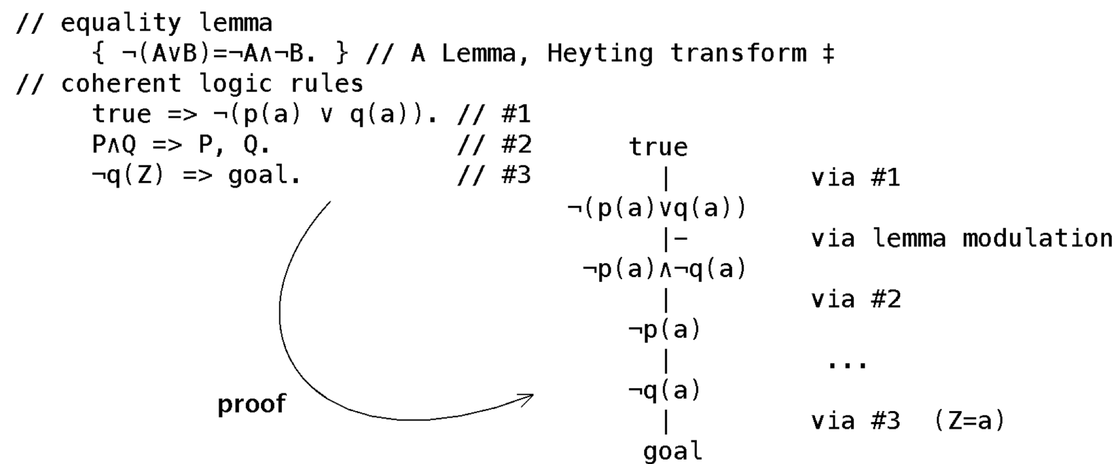
https://skolemmachines.org/
(Scroll to bottom of target webpage)

Additional lecture notes are added from time to time.

The following images show three of the presentation slides (#12, 14, 15) from

Autolog Equality Inference, *BLAST18*, University of Denver ↬ **slides**.

12– AutoLog example A: modulated proof terms



Proof terms can be algebraic expressions modulated (substituted) by equality lemmas (which could be weighted towards a preferred subsitution). The "bigger plan" here is to devise mechanisms for "universal logic computations", implemented using a Skolem Machine. This example illustrates how we might incorporate constructive negation into our logic computations.

14– AutoLog example B: more metalogic

```
// rule has types, impredicativity, indexicality
   P:T→prop, Q:T→prop, X:T, P(X) => P(X)∨Q(X).  // #1
// coherent reification of ∨
   P ∨ Q => P | Q.                              // #2

true => p:int→prop, q:int→prop, a:int.       // data
p(a) => goal.
q(a) => false.
```

A => A ∨ B.

will not work
as intended for #1

proof

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

15– AutoLog example C: ∀, ∃ metalogic

```
P:T→prop, ∀(X:T,P(X)), Z:T => P(Z).  // #1

P:T→prop, ∃(X:T,P(X)) => X:T, P(X).  // #2

// -----
true => p:int→prop, ∃(X:int,p(X)).   // data
p(Z) => goal.
```

proof

```
       true
        |-           data
    p:int→prop
        |
   ∃(n:int,p(n))
        |-           rule #2
      n:int
        |
       p(n)
        |-           Z=n
       goal
```

5 Autolog Implementation design aspects ⋇

This Chapter describes structures and algorithms for implementing the AutoLog search engine. The last public distribution of the mother prover was colog14l, which has had only minor (but many) modifications or corrections since 2014.

Some of new language features of AutoLog date from about 2013 and have been refine over time since then, and tested in prototype, but were never added to the public colog14l.jar. The new aspects of the language -- such as impredicativity, type expressions, detection of indexicality, equality modulators, concurrent inference mechanisms -- require many intricate changes to the implementation designs for colog14l. However, AutoLog borrows many ideas from the colog14l implementation, with refinements of detail in many cases. The borrowed aspects of the Skolem Abstract Machine are included here in updated form, for emphasis and motivation for the expanded implementation designs.

The major tool for testing implementation design is the `AutoLogAnalyzer`, located as follows: `sam.machine.lang.AutoLogAnalizer.java`, which is used to compute the structural parameters of rules and modulators (`sam.machine.lang` components). The corresponding active inference objects (`sam.machine.search` components) are designed using methods which put into effect the interded inference mechanisms for the rules and modulators. An analyzer tool was used extensively for colog14l. The AutoLogAnalyzer tool reports in the "aux" pane of the AutoLog Viewer.

5.1 Branch fact indexing and tableing ⋇

A Skolem abstract machine for autolog builds fact trees whose structure is determined by rules or modulators.

In the first subsection we will review how colog14l generated colog fact trees for coherent colog theories having only predicative literal factors in the rules. Predicative autolog largely concides with the colog language. A predicative literal factor has a \emph{bound} predicate for its literal. The key concept regarding predicative indexing is that the literal predicate is an essential part of an efficient hash key for locating branch facts that might match a literal expression.

In the second subsection we review tabular indexing for the predicative theories computed by colog14l.

adding new facts to branch for predicative rule theories

A predicative coherent logic theory is a finite ordered sequence of predicative rules. This subsection describes how such rules are used to expand a search branch for a Skolem Machine implementing the predicative theory. We describe structures to hold facts on the branch and a bindings array for each rule.

To review the basic concepts, let us reconsider a simple example from the introduction of

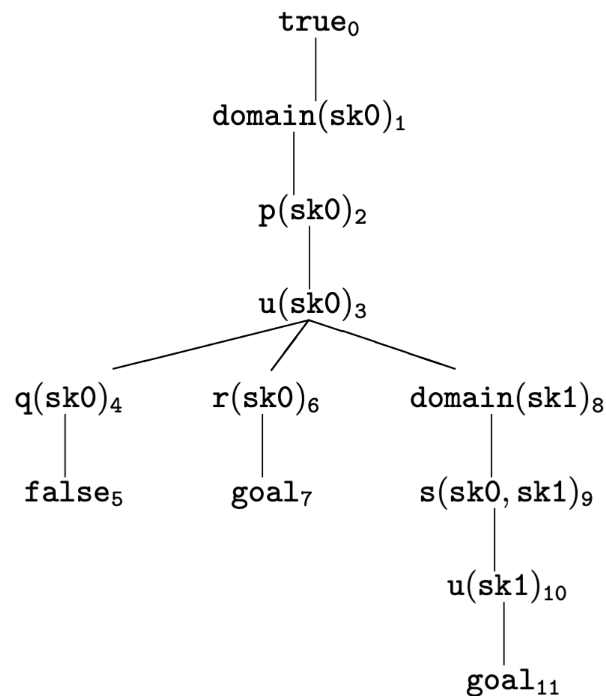
https://skolemmachines.org/reports/SkolemMachines.pdf

The following colog theory has six predicative coherent logic rules. The free variables in the consequent of rule #1 and one of the disjuncts of rule #2 are existential variables. When such a rule is satisfied and fires it introduces a new Skolem constant as an eigenvalue for the existential variable. The '|' in the consequent of rule #2 is disjunction. Conjunctions of atoms are separated with commas.

```
«
   // Example 1
   true => domain(X), p(X).                    % #1
```

```
    p(X) => q(X) | r(X) | domain(Y), s(X,Y).  % #2
    domain(X) => u(X).                        % #3
    u(X), q(X) => false.                      % #4
    r(X) => goal.                             % #5
    s(X,Y) => goal.                           % #6
»
```

The following figure shows a proof (generated by colog14l) for the little theory above. The \LaTeX~ source code for search tree pictures were automatically generated by QDF colog prover (e.g., colog14l or related).



Proof for Example 1

The LaTeX code for the figure was also automatically generated from the proof.

A old prover used a depth-first search algorithm and first saturates the leftmost branch and then backtracks to the branch point (3), saturates the second branch, backtracks to the branch point (3) again, and then saturates the final branch.

Serial implementations use just one active branch. When the machine backtracks, it then reuses the portion of the branch below the branch point and writes over the previous facts left over from the last branch probe. Branches are arrays of facts (`Fact` objects). A depiction of the pictured tree using a single branch might look like this:

```
  0       1          2       3      4            5           6      7
 [true,domian(sk0),p(ski),u(sk0),q(ski),      false                        ...]
 [                                ,r(sk0),       goal                       ...]
 [                                ,domain(sk1),s(sk0,sk1),u(sk1),goal,  ...]
```

The functors of a colog theory (predicates and functions) are themselves indexed by name/arity. For the theory of Example 1 the predicate indices are

```
    0 : true
```

```
1 : goal
2 : false
3 : domain
4 : p
5 : q
6 : r
7 : s
8 : u
```

The positions of current facts on the branches is managed by
three arrays

```
first[-] of size equal to number of predicate names

last[-]  of size equal to number of predicate names

jumpto[-] of size equal to size of branch
```

These arrays are updated by rule applications as facts are added to the branch, and the arrays are appropriately reset when control returns to a branch point. In particular, reconsider the third branch ***B*** of the tree above at the time that it saturates:

```
0      1           2       3         4           5      6     7
[true,domain(sk0),p(ski),u(sk0),domain(sk1),s(sk0,sk1),u(sk1),goal,...]

   first  = [0,7,*,1,2,*,*,5,3]
   last   = [0,7,*,6,2,*,*,5,3]
      jumpto[0]=*
      jumpto[1]=4
      jumpto[2]=*
      jumpto[3]=6
      jumpto[4]=*
      jumpto[5]=*
      jumpto[6]=*
      jumpto[7]=*      ...
```

The `*` indicates a sentinal value (say the size of the branch ≈ "off the branch") indicating that there is *no relevant* value. So, for a particular example, the first occurrence of `domain/1` (index=3) is located at position first[3]=1 on ***B***, and the next is located at position `jumpto[first[3]] = jumpto[1] = 4 =last[1]`, and so there are no more choices to jump to.This illustrates a basic calculation mechanism for predicate position indexing on ***B***. (This method does not explicitly employ hash tables, but it does mimic the indexing of a has table.)

This basic indexing is enhanced by adding other position information (distributive choices, tabular indexing) to ensure fairness of antecedent matching, avoid re-satisfaction of antecedent factors, and indexing speed .

A rule is *applicable* at the focus point on the branch provided that its antecedent conjunction of factors matches facts at or above the focus point and that its consequent (disjunction of conjunctive factors) is notalready satisfied using facts at or above the focus point.

Active rules are objects compiled from colog rules. Part of the dynamic data of an active rule is a binding table that is used when attempts are made to match rule factors (antecedent or consequent) against branch facts. The binding table records variable matchings in a factor by remembering which ground term is bound to a variable.

To illustrate how a binding table works consider another small colog theory:

```
«
   // Example 2
   true => p(a), p(b).
   p(X), p(Y) => q(X,Y,Z), p(Z).  // existential rule
   q(b,A,B) => goal.
»
```

Referring to search tree in Figure \ref{figure:x1.2} at branch
index 6, the second rule can be satisfied and produce the inference whichs asserts the facts
at positions 7 and 8.

A *variable binding table* is a matrix of ground functors indexed on the left by variables (which are indexed using nonnegative integers) and indexed across the top by nonnegative integers representing the predicate factors in a rule. The variable binding table is part of the structure of the `sam.infer.ActiveRule` wrapper component for a `sam.lang.Rule` component.

To illustrate, consider the second rule of Example 2. This rule has three variables, 2 antecedent factors (index 0 and 1) and two consequent factors (index 2 and 3).The bindings in the table are shown in the following table

factor	0	1	2	3
X/0	b	-	-	-
Y/1	-	a	-	-
Z/2	-	-	sk2	-
choice	2	1		

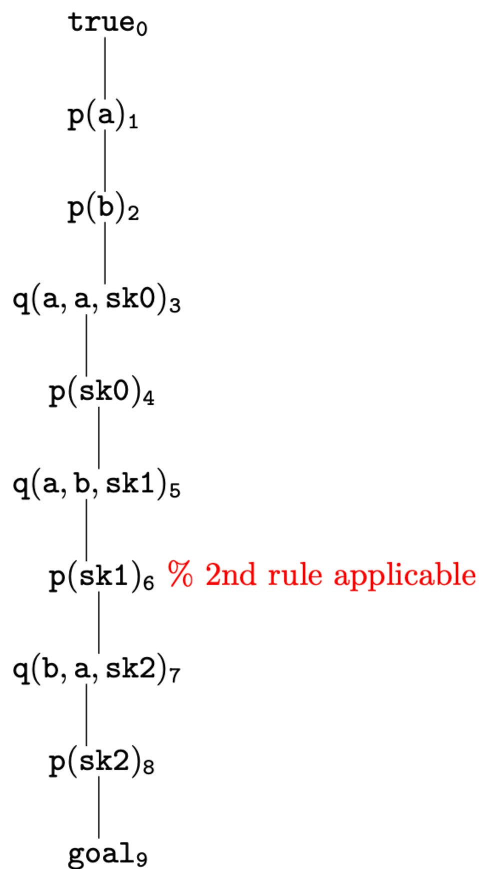
Variable binding table for 2nd rule of Example 2

The inference in question (and the factors) is

```
0      1           2          3
p(b), p(a) => q(b,a,sk2), p(sk2).
```

The table shows what the antecedent bindings were matching the antecedent, and the branch position choice that produced that match. The consequent was not already satisfied for those choices, so a binding of `sk2` was created for `Z` in the third factor, and these bindings were used to generate the facts asserted at branch positions 7 and 8.

These details do not explain *why* the particular choices for matching on the branch were made. That involves a distributed choice mechanism explained in §5.3. §5.2 explains why a fair choice mechanism is needed, of which the distributive choice mechanism is an excellent theoretical specification.



Proof for Example 2

predicative coherent tabular indexing

This subsection reviews methodologies from colog14l regarding tableing methods for predicative coherent logic. This is intended as additional motivation for generalizations to autolog designs.

When the colog reader reads a colog rule it analyses the rule by looking at all of the factors in the antecedent and consequent in left-to-right order. In order to fix ideas let us work with a specific example of a rule. Since we are reviewing predicative coherent logic, all functors are predicative so indexical via predicate and function **names**.

Additional indexicality using arguments requires more analysis. The idea here is to compute the patterns of bound variables in rule factors as they are encountered left-to-right. These are called *key patterns*. These key patterns can be computed using static analysis of the input theory when it is loaded. Here is a concrete example.

```
  0     1      2      3          0      1       0
p(X), q(Y), r(X,W), r(W,Y)  =>  s(Z), t(X,Z) | w(X,Z).
 [f]   [f]   [t,f]   [t,t]       [f]   [t,t]   [t,t]
```

where `t=true` and `f=false`. So, in particular, thethe 2-factor of the antecedent `r(W,Y)` has `X` bound and `W` unbound when an attempt is made by the rule to match this factor. That is, the variable binding table will have a value stored for X but not for W at that time, as outline in the discussion of the *variable binding table* in the previous subsection (using, of course, a table appropriate for the present rule). The key patterns for a factor are stored in the active factor components as boolean arrays.

Note that the key patterns can also be used to compute a *match order* for factors. Some efficiency is gained by attempting to match bound variables first, in order to allow fast-fail and also so as to limit the possibilities for branch facts to actually match the factor.

A factor is said to be *factual* provided that all of its variables are bound in the rule binding table at match time. A factor is said to be *subfactual* provided that at least one, but not all, of its variables are bound in the rule binding table at match time. If a factor is neither factual nor sub factual, then it is a **free** factor.

For colog14I, if every antecedent literal factor of a rule is a free factor, then an attempt to satisfy the antecedent of the rule uses the basic branch indexing (distributive choices with `jumpto[-]` as outlined in above. The component `sam.infer.ActiveDCRule` (extension of `ActiveRule`) is employed to satisfy such a rule.

On the other hand, if every antecedent literal factor of a rule is factual or subfactual, the *branch fact hashtable* is employed to satisfy all of the antecendent literal factors of the rule. The component `sam.infer.ActiveTabDCRule` (extension of `ActiveRule`) is employed to satisfy this kind of a rule.

For Autolog, an `ActiveFactor` component is planned as a wrapper for each literal in order to effect a matching process suitable for each rule literal on a literal-by-literal basis, depending upon the analysis of the indexicality aspects of the particular literal.

Checking to see if a consequent conjunction is already satisfied uses either the fact hash table for bound consequent factors or a stepwise approach for existential factors.

The branch fact hashtable is the essential device for tabular indexing. The table is a custom hash table which stores branch positions (indices) of facts on the branch which have hash values associated with key pattern for a rule factor. For Autolog with `ActiveFactor` components, it is anticipated that more elaborate hashing techniques will be deployed.

For colog14I, given a rule factor F (`Functor` object), the table bucket holding F is retrieved as follows

```
bucket[factor] = tree.table.get(keyHashCode(f))
```

which returns a bucket list of positions of facts on the current search tree branch table which have the same `keyHashCode` as the factor F,
and then linear search of this list attempts to find an exact computed match of the rule factor to some branch fact. This scenario is typical for table probe mechanisms.

Here is the Java source code for the factor `keyHashCode(-)` calculations:

```
protected int keyHashCode(Functor f) {
   int h = f.index ;
   for (int i = 0 ; i < f.arity ; i++)
      h = (h<<5) + (f.key[i] ? hashCode(f.args[i]) : 0) ;
   return h ;
}




/**
  *  @return factual hash value of terms using current table bindings.
  *  This call is only made for BOUND Terms.
```

```
   */
  public int hashCode(Term t) {
     int hc = 0 ;
     if (t.isVariable()) {
        int v = ((Variable)t).index ; // which variable?
        for(int c = 0 ; c < C ; c++)
           if (table[v][c] != null)
              hc= Fact.hashcode(table[v][c]) ;
           // N.B. variable must be bound
     }
     else {  //t is a Functor
        hc = ((Functor)t).index ;
        for (int i = 0 ; i < ((Functor)t).arity ; i++) {
           int h = hashCode(((Functor)t).args[i]) ;
           if (h == 0) return 0 ; // unbound arg   ???
           hc = (hc << 5)  + h ;
        }
     }
     return hc ;
  }
```

Although the other details may be sketchy, the essential calculation is the ubiquitous *shift-left 5 dictionary hash code calculation*. Experiments show that this hash function is reasonably diverse, but there are probably sharper functions that could be explored. Table entries are further distinguished as to whether entries in the buckets contain a factual or a subfactual entry. The table is populated when facts are asserted to branch and the table is used to retrieve possible matching branch positions when rule factors are matched against branch facts. Possible matches retrieved by hash code must then actually match, and this may cause new rule bindings for later factors. Of course, one needs to make some allowance for when the shift hash code overflows the return type.

It is important to emphasize that tableing described here affords an efficiency improvement for the distributive choice algorithm which will be formally described in §5.3, but was exemplified in this section using `first`, `last` and `jumpto` (and where branch indexing used only predicate names and arity).

One uses a hash table to store branch positions that hold facts that entangle a more restrictive hash value (e.g., name, arity and combined argument hash values). The more restrictive hash functions eliminate failing choices that would otherwise entangle a factor using a more generous hash value (name, arity). So, more restrictive entanglement profiles eliminate many choices that were bound to fail anyway.

We assume that tabeling uses buckets where branch choices are linearly ordered in a way that if choice `a=branch[i]` is retrieved before choice `b=branch[j]` then `i<j`, and vice versa. In this way, an earliest-first inference regimen would be imposed, which makes run-time inference testing or debugging more intuitive.

The GUI version of the colog14I prover allows inspection of the branch fact table (under "options" menu).

In §5.6, we will extend the colog14I tabling methods describe in this section for autolog rules and modulators. The extended approach will be called *Multihash Branch Term Indexing* (**MBTI**). We will also provide comparisons of autolog **MBTI** methods with historical term indexing/retrieval methods in §5.6.

5.2 Fairness ⋇

An inference search algorithm is *fair* provided that the methods it uses to apply rules and equality modulators does not hide, miss or avoid any possible matching choices, either for the antecedent of a rule or the left side of an equality rewrite. A *hidden* matching choice is one which never occurs no matter how long the search algorithm processes rule applications on a branch.

Inference fairness is related to the *completeness* of the inference system, from the logical point of view.

We can illustrate this for rules by reconsidering Example 2 from §5.1

«
```
// Example 2
true => p(a), p(b).
p(X), p(Y) => q(X,Y,Z), p(Z).  // existential rule
q(b,A,B) => goal.
```
»

The relevant issue (for this particular example) is *how* the choices are determined for matching the antecedent of the second rule.

Let us first investigate an unfair regimen for applying the second rule.
Suppose that the branch expands as follows …

```
true
  |
p(a)
  |
p(b)
  |
q(a,a,sk0)
  |
p(sk0)
  |
q(a,b,sk1)
  |
p(sk1)
  |
q(a,sk0,sk2)
  |
p(sk2)
  |
q(a,sk1,sk13)
  |
p(sk3)
  |
q(a,sk2,sk4)
  |
p(sk4)
  |
q(a,sk3,sk4)
  |
p(sk4)
... forever
```

The second rule is being applied to each new possible match binding for Y, ignoring the fact that X might make new matches. This resembles a control order similar to nested for loops

```
for all X
   for all Y
       ....
```

where the inner loop on Y continues before the outer X loop is revisited for new values. The relevant issue is that X never *sees* b because Y continues to have new matches.

This is, of course, not the only manner in which unfairness could arise using an antecedent rule choice mechanism. It is only a simple concrete illustration. Early versions of the colog prover had this fault. Similar examples could be used to illustrate missed matches for equality modulators.

A general solution requires mechanisms for achieving all relevant matches in a current stage of inference before overall advancement to a next stage.

In the next section, we describe a distributive choice algorithm for rule antecedents that is part of a combined staged approach to fairness for rule application. A nice advantage for the distributive choices algorithm is that, correctly employed, it can neatly avoid retrying previous choices that would already have been attempted!

Fairness of modulator applications is considered in more detail later in this chapter, §5.9.

5.3 Distributive choice algorithm ⁎

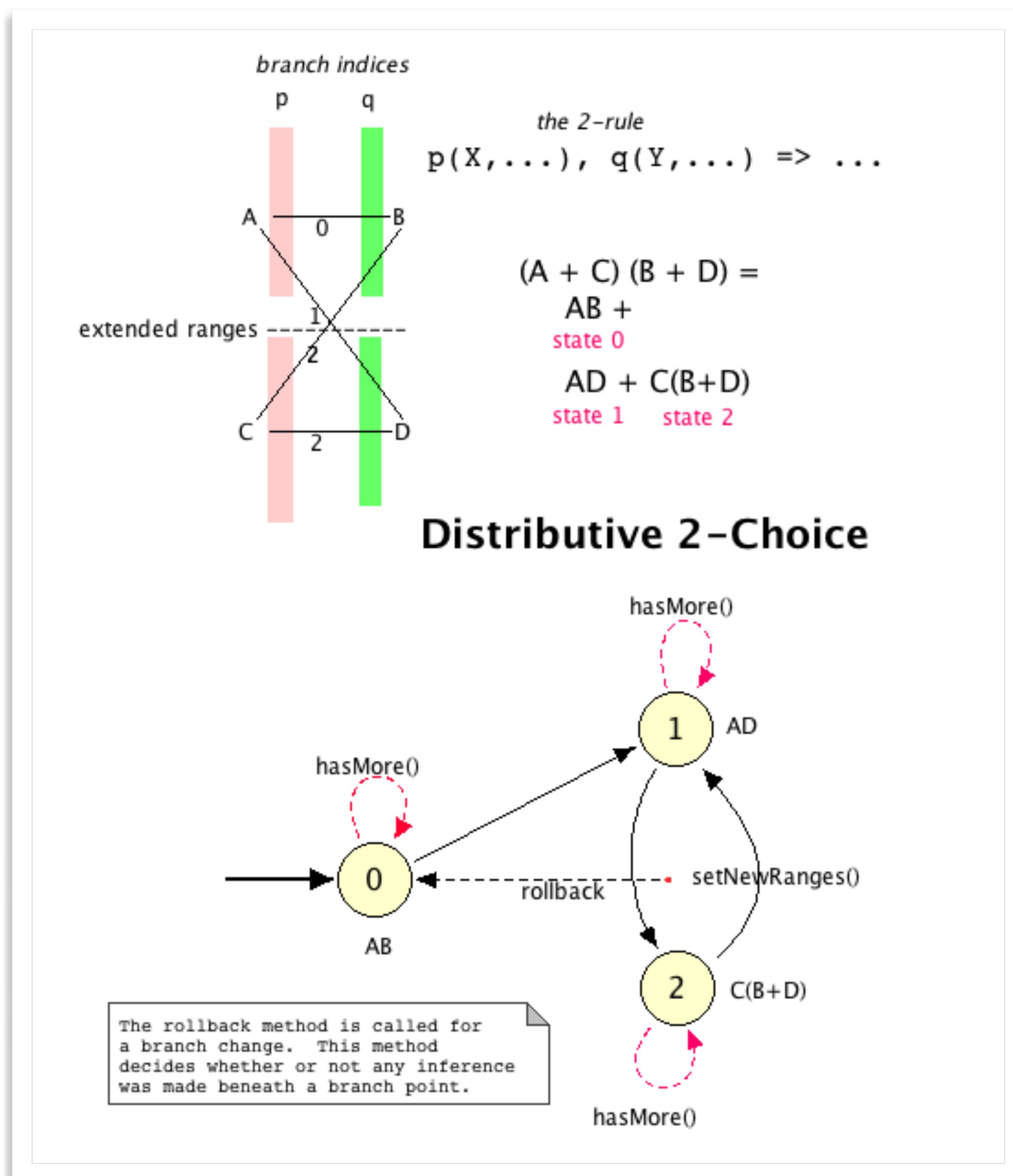
Distributive choices, or DC for short, is an algorithm for selecting facts on the current branch which satisfy the antecedent literal factors of a logic rule. We motivate the algorithm by descriptions first for rules with two antecedent factors in the rule, then three, etc. Each case, dependent on the number of antecedent factors, gives rise to a finite state machine that marshals the matching choices for the rule.

An effective implementation then compiles autolog rules into active components that correctly compute the finite-state machines.

The DC approach is superimposed upon other detailed methods for deciding specifically what facts on the current branch are appropriate matches for a rule literal. The DC approach specifically monitors only the *choice selections* in a fair manner.

2-choices

The following figure depicts choice ranges for a rule with two antecedents. In the figure the ranges for choices for p are labeled A and C, and the ranges for q are labeled B and D. The C and D ranges depict *extended* ranges. An expression like AB stands for all choice pairs `[i,j]` where `i` falls in range A and `j` falls in range B. The iteration of choice within ranges can be effected either using the indexing arrays `first[-]`, `jumpto[-]`, and `last[-]` for the predicate literals or equivalent choice tabling methods (both approaches from §5.1).



The terminology *distributive choice* ranges draws on an analogy with the distributive law formula.

$$(A+C)(B+D) = AB + AD + C(B+D)$$

where the range (A+C) signifies A together with C, for example. The formula can be read to mean that, after considering the initial choice range AB, and then extending with C snd D, compute the new choices in a manner indicated by the expressionsAD and then C(B+D). Notice that the ranges AB, AD, and C(B+D) are mutually exclusive, and thus the initial or prior range AB is not revisited when considering the extended ranges, and neither does either extended range AD or C(B+D) impinge on the other.

Conventional backtracking algorithms easily apply to give complete iteration of all possible choices within the subranges AB, AD and C(B+D). *If* new range choices *were* dynamically allowed -- rather than delayed for later DC state -- conventional backtracking algorithms fail to be *fair*: the last item for iteration blocks later items for earlier choices as explained in §5.2. The distributive choices approach, together with saturation of ranges within a distributed choice state, corrects this fairness problem and it also

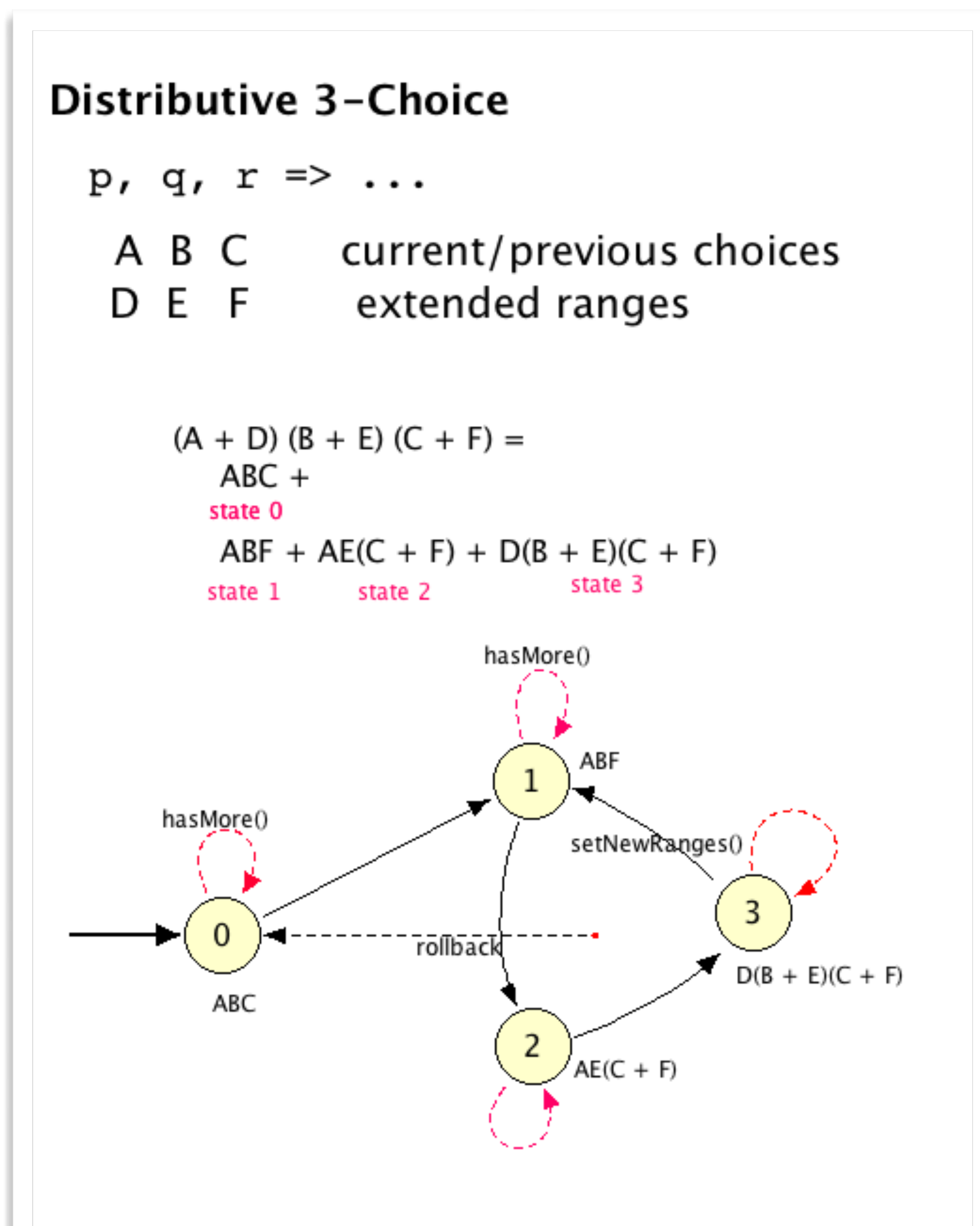
speeds up new inferences because it obviates the need for reconsideration of previous choice ranges. Also, in-state saturation of ranges can easily be designed to not repeat those specific choices.

The 2-distribution formula applies similarly for each subsequent extension to the ranges, after the prior range choices are exhausted. The distribution formula is the basis for a finite state machine also depicted in figure for a 2-choice distribution. The start state is state 0 (range AB). After depleting choices in the range of state 0, transition to state 1 (range AD). After depleting choices in the range of state 2, transition to state 2 (range C(B+D)). The states 1 and 2 then constitute a super-cycle for additional range extensions. One can demonstrate that a subsequent extension E and F distributes in the same 2-pattern as that already considered: (A+C+E)(B+D+F)=(A+C)(B+D)+… Thus, the distributive choice algorithm is dynamically stable as choice ranges extend while the search branch extends.

The 1+2 state machine thus becomes the basis for the compilation of a 2-rule into a process that applies the rule choices on a branch in accordance with the range choices determined by the state machine. The *rollback* transition corresponds to the control adjustments that need to be made when a delayed branch is considered. This depends upon whether the choices are serially considered (rollback must check ranges on the previous branch) or concurrently (rollback does nothing but a new branch is spawned).

3-choices

For further motivation to establish the general DC pattern, consider the case of a 3-rule, a rule having three antecedent factors. The state-machine picture is depicted in yer following figure.



In this case, the corresponding state machine has 1+3 states, and a 3-rule is compiled to a process that applies the rule choices on a branch in accordance with the range choices determined by the state machine.

4-choices

Four an autolog rule having 4 antecedent choices, consider the following figure illustrating the sequence of 4-choices. The red bars indicate *active* range choices for the state.

4-pattern for Distributive Choices

A B C D

<-- current/previous ranges

the situation

<-- next ranges

E F G H

state 0 ABCD

state 1 ABCH

state 2 ABG(D+H)

state 3 AF(C+G)(D+H)

state 4 E(B+F)(C+G)(D+H)

The general algorithm is to consider an autolog rule having N factors in its antecedent. The search algorithm is modified so that the rule applications are restricted to a current range-choice state until all of those choices have been tried, and then advance to the next state. When all states have ben visited, the

choice ranges are enlarged in state 0 to include new branch facts that rules have asserted to the branch since the previous visit to state 0.

A distributed choice rule is inherently *fair*: Given *any* possible antecedent choice on a branch, there is some distributed range that includes the given choice. However, distributed choices do not necessarily produce *earliest-first* choices (which was involved in an earlier algorithm that was not nearly so fast as the distributive choices algorithm,.

An interesting statistic for measuring the efficiency of an implementation is a *ratio of effective inferences*, defined as

$$\rho = \texttt{\#effective inferences} \ / \ \texttt{\#attempted inferences}$$

An *attempted* inference occurs when the rule choices do match the antecedent of a rule. An *effective* inference occurs only when the consequence is new (not subsumed on a branch by previous facts), in which case the inference adds new facts to the branch. If no rule ever revisits previous choices (as with DC) then ρ measures some kind of maximal effectiveness, which is a property of the autolog theory itself. Keep in mind that a rule can easily produce repeated consequences for many different choices of antecedent factors, so we would seldom have $\rho = 1$.

branch switching

Suppose that R is an active rule of the autolog theory. A critical computation is one that determines which choice ranges for R are safe to use *after* a branch ***B*** is saturated and the appropriate new branch ***B'*** is explored at a branch point p from ***B***. R$may have asserted facts to the old branch ***B*** below the branch point p, and so those choices are not relevant to ***B'***. R may have used some unsuccessful choices on ***B*** which lie below p, and so those choices are again not relevant to ***B'***.
In either case, the choice ranges need to be adjusted.

A completely safe strategy is to restart the distributive choice ranges after a branch switch, for all active rules. The essential problem with this conservative strategy is that many of the DC choice-range computations (and any successful inferences) may be above the branch point. Deciding how to *rollback* the choice ranges is a challenging calculation.

5.4 Rule predicativity and indexicality definitions ⁂

This section describes the design issues regarding predicativity and indexicality regarding autolog rules, and modulators.

We wish to extend the branch indexing and the tableing techniques specified for predicative coherent logic, described in §5.1. The old methods depend on literals being predicates having names statically determined at compile time. The student is advised to review the first two chapters of this Notebook before getting into precise implementation technicalities in the following sections.

For the sake of the definitions in this section (and the next), let use characterize an autolog rule as having the intuitive meta-coherent form

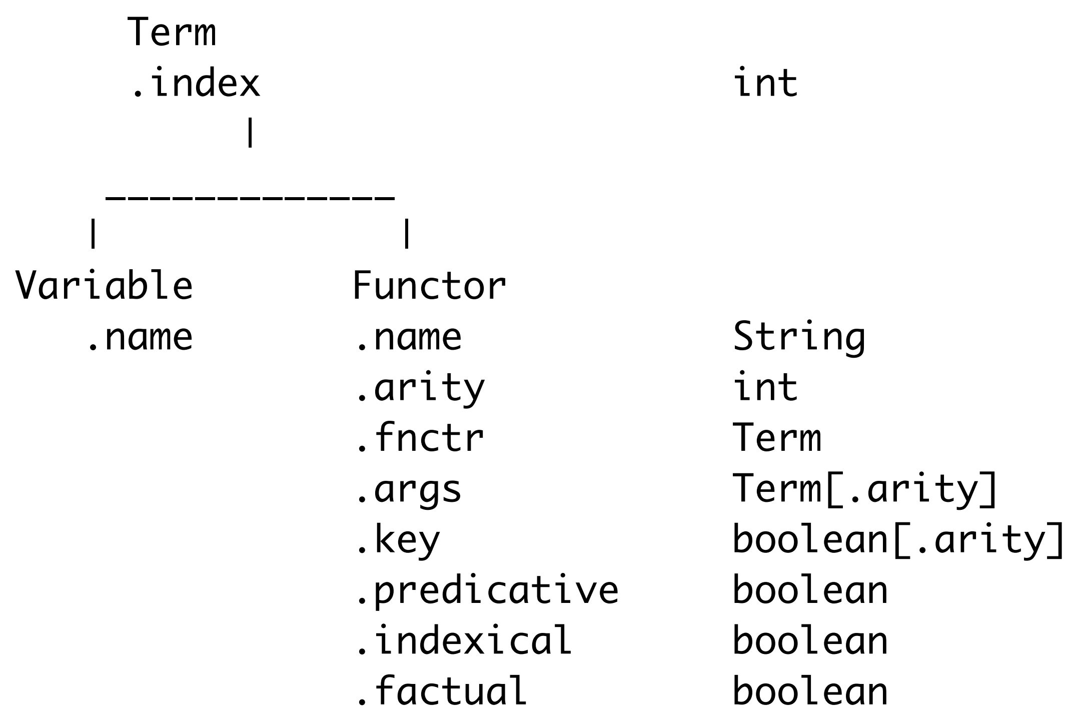
$$A_1, \ldots, A_m => C_1, \ldots, C_n \mid \ \ldots .$$

where the antecedent A's and the consequent C's are literal term factors. Rule literals are `sam.machine.lang.Functor` components having a `predicate/argument` form, that is, a *predicate* part and an *arguments* list part (possibly null). The predicate part of a literal `Functor` factor consist two code fields: `name` and `fnctr` and exactly one of these fields is null. The `name` field is a

java.lang.String component (or null) and the fnctr field is a sam.machine.lang.Term component (or null).
The args part of a literal is a list of terms (possibly empty). An argument term, if it is a Functor would itself have the form predicate/argument, and so on recursively inside terms.

The autolog compiler translates AUTOLOG.g4 grammer forms, whose input display forms are discussed in this section, into finer machine codes.

Let us review the machine codes as characterized in §3.5. Consider again the following little sam.machine.lang component diagram:



We define settings for the attributes for Functor below. The implementation algorithms based on these attribute settings are intended to make both correct and efficient Skolem machine inferences. So again, we refer to the Functor forms, both literal factors and other function terms loosely as *predicate/ argument* terms . For a literal rule factor the predicate corresponds to a logical predicate. For embedded Functor arguments, the predicate part is what would conventionally be called a *function* or a *parametric* (functorial) expression.

Exactly one the .name or .fnctr attributes of any Functor is non-null. A *named* Functor has a non-null .name String value, and a *parametric* Functor has a non-null .fnctr Term value. For example p(X) is named, p(a)(X) is *functorial* parametric and P(X) is *variable* parametric.

Some predicate-args terms input as binary infix expressions or as prefix or suffix unary operators, but do not confuse abstract component structure with a preferred display or input form of a structure. For example, the binary Functor term P∧q is named (∧) with 2 arguments, P (Variable) and q (named Functor with no arguments).

The following two *design-concept definitions* involve key concepts for autolog expressions, so their presentation is highlighted.

definition of rule predicativity

Suppose that **F** is a compiled Functor term residing anywhere in an autolog Rule component. Then **F** is *predicative* if either it was input as a named Functor or it is a parametric functor whose .fnctr component contains no Variable term. A compiled autolog rule is *predicative* provided that every antecedent and consequent literal actor is a predicative predicate-args term.

definition of rule indexicality

Assume that all rule literal are matched in given left-to-right order. Suppose that ***L*** is a compiled literal factor `Functor` occurring in the antecedent of an Autolog rule. ***L*** is *indexical* provide that either it is a named functor or it is a parametric functor all of whose variables were bound by matching literals to the left of ***L*** in the antecedent. Suppose that ***L*** is a compiled literal factor `Functor` occurring in a consequent conjunction of an Autolog rule. ***L*** is indexical in this situation provides that either ***L*** is named, or it is a parametric functor all of whose variables were bound by matching the antecedent of the rule, or were matched by satisfactions of literals to the left of ***L*** in its redsident consequent conjunction. A compiled autolog rule is *indexical* provided that all of its literal factors are indexical.

The two definitions will have analogous formulations for autolog modulators in the next section, §5.5.

The `.key` attribute for `Functor` is used for literal factors of a rule to record the boundedness of the argument entries of the literal factor which would be bound after the left-to-right matching regimen described in the indexicality definition above. If all entries of the `.key` array are `true` then the literal is *factual*. The `key` values are used to compute useful hash codes, as outlined in the following examples.

some worked examples

The following examples explain some of the finer issues involving predicativity and indexicality. The examples are also intended to motivate design issues and specifications for active literal factor search methods, which is discussed in §5.6. The active factor algorithms control the manner in which the satisfaction of a literal factor is entangled with the facts on the current (relevant) search branch. The component which decides an active factor matching algorithm specification is the `sam.machine.lang.AutoLogAnalyzer`, and the examples in this subsection are analyzed in the linked document. Hopefully a presentation of `AutoLogAnalyzer` reports for a variety of specific autolog rule literal factors will help explain what is supposed to be going on under the engine hood. The primary design goal is to advance the completeness and efficiency of finding matches for literal factors on the current search branch.

The discussion of possible matching strategy accompanying the examples is intuitive but not completely precise with regards to detailed matching algorithms design. That issue is approached in §5.6.

The Analyzer report for the following examples is linked here, and can be read along with the descriptions of the examples (copy link and open in new browser window if necessary).

> https://skolemmachines.org/adn_pages/Analyzer/AA_report_5.4.pdf

The Analyzer definitions page is linked here (copy link and open in new browser window):

> https://skolemmachines.org/autolog/help/Analyzer.html

1-
The rule

```
«
   p(a)(X) => p(b)(X).  %1
»
```

is predicative by definition since literal factors all have bound predicates. Satisfaction methods

for the predicates could use hash methods similar to those discussed in §5.1 regarding colog system methods. We discuss enhanced hash methods for autolog in §5.6.

2-
In the rule
«
```
P:T->T, X:T, P(X)(a) => P(X)(b).  %2
```
»
the antecedent and consequent parametric predicate P(X) is not predicative because it is not bound in its input form. However it is indexical because it would be dynamically bound when the rule literals are satisfied on branch in left-to-right order, thus the subsequently bound literals `P(X)(a)` and `P(X)(b)` could be retrieved from a table using hash methods similar to those discussed in §5.1 (regarding colog system methods). We discuss enhanced hash methods for autolog in §5.6.

3-
The rule
«
```
p(a,F(X))  => q(b,F(X)).  %3
```
»
is predicative and indexical, because the literal factors are named.The term argument `F(X)` in the antecedent literal is NOT indexical, because it is neither named nor bound at match-time, and this is reflected by the analyzer report when it assigns the value `key=[true,false]` for the literal.The literal could still be retrieved from a branch fact table using a hash key computed using only the bound parts of the literal `p`, arity `2`, first argument `a`, and unknown second argument `?`, for example.

The term argument `F(X)` in the consequent literal IS indexical, because it was bound in the antecedent and this is reflected by the analyzer report in the next subsection when it assigns the value `key=[true,true]` for the literal. This literal could be retrieve from a table using a hash key computed from all of its bound data.

4-
The rule
«
```
F:T->T, X:T, p(a,F(X)) => q(b,F(X)).  %4
```
»
is predicative because each literal factor is a named `Functor`. Guess the status of the arguments for the literal factors across the rule, and check your answers against the linked report.

5-
The antecedent literal factor for the rule
«
```
P(a) => Q(a).  %5
```
»
in not indexical, but the Functor does have arity 1 and a bound argument which can be used to search for a matching branch fact. Perhaps surprisingly, if the antecedent matches any branch fact, then the rule fails to apply because `Q(a)` would match the fact which was matched by `P(a)`. If the antecedent fails to match any branch fact, then again the rule fails to apply. Thus, *this strange rule always fails*. But the explanation "why" is instructive regarding how impredicativity could work.

6-
The first literal factor in the antecedent of the rule
«

```
¬P, P => false.  %6
```
»

is predicative, having predicate ¬. The second literal factor is not predicative, but it is indexical because it becomes bound after the first factor is matched. Predicative match methods are suitable for the first factor and indexical match methods will be suitable for the second factor.

7-
Reversing the order of the literal factors in the previous example
we would have

«
```
P, -P => false.  %7
```
»

Now the first factor is not even indexical, and a match method would be of the least restrictive kind (a complete wildcard, so to speak.). However, with P bound then the second factor is fully bound and an efficient match method can be used. However, because of the inherent vagueness of the first factor, using this form of the rule (rather than 5) is ill-advised. Matching methods could not use tableing, and so incremental *creep* matching along branch would be required.

8-
Consider the rule

«
```
true => P(a).  %8
```
»

This example is a little different than ***4-*** above. If the consequent literal factor P(a) fails to match current branch fact then some new witness symbol `s` is generated to replace P and the new fact `s(a)` is asserted to the branch.

9-
The rule

«
```
a:T => P:T->prop, P(a).  %9
```
»

has an analysis similar to the previous rule ***7-***, except for the type literal factors. Note that the antecedent factor and the first consequent literal factor are predicative, with named Functor '`:`' .

A non-indexical existential variable in the consequent of a rule should be replaced by a *new unique name* whenever the relevant consequent literal factors are asserted as new facts to the current search branch.

An entire AutoLog theory is said to be *indexical* provided each of its rules is indexical.

The terminology *indexical* comes from natural language usages.

"In semiotics, linguistics, anthropology and philosophy of language, indexicality is the phenomenon of a sign pointing to (or indexing) some object in the context (sense) in which it occurs. A sign that signifies indexically is called an index or, in philosophy, an indexical." (https://en.wikipedia.org/wiki/Indexicality)

In the context of AutoLog theories, indexicality means specifically that functors appearing in conjunctive contexts of rules can be indexed and have referents using tables that record facts previously inferred on a branch of a Skolem Machine. Thus, indexicality enables is an extended
kind of tableing principle for AutoLog.

The examples so far hint at a general phenomenon, that being that any **AutoLog rule can be reformulated so that the resulting version is indexical**. Actually, there would be many ways to achieve indexicality -- e.g., using type judgements before impredicative literals -- but it would be difficult to devise

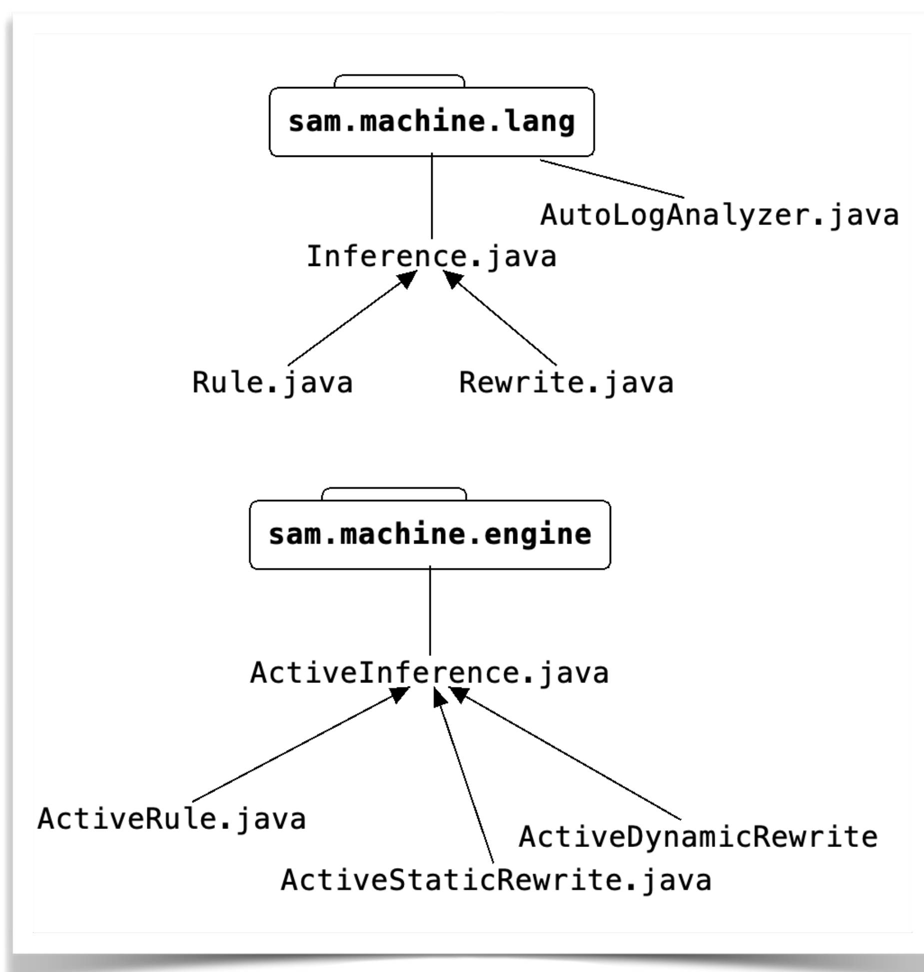
any uniform way to do this that would be strictly compatible with the intentions of the programmer. The reformulated version would generally not be logically equivalent to the original rule. It would likely apply in a more restricted manner than the unrevised rule (or even not at all).

5.5 Static and dynamic equality modulators ⁂

Review §3.6 regarding modulators as rewrite inference expressions. This section has a more detailed explanation of how Autolog implements equality modulation reasoning as a kind of *Rewriting System*, whereas §3.6 only used naive algebraic examples for motivation, based upon the `#inference_rewrite` parse rule in the `AUTOLOG.g4` grammar.

This section will explain the difference between static and dynamic rewrite modulators in the context of Autolog programs. Both kinds of rewrites generate `sam.machine.lang` components Static rewrite modulators are generated when source code is compiled and dynamic rewrite modulators are generated at runtime.

§5.6-8 will explain more details regarding the `sam.machine.engine` active rewrite components. The relevant rewrite components can be diagrammed as follows.
These active runtime components reside in package `sachine.engine,` which contains all of the active runtime components for Autolog search.



Not all components in the two packages are shown in the diagram—just the ones relevant to the descriptions in this section.

The AutoLogAnalyzer analyzes the data stored in the sam.machine.lang components. The `sam.machine.lang.Rewrite` component is the compiled language component for a static rewrite. Active dynamic rewrites are derived
from equality Functors in the consequent of a rule at run-time. These somewhat intricate relationships can be illustrated with some telling examples.

EXAMPLE A

This example illustrates both static and dynamic equality modulators, and explains the general intentions regarding autolog equality modulator implementation.

```
«
   /*
      Show that left and right inverses
      in a monoid are equal.
    */
   true => z=z,           %1 dynamic rewrites
           y∘x=e,
           x∘z=e.
   A=B, X=Y => X∘A=Y∘B.   %2 dynamic rewrite
   ((X∘Y)∘Z) = (X∘(Y∘Z)). %3 static rewrite
   (X∘e) = X.             %4 static rewrite
   (e∘X) = X.             %5 static rewrite
   y=z => goal.           %6
»
```

Inference forms `%1,2` have *implicit* dynamic equality modulators associated with equality literals in rule consequences. The equality literals are parsed and compiled as `sam.machine.lang.Functor` components. If such a rule asserts new grounded equality facts at run-time, then new active equality modulators (`sam.machine.engine.ActiveDynamicRewrite` components) may also be added to the collection of active dynamic branch modulators.

Inferences %3,4,5 are parsed as sam.machine.Rewrite components which are explicit static equality modulators whose inference actions are controlled via a `sam.machine.engine.ActiveStaticRewrite` component. Static modulators may contain variables and they may modulate terms in current branch facts. The input form for a static Rewrite equality modulator ***M*** is an *equality* term expression of the form

L = ***R.***

Such a rewrite encapsulates data which can be employed to effect the following inference action: In a current branch fact ***F***, find a term ***T*** matching ***L***, replace ***T*** with the corresponding instance of ***R***, resulting in an expression ***F '***, and eventually assert the fact ***F '*** to the branch if it is indeed a *new* fact.

An `ActiveDynamicRewrite` modulates branch facts in a similar fashion. However, dynamic rewrites only employ *grounded* term substitutions.

The `sam.machine.lang.Rewrite` static modulator component has the following (summarized) structure

```
Rewrite  extends Inference
 .Lterm           Term     the L in the modulator L=R.
 .Rterm           Term     The R …
 .converse        Rewrite  null if none
 .converse_index  int
 .predicative     boolean  Lterm has bound predicate
 .factual         boolean  Lterm is bound
```

```
.existential    int      #Vars(Rterm)-#Vars(Lterm).
.complex        boolean  Rterm more complex than Lterm
.warnings       String
```

The boolean instance method `isIndexical()` returns true provided that both `Lterm.isIndexical()` and `Rterm.isIndexical()` return `true` and `existential==0` (no free vars in `Rterm`), otherwise `false`. A static rewrite is indexical provide that it occurs in the consequent of an indexical autolog Rule. More detailed definitions are given in §5.7.

We present several partially worked examples to illustrate some of the salient issues involved in implementing a `Rewrite` as an effective active equality inference component.

As mentioned above, the finer details of the active rewrite designs are delayed until §6.2 and §6.3 . The first designs of the active components will likely require that the active rewrite is indexical. Later designs might relax this requirement is some ways. The following analyzed examples will explain in a general manner how the rewrites are intended to control equality inferences.

Example A (cont.)

We repeat the the code of this example and the comments about static and dynamic rewrites.

```
«
   /*
      Show that left and right inverses
      in a monoid are equal.
    */
   true => z=z,              %1 dynamic rewrite
           y∘x=e,
           x∘z=e.
   A=B, X=Y => X∘A=Y∘B.      %2 dynamic
   ((X∘Y)∘Z) = (X∘(Y∘Z)).    %3 static rewrite
   (X∘e) = X.                %4 static
   (e∘X) = X.                %5 static
   y=z => goal.              %6
»
```

Example A involves both static and dynamic rewrites (marked in % side comments). The two rules %1,2 have dynamic equality rewrites *in the rule consequent*. Dynamic rewrites are parsed and analyzed as sam.lang.Functor components. Rewrites %3,4,5 are static equality rewrites. Static rewrites are entered as source code forms.

An `AutoLogAnalyzer` report for the Example A input program is available online.

https://skolemmachines.org/adn_pages/Analyzer/AA_report_5.5A.pdf

Exercise: copy the program, load it into the Autolog app prototype, and generate the report using the `Compile` button.

Notice that the `AutoLogAnalyzer` analyses a rule consequent *equality* literal as a `sam.machine.machine.lang.Functor`, that could generate a dynamic rewrite at runtime, whereas

the analyzer analyzes the input form of a static rewrite as a `sam.machine.lang.Rewrite`. For example, the analyzer report for%2,3 looks as follows

```
…
2. A=B, X=Y => (X∘A)=(Y∘B).
  Rule data: A=2 C=3 U=4 V=4 existential=0 predicative=true indexical=true
             A:0 B:1 X:2 Y:3 => A:0 B:1 X:2 Y:3 {Var:index}
             factual=false complex=true
   antecedent literal factor data:
   0 key=[false,false] order=[0,1] forward=[]
     predicative=true .indexical=true .isIndexical()=true factual=false
   1 key=[false,false] order=[0,1] forward=[0]
     predicative=true .indexical=true .isIndexical()=true factual=false
      =>
   consequent literal factor data:
      0 <conjunction>.factual=true
         0 key=[true,true] order=[0,1]
         predicative=true indexical=true .isIndexical()=true factual=true
3. (X∘Y)∘Z = X∘(Y∘Z).
  Rewrite data:  U=3 V=3
             Lterm = (X∘Y)∘Z
             Rterm = X∘(Y∘Z)
             X:0 Y:1 Z:2 = X:0 Y:1 Z:2 {Var:index}
             Lterm.key=  [false,false]
             Lterm.order=[0,1]
             complex=false predicative(Lterm)=true factual=false
…
```

The analysis of inference form %2 shows that the an application of the rule can generate a grounded consequent equality fact (predicative=true indexical=true factual=true) and thus also a potential dynamic equality rewrite.

The analysis of inference form %3 shows that the an application of the static rewrite to a term in a branch fact would result in the replacement a grounded term in the branch fact by another *grounded* term:
`U = #vars(Lterm) = 3 3 = #vars(Rterm) = V.`

The following textual illustration is a hypothetical proof of the goal which suggests potential rewrite `sam.machine.engine` actions (without yet specifying a larger control regimen, which is delayed to §5.7-8). The proof is arranged in three columns. The first column is the branch index n of the generated fact, the second column is the proof tree annotated to show the inference actions performed at each step and the third column shows the potential dynamic rewrite actions generated at each inference step.

```
n          BRANCH           dynamic rewrites

0           true
             |- %1
1          z = z             {z/z},  {z/z} discard
             |- %1
2      (y∘x) = e            1{(y∘x)/e}, 2{e/(y∘x)}
             |- %1
3       (x∘z)=e             3{(x∘z)/e}, 4{e/(x∘z)}
             |- %2 (2,1)
4    (y∘x)∘z = e∘z          5{(y∘x)∘z/e∘z}, 6{e∘z/(y∘x)∘z}
             |- %3
5    y∘(x∘z) = e∘z             ...
             |- dynamic_mod3
6        y∘e = e∘z             ...
             |- %4
7          y = e∘z             ...
             |- %5
8          y = z               ...
             |- %6
9           goal

                  Example A, Autolog proof
```

Some more detailed explanation is required to understand what is supposed to be going on in the (handmade) proof.

At branch inference n=5->n=6, for example, the dynamic rewrite substitution `3{x∘z/e} – generated at the time that the new fact at n=3 was asserted` to the branch — is applied to the left term of the fact at n=4.

At branch inference n=6->n=7, for example, the static rewrite %4 is applied to the left term of the equality fact at n=6.

Notice that potential dynamic equality rewrites associated with new facts at n=`5,6,7,8` are reference as "…" in the proof illustration but were not needed to
complete the displayed proof.

Example A is a modification of a coherent logic problem presented in §6.3 of

https://skolemmachines.org/reports/colog/colog.pdf

Example B

This reprogrammed version of Example A uses only dynamic *guarded* rewrites.

```
«
   true => x:dom, y:dom,
           (y∘x)=e.
   true => z:dom,
           (x∘z)=e.
   A:dom, B:dom, X:dom, Y:dom,
```

```
      A=B, X=Y => X∘A=Y∘B.
   X:dom, Y:dom, Z:dom =>
       (X∘Y)∘Z = X∘(Y∘Z).
   X:dom => X∘e = X.
   X:dom => e∘X = X.
   y=z => goal.
»
```

Notice that domain *type* guards are employed. A *predicate* guard would work as well — dom(X) rather than X:dom. The issue of guarded rewrites is continued below.

Exercise: Attempt a proof that mimics the proof of Example A. If that is not possible, then explain why.

Example C

Consider the pair of *converse* static rewrites

```
«
   X+0=X.
   X=X+0.
»
```

The code analyzer produces the following report

```
1. X+0 = X.
  Rewrite data:  U=1 V=1
             Lterm = X+0
             Rterm = X
             X:0 = X:0 {Var:index}
             converse = #2
             Lterm.key=  [false,true]
             Lterm.order=[1,0]
             complex=false predicative(Lterm)=true factual=false
2. X = X+0.
  Rewrite data:  U=1 V=1
             Lterm = X
             Rterm = X+0
             X:0 = X:0 {Var:index}
             converse = #1
             complex=true predicative(Lterm)=false factual=false
   WARNING: variable Lterm.
```

The warning for the second rewrite announces that the rewrite could apply to any term in any branch fact, which is not necessarily unfounded if all terms in the program containing this converse pair of rewrites share the same algebraic type (e.g. all of the same variety). But otherwise, in programs having different sorts of variables in different rules and rwrites, it may be better to use guarded dynamic rule-based rewrites, e.g.,

```
«   // type guard
   X:t => X+0=X.
»
```

or a predicated guard

```
«   // predicate guard
   dom(X) => X+0=X.
»
```

The converse static rewrites can be reprogrammed using either converse of a dynamic rewrite. The corresponding guarded dynamic rewrite can be introduced using either of a pair of rewrites. For example, ….

```
«
    true => p(a).
    X+0=X.
    p(a+0) => goal
»
```

will not yield not a proof without the symmetric static rewrite, but

```
«
    true => a:t, p(a).
    X:t => X+0=X.
    p(a+0) => goal
»
```

does yield a proof, because the symmetric grounded dynamic rewrites are automatically generated. Exercise: demonstrate these claims with hand-generated proofs. This exercise illustrates that ***static equality rewrite equalities are not automatically symmetric but grounded dynamic rewrites are symmetric***. (This is a current intention, Feb. 2021, but things may change in some regards for autolog rewrite technical implementation issues.)

The important issue of efficient control over (avoiding) redundant branch actions resulting from symmetric rewrites is considered in §5.8-9. Generally speaking, redundant generation of repeated facts is check and avoded but here may be more efficient algorithms that simply avoid symmetric rewrites in the first place.

The issue of ensuring that static rewrites do not appear in autolog programs where there may be unintended substitutions is tricky, but enhancements to the autolog analyzer may catch some such occurrences. This is another issue that will be revisited in §5.7-8.

(It is recommended that the reader generate and inspect an analyzer report for all of the rewrite examples.)

RST Examples D

Let us consider the *reflexive*, *symmetric* and *transitive* properties of equality as *put into effect* using static and dynamic rewrites.

1. ***Symmetry***

A coherent symmetry rule for equality might be expressed as follows

```
X=Y => Y=X.
```

We are interested in verifying that the intent of such a rule can be put into effect using dynamic rewrites rather than explicitly employing the rule itself. To this end, consider a concrete example:

```
«
    true => a+b=c.  // 1
    c=a+b => goal.  // 2
»
```

```
    true
     |- // 1
   a+b=c  * {a+b/c , c/a+b}
     |- * rewrite a+b/c
    c=c
     |- *  rewrite c/a+b
    c=a+b
     |- // 2
    goal
```

Note that dynamic rewrites effect distinct individual substitutions in one stage of application, requiring perhaps several stages to effect blanket substitution.

A *grounded* static rewrite could be treated as a dynamic rewrite. That is

```
«
   a+b=c.          // 1
   c=a+b => goal.  // 2
»
```

might be implemented so as to afford the same proof. We will revisit this later in §5.7-8.

2. ***Transitivity***
A coherent transitivity rule for equality might be expressed as follows

```
X=Y, Y=Z => X=Z.
```

We are interested in verifying that the intent of such a rule can be put into effect using dynamic rewrites rather than explicitly employing the rule itself. To this end, consider a concrete example:

```
«
   true => x=y, y=z.  // 1
   x=z => goal.       // 2
»
```

```
0  true
    |- // 1
1  x=y  *  {x/y , y/x}
    |- //1
2  y=z  ** {y/z , z/y}
    |- **  rewrite y/z at branch 1
3  x=z
    |- // 2
4  goal
```

3. ***Reflexivity***
A —*useless*—static reflexivity rule for equality might be expressed as follows

```
X=X.
```

Reflexivity has no dynamic rewrite *computational content* (replacing a ground term by itself). But *the fact of reflexivity* can be used as an inductive start for building more elaborate dynamic rewrites as in a rules like %2 in Example A (or the Example B version of that rule).

Asserting the fact of reflexivity can follow at least two styles, via type or predicate *guards*:

«
```
dom(X) => X=X.
X:T => X=X.
```
»

where implementation limitations do ***not*** generate dynamic rewrites. We are open to suggestions for alternate computational implementations.

Parametric rewrite Example E

There are several ways of expressing aspects of lambda calculus (and typed lambda calculus) using an autolog meta-programming approach, but we have not yet posted a lecture on that subject in particular on the SkolemMachines.org webpage. We suspect that a general parametric function approach using autolog requires careful code design (which is in in line with a similar issue regarding lambda calculus notational intricacies).

The following code fragment illustrates how one might mimic both *α-conversion* and *β-reduction* in a nutshell using parametric functors and equality rewrite modulators.

«
```
true => dom(3).              %1
dom(Z) => λ(x,f(x))(Z)=f(Z). %2
f(Z)= Z*Z+2.                 %3
λ(x,f(x))(3)=A => goal.      %4
```
»

The following is a dry-lab proof tree (and answer A extraction) for the little program.

```
0   true
      |- %1
1    dom(3)
      |- %2
2     λ(x,f(x))(3)=f(3)  & rewrites generated
      |-%3          ^ modulate
3    λ(x,f(x))(3)=3*3+2
      |
4     goal  (A=3*3+2)
```

The inference from branch step 1 to 2, using the guarded rule %2, involves both an *α-conversion and the β-reduction for* dom *argument* 3. The dynamic rewrites generated at the application of %2 at step 1 are not shown (and not used in the rest of the displayed proof). The lambda expression

Exercise*: Reprogram Example E so that a dynamic rewrite generated by %2 is employed in the proof. (Hint: modify %3 with a guard.)*

The AutoLogAnalyzer *has this to say about our rule %2:*

```
2. dom(Z) => λ(x,f(x))(Z)=f(Z).
  Rule data: A=1 C=2 U=1 V=1 existential=0 predicative=true indexical=true
             Z:0 => Z:0 {Var:index}
             factual=false complex=true
   antecedent literal factor data:
   0 key=[false] order=[0] forward=[0]
     predicative=true .indexical=true .isIndexical()=true factual=false
      =>
   consequent literal factor data:
      0 <conjunction>.factual=true
         0 key=[true,true] order=[0,1]
         predicative=true indexical=true .isIndexical()=true factual=true
```

Note in particular that, with the guard, and the use of a *faux* variable x (rather than autolog variable X), the parametric lambda predicate λ(x,f(x)) *is indexical* (see §5.7*)*. One could consider how not to use faux variables.

Names vs Terms Example F

This example may seem perplexing at first glance.

```
«
  true => p(a).
  p=q.
  q(b) => goal.
»
```

Answer the following questions:
a) Carefully explain WHY autolog should NOT prove!
b) Reformulate in colog14l (using equality axioms) and
   again explain why NO proof should obtain.
c) Now modify the theory like this:

```
«
  true => p(a).
  p(X)=q(X).
  q(b) => goal.
»
```

and now explain how an autolog proof could procede.

The static and dynamic modulator designs in this section require more refinement before integration with rule inference mechanisms in §5.7-8. The aspects discussed in this section form a basis for *coherent equality modulators*.

We have side-stepped some issues regarding non-indexical equality modulators — leaving the issue to §5.7-8.

5.6 Term profiles and branch fact entanglements ⋇

Generally speaking an *entanglement* refers to a situation in which a term in a rule or rewrite modulator *might* match a term in a branch fact, because of its structure. Branch facts affect inferences by *exactly* matching terms in rules or rewrites.

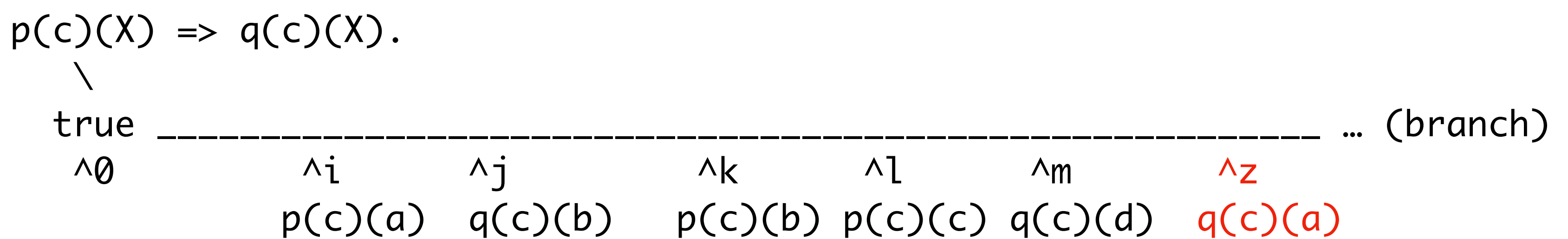
This section concerns designs for term *profiles* in order to implement efficient procedures for matching rule literal factor term with branch facts, and efficient procedures for the matching of Rewrite Aterms With terms that appear in branch facts. We will call these designs *entanglement designs* because they involve how it is that terms ***T*** in rules and modulators become associated with possible terms ***T'*** occurring in branch facts that might match ***T.***

The first subsection gives additional motivational examples of entanglements for rule literals, and the second subsection gives some motivational examples of entanglements useful for rewrite modulators. In §5.8 we will explore detailed analysis which specifies general autolog term profiles and entanglement implementations using hash tables.

Rule literal entanglements with branch facts (examples)

This subsection is intended to review §5.1 (which used predicative coherent examples) using the profile and entanglement metaphors for a simple parametric autolog examples .

Now rule antecedent literals are autolog terms which must match branch fact terms in order for the rule to apply to the branch. Suppose that the rule has the simple predicative form `p(c)(X)=>q(c)(X)` and that the branch has some facts already asserted. Assume that the displayed p(c)-facts and `q(d)`-facts shown below are the only `p(c)` or `q(c)`-facts currently on the branch.

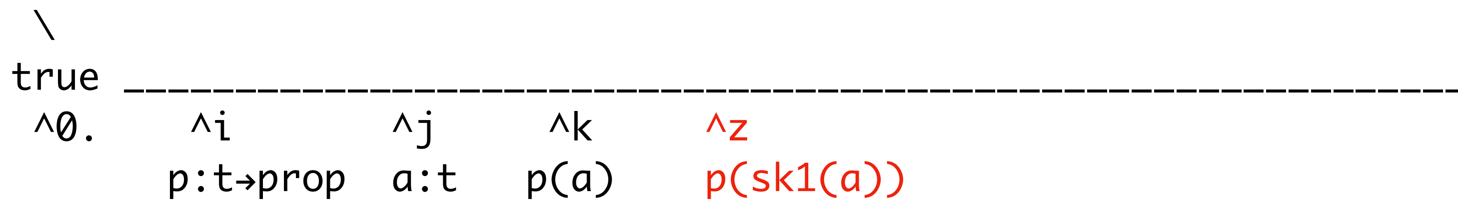


The antecedent literal `p(c)(X)` is entangled with the facts at indices `i`,k, and `l` on the branch, as follows: Let us refer to the hashcode (profile) for the literal factor `p(c)(X)` as `[<p/1(c)>/1]` (because predicate `p, arity 1, applied to c is a parametric functor of arity 1)` is the bound profile of the literal. If the table of entanglements is T, then `T.get([<p/1(c)>/1])` would return something like `<i,k,l>`, the last expression referring to the bucket list of positions which was updated as the `p(c)`-facts were previously asserted to the branch. These assertions were associated with table hashcode `[<p/1(c)>/1]` because the Analyzer determined this possible hashcode entanglement profile of the literal factor when the rule was compiled.

Similarly, `T.get([<q/1(c)>/1])=<j,m>`. When the rule now attempts to apply itself to the current branch, it finds that it matches antecedent at index `i`, `p(c)(a)`, and does not already satisfy `q(c)(a)` among bucket <j,m> branch positions, so asserts `q(c)(a)` to branch and updates the q(c) bucket with the corresponding branch index z . Similarly the rule attemps application at k, but `q(c)(b)` is already satisfied at `j`, so this match of `p(c)` at index k is ignored. Continuing, … `q(c)(c)` will also be asserted to branch and the relevant `q(c)`-bucket updated. (Notice that this rule is predicative, which simplifies the entanglement mechanism.)

As exemplified above, it is somewhat tedious to explain the details of profiles and entanglements for parametric autolog in a simple manner.

As a second example consider the application of the active rule corresponding to the autolog rule displayed below, using initial setup assumptions similar to the previous example.



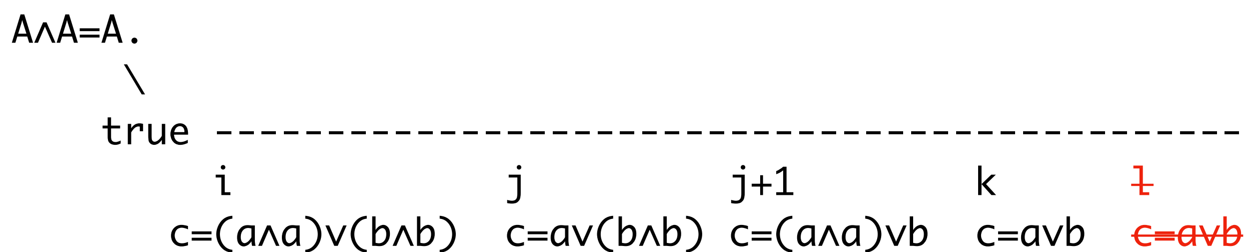
For this example, analysis details would determine that facts at branch positions i,j,k conjunctively combine to satisfy the antecedent of our rule, and bind parametric proposition P to p. So, there is a new existential skolem eigenvalue sk1 generated for F and p(sk1(a)) is asserted at bottom branch position z. (For this example we hid some dirty notational detail for smoother reading.)

The general restriction of matching to bucket lists rather than the entire branch enjoys a significant increase in efficiency for *long* branches. However, finding efficient profile/entanglement mechanism is a big challenge, as we will see in §5.8.

Rewrite entanglements with branch fact terms (examples)

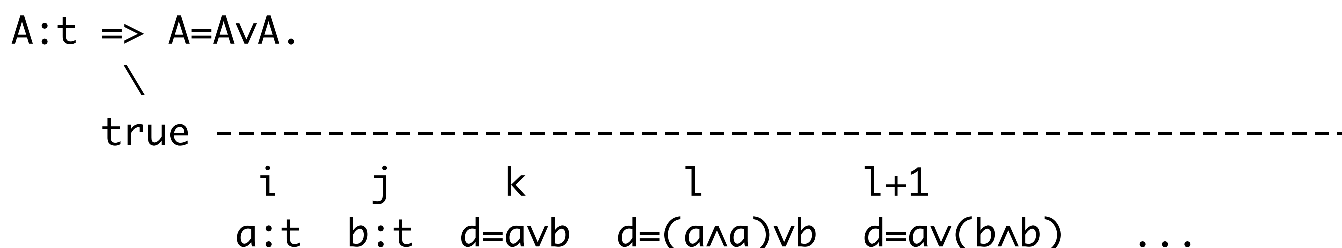
This subsection uses the profile and entanglement metaphors to motivate some possible aspects of rewrite actions using profiles and entanglements.

Consider first an unguarded rewrite



For this example, we do not specify exactly how it is that rewrite Aterm A∧A entangles itself with facts on the displayed branch (that is open for specific development design decisions). However, suppose that algebraic fact c=(a∧a)v(b∧b) has been asserted at position i on the current branch somehow and that the active version of the rewrite finds this relevant fact on the branch. A saturating action of the rewrite yields the two modulates at positions j and j+1. Later (next inference stage) the rewrite, acting on facts at j and j+1 yields new facts at positions k and l, except that at l we have a repeat which is not actually asserted to the branch!

Now consider the reverse modulator rewrite inference, as a guarded dynamic rewrite, applied to a branch. *For blessed simplicity*, read this related example as being not specifically involved with the rewrite actions just discussed above — even though they would be in practice if both rewrites were paired in our theory.



Let us assume that the guard is used to entangle variable X with object of type t (at i and j) and that fact d=avb (at k) has been asserted to this active current branch. At positions l and l+1, our rewrite

saturates d=a∨b by acting on its relevant terms, a and b. Exercise: What is it that could happen further down this branch in subsequent inference stages?

Since *inference staging* affects the manner in which inferences (rules and rewrite modulators) will act on the current branch, we next include a section on inference staging, before continuing, in §5.8, with the finer details regarding term profiles, entanglement, hash table mechanisms and branch fact match searching mechanisms.

5.7 Staged concurrent inference actions ⁂

This section outlines an algorithm for implementing *staged* active inferences for the Autolog search engine. The staged applications (together with distributed choice methods) allows advanced inference fairness, and termination detection. In addition, the algorithm employs concurrent definite rule and rewrite applications during each stage, followed by controlled application of selected existential rules. Allowing concurrent application of active definite inferences has the potential to significantly speed up the overall speed of inference action.

This section does not take into consideration the staging of Autolog active static and dynamic equality rewrite modulators. It does however illustrate the general idea of stage inferences. The inclusion of other active inference components is explored in detail in §6.2 and §6.3.

staging inferences

A *definite* autolog rule has no free variables in the consequent of the rule; that is, every variable in the consequent of a definite rule occurred in the antecedent of the rule. The free variables in the consequent of a nonfinite rule are existential variables. (Review equation 1.1 in Chapter 1.)

A *definite* autolog rewrite has no free variable in its Bterm; that is, every variable in the Bterm occurred in the Aterm. However, a free variable in the Bterm of a rewrite are not treated as a kind of existential variable for autolog. The design plan is to warn about such variables occurring in any rewrite Bterm or perhaps generate an input exception. (This issue is still open to some consideration).

For an autolog theory T we have that

$$T = D \cup E$$

where D consists of the definite rules and rewrites, and E consists of the existential rules. The staging of the definite rules D involves a sequence of phases for applying the rules of the theory. The initial phase or stage is

S_0 = `{true}`

For stage 1 the only applicable rules would be `true` rules of the form

```
true => ….
```

For example, the aplication of the single rule `true => p | q` would produce

S_1 = `{true, p}`

The stage S_1 also represents the current branch of the search tree.The second consequent choice q would be delayed for consideration after the termination of the branches below p. The details regarding how choices are handled as described in §5.1 That aspect of rule search is not changed. It is the details for staging that are outlined in this section. Upon termination of all branches under p, a new branch or stage (which we also call) S_1 would use the next choice, that is S_1 = `{true, q}`.

The stages are considered to be ordered sequences rather than just sets. If there are several `true` rules, they are all applied in the order that they appear in the sequence of rules of the theory. The `true` rules are never applied except at the very top stage S_0, and then do not need to be considered beyond that stage.

Let us suppose that the definite rules and rewrites (in theory *input* order) are

$$D = \{ d_1, \dots , d_m \}$$

and that the existential rules (in given *input* order) are

$$E = \{ e_1, \dots , e_n \}$$

Assume that S_i, is the current stage. The next stage is developed in two ways. First, all of the rules is D are applied to S_i, ***concurrently***, and S_{i+1} is sequenced by adding all new inferences in the order produced by the rules of D in the order of the *D rule*. (The detail about "in given order" is merely a device to insure that different searches yield the same reproduceible trace results, which is a convenience for output checking and debugging search code.) We say that this pseudo algorithm produces a new substage $D^*(S_i)$ -- which amounts to all of D exhaustively applied to S_i,.

The next step in the search algorithm, extending the substage $D^*(S_i)$ is to attempt one application (if possible) from each existential rule E, and this results in the next stage

$$S_{i+1} = D^*(S_i) \cup E(S_i) = (D^* + E)(S_{i,})$$

The rules and rewrites ($D^* + E$) are processes which use a *distribute choice algorithm* (or another fair algorithm) to decide what the next fair choice for inference is.The remainder of the search algorithm is then to continue the process until termination of the current branch, initialize any pending new branch, and continue, until there are no more open branches. For short, we refer to this as a ($D^* + E$) *staging algorithm*.

the (D + E) algorithm illustrated*

To illustrate the D^* aspect of the staging algorithm, consider the following coherent algebraic logic problem in a rule formulation : *Show that left and right inverses in a monoid are equal*. This example has E=∅.

```
% rule1                                             to show
y=z => goal.
% rule2                                             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
% rule3                                             closure for *
dom(X), dom(Y) => dom((X*Y)).
```

```
% rule4                                              associativity of *
dom(X), dom(Y), dom(Z) => ((X*Y)*Z)=(X*(Y*Z)).
% rule5                                              e is identity for *
dom(X) => (X*e)=X, (e*X)=X.
% rule6                                              = reflexive
dom(X) => X=X.
% rule7                                              = symmetric
X=Y => Y=X.
% rule 8                                             = transitive
X=Y, Y=Z => X=Z.
% rule 9                                             substitutivity for =
A=B, C=D => (A*C)=(B*D).
```

The following display shows a proof stage-by-stage. The actual proof was generated by colog14I (which is not staged). However the proof was rearranged by hand edit to show a staged proof.

Note well that colog14I did not employ either static not dynamic rewrites per se. All equality inferences were coherent rule based. ***If we were to use Autolog to compute this program we would disable the generation of dynamic rewrites since they would not be required for proof.***

```
           uses
 stage  rules#  stages   inference
  10      1       9      y=z => goal
------------------------------------------------------------------
  9       7       8      z=y => y=z
------------------------------------------------------------------
  8       8      5,7     z=((y*x)*z), ((y*x)*z)=y => z=y
  ----------------------------------------------------------------
  7       7       6      y=((y*x)*z) => ((y*x)*z)=y
------------------------------------------------------------------
  6       8      5,2     y=(y*(x*z)),(y*(x*z))=((y*x)*z) => y=((y*x)*z)
--------------------------------------------------------------------
  5       8      3,4     y=(y*e), (y*e)=(y*(x*z)) => y=(y*(x*z))
  5       8      3,4     z=(e*z), (e*z)=((y*x)*z) => z=((y*x)*z)
--------------------------------------------------------------------
  4       7       3      (y*(x*z))=(y*e) => (y*e)=(y*(x*z))
  4       7       3      ((y*x)*z)=(e*z) => (e*z)=((y*x)*z)
--------------------------------------------------------------------
  3       7       2      ((y*x)*z)=(y*(x*z)) => (y*(x*z))=((y*x)*z)
  3       9       2      y=y, (x*z)=e => (y*(x*z))=(y*e)
  3       9       2      (y*x)=e, z=z => ((y*x)*z)=(e*z)
  3       7       2      (e*z)=z => z=(e*z)
  3       7       2      (y*e)=y => y=(y*e)
--------------------------------------------------------------------
  2       4       1      dom(y), dom(x), dom(z) => ((y*x)*z)=(y*(x*z))
  2       5       1      dom(z) => (z*e)=z, (e*z)=z
```

```
  2      5     1     dom(y) => (y*e)=y, (e*y)=y
  2      6     1     dom(z) => z=z
  2      6     1     dom(y) => y=y
----------------------------------------------------------------------
  1      2     0  true => dom(e),dom(x),dom(y),dom(z),(y*x)=e,(x*z)=e
----------------------------------------------------------------------
  0                  true
```

The table only shows stages for a resulting *proof*. The actual search required 5538 inferences, or 1384 inferences with a complexity cut. (Exercise: download colog14l prover and use it to approximate how many inferences might be required at each stage.)

Exercise: Reformulate the example using algebraic rewrite modulators, a data rule and a goal rule . We can later use it a good proof-test for the initial autolog implementation having fully implemented rewrite actions. Examples A and B of §5.5 outline preliminary approaches to this exercise.

*concurrent design for D**

As shown by the example is the previous subsection, an algebraic logic problem might require abundant inferences at each stage.The same can be true for geometric logic problems with branching.Computing these inferences concurrently is worthy of many software experiments, possibly employing neural net techniques. To that end, we will employ experimental designs for AutoLog using the `java.util.concurrent package.` The specific design details are pending, and will be described in §6.6, as an issue of design integration.

6 Autolog24 specific design proposals ⁜

AUTOLOG24 is the first planned prototype release, hopefully some time in year 2024, or soon thereafter. All of the foregoing sections of this **Autolog Design Notebook** are considered as general autolog system conceptual development notes. Chapter 6 contains specific AUTOLOG24 implementation descriptions. New development design components are used to create the active engine runtime match and search components. The look of the desktop development app (editor and viewer) has been changed

6.1 Indexicality related definitions ⁜

Autolog23 checks input programs (`Rule` or static `Rewrite` Inference forms) for indexicality, and gives warning if a program input form does not satisfy various indexicalty conditions. The `AutoLogAnalyzer`, in conjunction with component constructors, computes the indexicality conditions. Code warnings (marked with ↯) regarding non-indexicality are added to the program listing in the Autolog23 Viewer. At this time the indexicality warnings do NOT necessarily prevent an attempt to compute the test routines that are described in §6.2 (matching methods) or §6.3 (branch fact entanglements).

Previous informal definitions for indexicality and related concepts are repeated here along with code specifications for computing recent thinking regarding indexicality. This section will be amended as Autolog23 requirements develop, depending on the outcomes of further development testing.

The details of what constitutes indexicality requirements for an Aurtolog program is also an aspect of Autolog system development itself and is not necessarily a concept that is totally fixed in advance of future testing. Indexicality restrictions will ultimtely be governed by how effectively and efficiently match and search methods work, based upon early system testing.

indexicality requirement for an autolog `Rule` *inference form*

Assume that the rule R is expressed as

```
         A        =>       C1       | ... |      Cn
     A1, ..., Am => B11, … ,B1k1 | ...  | Bn1,...,Bnkn
```

where the A's are antecedent literals and the B's are consequent literals and the C's are the consequent conjunctions. Each literal A or B is a `sam.machime.lang.Functor` component. It it is a named functor (its `.name` attribute is a String fixed at input of the rule R) or it is a parametric functor (its `.name` attribute is null and its `.fnctr` attribute is itself a `Variable` or `Functor` term at input of the rule R). Named literals L are predicate (`L.predicative==true`), set by the `AutoLogAnalyzer` at compile time.

A literal L in the antecedent A is indexical provided that it is either predicative (named) or, if L is a parametric Functor we have that all of the variables in `L.fnctr` have all been seen during the processing of the literals to the left of L in the antecedent A, and then `L.indexical=true` (*shallow indexicality,* disregarding arguments).

We have `L.isIndexical()` is `true` (*deep indexicality*, considering arguments also) provided that all of `L.fnctr.isIndexical==true`, computed via recursive descent into L's term structure.

For a rule literal B_{ij} in consequent conjunction C_i the definition of predicative is the same and just given for a literal in antecedent A. The definitions for indexicality are similar, except for an important detail: When the `AutoLogAnalyzer` determines B_{ij}`.indexical` it checks that B_{ij} is either predicative, or that the variables in B_{ij}`.fnctr` were bound while processing literals in antecedent A OR literals in C_i that appear to the left of B_{ij} .

Literal terms have "deep" indexicality as determined by the following method defined in `sam.machile.lang.Term` . Notice that the deep indexicality descends into functor arguments. The field value `f.indexical` for a Functor is assigned by the `AnalogAnalyzer` as parse forms for `Variable` bindings in a ***left-to-right*** manner.

```
/**
  *  method in sam.machine.lang.Term
  *  Is this term deep indexical?
  *  Called after AutoLogAnalyzer analyzes
  *  input autolog program.
   */
public boolean isIndexical() {
   if (this instanceof Functor) {
      Functor f = (Functor)this ;
      if (! f.indexical) return false ; // marked by analyzer
      for (int i=0 ; i < this.arity ; i++)       // deeper
         if (! f.args[i].isIndexical()) return false ;
      return true ;  // indexical functor with indexical args
   }
   else return true ; // Variable argument is indexical (not Functor)
}
```

If R is a rule then `R.isIndexical()` is defined so that the method returns `true` only when each literal factor L of R has `L.isIndexical()==true`.

Some of the following autolog rule examples are marked with ↯, indicating that they are not indexical (otherwise ✓). In each case explain why. And, for those which are indexical, explain how variables get bound to make the rule indexical.

```
p(X) => goal.           ✓
P(X) => goal.           ↯
pred(P), P(X) => goal.  ✓
P:t→t, P(X) => goal.    ✓
```

For example, the first rule is indexical because it is actually predicative. The second rule is not indexical because the antecedent literal has a variable parametric predicate. The third rule, in effect, binds the predicate P before it would be matched in left-to-right fashion; the fourth rule similarly types the predicate P.

When an autolog program is compiled from the AutologEditor, it uses the ↯ mark (only) to signal a *currently* non-indexical Inference form.

Continue with more rule examples:

```
true => p(X)(a).                  ↯
 true => X:T, p(X)(Y).            ✓
 true => pred(P), P(X).           ✓
 true => p | Q.                   ↯
 pred(P), P(H(X)) => goal.        ↯
 p(X) => q(X) |  p(H(t)).         ↯
 p(X) => q(X) |  H:t→t, p(H(t)).  ✓
 H:t→t, p(X) => q(X) |  p(H(t)).  ✓
 dom(X) => X=X∧X.                 ✓
 true => X=X∧X.                   ✓
```

Load examples into the autolog editor, compile and then read the Analyzer reports. In addition, generate your own examples with various levels of rule complexity to test for indexicality. The last two examples just above anticipate the discussion of dynamic rewrite modulators below; however, notice that the last rule would not likely be a useful rewrite modulator since its action would only be to introduce a useless equality s=s∧s, for some new Skolem constant s.

indexicality condition for static Rewrite inference forms

The static rewrite modulator specification is defined in the input language component `sam.machine.lang.Rewrite.java`. There are various proposals being considered as an indexicality requirement for static program rewrite modulators. A *preliminary* definition can be summarized as follows.

If Lterm=`Rterm` is the input form for the static rewrite modulator `M` then an intuitive description of when `M.isIndexical()` returns `true` is that

- `Lterm instanceof Functor` returns `true`,
- both `Lterm.isPredicative()` and `Rterm.isPredicative()` return `true`, and
- `Rterm` contains no argument variable that did not occur in `Lterm`

Notice that this the **indexicality** of static rewrite modulators is more stict than that of rules, requiring named or ground parametric operators (i.e.m predicative) in input forms for `Lterm` and `Rterm`. We will discuss the predicativity condition more fully at the end of this section.

Using the preliminary definition, some of the following static rewrite modulators are marked with ↯, indicating that they are not indexical (otherwise ✓). In each case explain why.

```
X∧X=X.               ✓
X=X∧X.               ↯
X•e=X.               ✓
(X•Y)•Z = X•(Y•Z).   ✓
e=X•e.               ↯
f(a)(b)=f(b)(a).     ✓
f(a)=f(H(a)).        ↯
p(a)(X)=X.           ✓
a=a•e.               ✓
```

The last example of a static rewrite modulator, being ground, may be implemented as if it were the dynamic rewrite modulator generated by the rule

```
true => a=a•e.     ✓
```

That is, the static rewrite of a ***grounded*** identity can be implemented as both a rewrite modulator AND as a fact to be added to the current branch. We discuss automatically forcing this behavior also for the static rewrite modulator later.

Alternative definitions of indexicality for static rewrite modulators are considered in §6.3, where branch fact entanglements for rules and modulators is considered using motivational examples.

indexicality for active dynamic rewrite modulators

A dynamic rewrite modulator arises from an L=R identity literal form in the consequent of an indexical autolog Rule form, and all such dynamic rewrites are allowed (✓) by default.

Here are some examples:

```
fcn(H) => f(a)=f(H(a)).   ✓
dom(X) => e=X•e.          ✓
dom(X) => X=XvX.          ✓
dom(X) => dom(Y), X•Y=e.  ✓
true => a=a•e.            ✓
dom(X) => X•a=b | X•b=a.  ✓
```

The next-to-last rewrite was discussed above in comparison with
the static modulator (`a=a•e.`)

Notice that a possible use for a dynamic rewrite modulator is when one intends the rule antecedent to limit the substitution for a variable (rather than using a related static rewrite modulator).

In regards to dynamic modulators arising from indexical rules which have multiple disjuncts in the consequent (such as the last rule above), if the relevant dynamic rewrite is a literal factor in the current chosen consequent for branch expansion, then the dynamic rewrite identity is added to the branch as a fact and the dynamic rewrite modulator is added to the dynamic list of active modulators. Upon branch switching (corresponding to consequent switching) the appropriate dynamic modulator is removed from the active list of dynamic branch fact modulators. (A description of detailed algorithms for this is delayed to §6.2 or §6.3.)

Equality reasoning for colog14I typically used ***intensional*** meaning for equality identities specified in colog rules. This involved using axioms for reflexivity, symmetry and transitivity of the equality operator, AND substitution rule axioms for the replacement of term L by term R whenever ground fact L=R resides on the current branch. In this way L can in effect be eventually replaced by R in branch facts using appropriate active rule inferences (only, no modulation) for appropriate user-specified substitution rules.

Equality reasoning for Autolog23 on the other hand proposes to allow direct active rewrite modulators as an alternative active inference mechanism for (equality) identities. This can be thought of as an ***extensional*** method for direct-substitution equality reasoning.

For a specific equality problem formulate intensionally in colog14Isee Example 13 of §6.3 in the colog primer https://skolemmachines.org/reports/colog/colog.pdf . This algebraic logic example is reworked (by hand) in this Notebook using level saturation in §5.7, assuming that active dynamic modulators are turned OFF (see this Notebook §6.3 re turning off modulation but keeping inferred rule identity inferences as facts). For the equivalent problem formulated extensionally, using both static and dynamic modulators, see **Example A** in this Notebook §5.5.

Note that this characterization of intensional/extensionsal equality inference specification for Autolog23 could affect the operational meanings of inference forms, e.g.,

```
X=Y => P:X=Y.
P:X=Y => X=Y.
```

Why is it not possible to express an intensional meaning for identity types using a static modulator?

predicativity and by-name conditions for inference input forms

The active inference component designs in the next section, §6.2, depend upon ***structural aspects*** of the input program inference forms. Before proceeding to the active component designs, we will describe several of these input structural forms. Then, various ATPR scenarios can be specified as determined by corresponding structural aspects of the input inference forms. Of primary interest will be the following structural inference forms (designated by their input inference-form recognition methods): `-. isIndexical()`, `.isPredicative()`, and `-.byName()`.

The `-.isIndexical()` recognition method was described in the previous section. The `-.isIndexical()` method can be called for any `Term, Rule, Rewrite, or Theory.` This indexical scenario is REQUIRED for the input theory/program for the AUTOLOG23 system. The other relevant scenarios are restrictions of the indexical scenario.

The `AutoLogAnalyzer` parses inference input forms and detects predicativity, which is that any predicates or operators are grounded at input. The AA marks `Functor` language components as `.predicative=v` where v is `true` provide that the `Functor` is named or parametric grounded. Here is the code from `sam.machine.lang.Term.java`

```
/**
  *  Is this term deep predicative?
  *  Called after AutoLogAnalyzer analyzes
  *  input program.
  *  @version 11/13/2021
   */
 public boolean isPredicative() {
    if (this instanceof Functor) {
       Functor f = (Functor)this ;
       if (! f.predicative) return false ; // marked by analyzer
       for (int i=0 ; i < this.arity ; i++)       // deeper
          if (! f.args[i].isPredicative()) return false ;
       return true ;  // predicative functor with predicative args
    }
    else return true ; // true for Variable arg by default
 }
```

As expected, for a `Rule r`, `r.isPredicativr()==true` provided that each literal factor in the antecedent and consequent is predicative. By definition, an indexical rewrite is automatically predicative.

The compiler parses input inference forms and detects either input `.name` or `.fnctr` input forms for each input `Functor` in a `Rule` or static `Rewrite`. By-name input requires that all predicates and

operators are named at input, that is, the parametric field `-.fnctr==null`. Here is the `byName()` method code from `sam.machine.lang.Term.java`

```
/**
  *  Is this term byName()?
  *  No parametricity.
  *  @version 11/17/2021
   */
public boolean byName() {
   if (this instanceof Functor) {
      Functor f = (Functor)this ;
      if (!(f.fnctr==null)) return false ;  // named
      for (int i=0 ; i < this.arity ; i++) // arguments also
         if (! f.args[i].byName()) return false ;
      return true ;  // named functor with named args
   }
   else return true ; // true for Variable arg by default
}
```

The AutoLogAnalyzer has been updated to accomodate the indexical, predicative and by-name profiles. For example, consider the following sequence of inference inputs.

```
1. f(a)(X)=g(b)(X).
2. f(X)(Y)=g(X)(Y).
3. f(X)=g(X).
4. XvX=X.
5. p(X), q(X)(Y) => goal.
6. p(X), q(a)(Y) => goal.
7. X:T => p(X)(a).
```

The updated AutoLogAnalyzer's summary (at end of detailed report) looks like this.

```
Theory Summary
   theory structural summary
      theory.isIndexical()=false
      theory.isPredicative()=false
      theory.byName()=false
   inference forms structural summaries (Rule/Rewrite)
      1 .isIndexical()=true .isPredicative()=true .byName()=false
      2 .isIndexical()=false .isPredicative()=false .byName()=false
      3 .isIndexical()=true .isPredicative()=true .byName()=true
      4 .isIndexical()=true .isPredicative()=true .byName()=true
      5 .isIndexical()=true .isPredicative()=false .byName()=false
      6 .isIndexical()=true .isPredicative()=true .byName()=false
      7 .isIndexical()=true .isPredicative()=false .byName()=false
```

Refinement of the code Analyzer itself (`sam.machine.lang.AutoLogAnalyzer`) evolves based on results for component testing .   The code tests and Analyzer reports in this section were checked prior to 5/1/2022, using the preliminary indexicality condition for static program rewrite modulators. There is a

new definition (5/12/2022) for `.parapredicative` and `paraPredicative(Term)` in `sam/machine/lang/Rewrite.java` (see §6.4 example2).

6.2 match and substitute methods - `MatchTest` tool and testing ⋇

The `MatchTest` tool — `sam.machine.engine.MatchTest.java` – is a test tool for matching bound terms with terms containing literals, such as is used to match Rule literals with branch facts or to match rewrite modulator Terms with branch fact subterms. The tool also tests a substitution method where the variable bindings from a substitution are used to evaluate another term containing the variables.

The `MatchTest` is designed for convenient usage using the Autolog program `Editor`, wherein the match problem can be entered, and the Autolog `Viewer`, wherein the results are displayed along with match data. A MatchTest input scenario takes the form of an Autolog `Rule`

`match`, ***fixed***, ***flexed*** => ***result***.

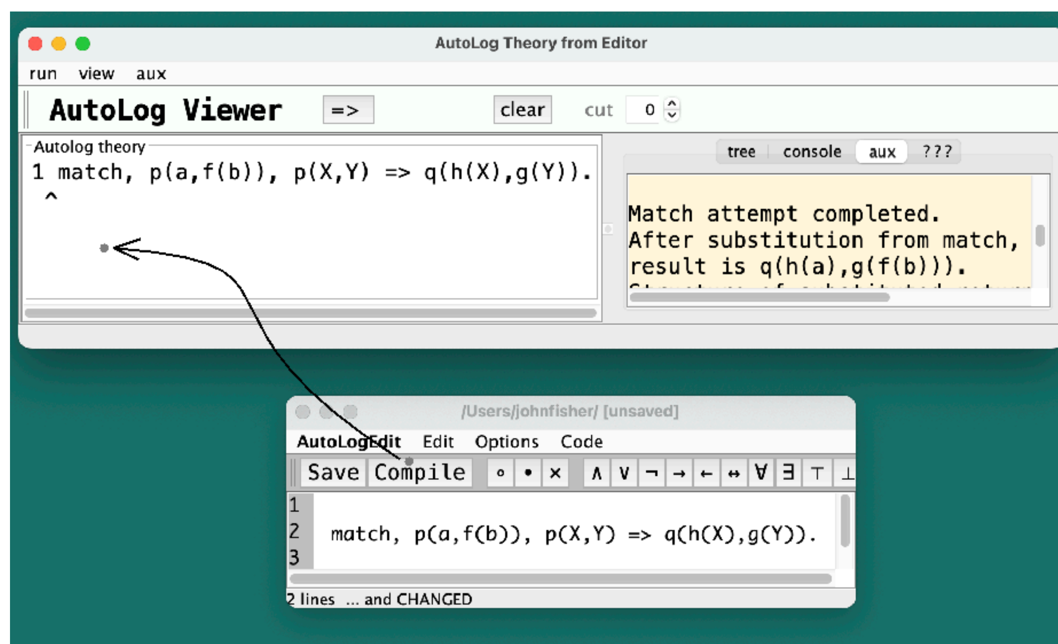
entered into the `Autolog` editor. '`match`' is verbatim, ***fixed*** is any bound term and ***flexed*** is a term allowed to contain variables. The `MatchTest` attempts a match of ***fixed*** and ***flexed*** via a table of variable bindings **T**. The variable bindings table is used to substitute into the term ***result*** and that Is the return value for the MatchTest. The `MatchTest` rule format is used so that the compiler will accept correct Autolog `Term` forms, and use the `AutoLogAnalyzer` to analyze the term structures when the input is compiled into the Autolog Viewer. In the Viewer window, the MatchTest is called using the `aux` toolbar menu.

The `MatchTest` serves two purposes. One is to prototype the matching of `Rule` literals with branch facts. The other is the to prototype the matching of rewrite modulators with subterms of branch facts. Both of these actions then require substitutions to effect Rule inferences or modulator inferences, which is a topic the next two sections, §6.3 §6.3. This is an example of using an emerging system to prototype its own development, where the protype components are eventually incorporated into the target system.

The following examples illustrate the MatchTest actions and various component design issues.

Example 1

```
match, p(a,f(b)), p(X,Y) => q(h(X),g(Y)).
```



Enter this match test into the `AutoLogEditor` and compile it to the `AutoLog Viewer. In the Viewer` execute the `MatchTest` via the aux menu (`shift command M`) and the following report appears in the aux window ...

```
Match fact= p(a,f(b)) with literal= p(X,Y), and
then substitute match bindings into
result = q(h(X),g(Y)).

Match table indices:
X:0 Y:1 => X:0 Y:1

bindings:
0:a
1:f(b)

Match attempt completed.
After substitution from match,
result is q(h(a),g(f(b))).
```

This match test imitates a Rule inference, showing what proposed new fact the consequent literal `q(h(X),g(Y))` would generate using the match for `p(a,f(b)) and p(X,Y)`. Additional Rule inference
tests are presented in §6.3.

Example 2

```
match, a∧b, A∧B => B∧A.
```

This match test generates the following report in the viewer

```
Match fact= a∧b with literal= A∧B, and
then substitute match bindings into
result = B∧A.
Match table indices:
   A:0 B:1 => A:0 B:1

bindings:
   0:a
   1:b

Match attempt completed.
After substitution from match,
result is b∧a.
Structure of substituted return:
< -.name=∧,
  -.fnctr=null,
  -.arity=2,
  -.type=2   >
```

This match test imitates part of the action of a rewrite modulator A∧B=B∧A on the ground term *a*∧b. Additional modulator tests are presented in §6.4

Example 3

```
match, ¬a, F(a) => F(b).
```

This match test illustrates a *forcing* method for matching and substitution

```
Match fact= ¬a with literal= F(a), and
then substitute match bindings into
result = F(b).
Match table indices:
   F:0 => F:0

bindings:
   0:¬<args error for prefix op>

Match attempt completed.
After substitution from match,
result is ¬b.       NOTE FORCING
Structure of substituted return:
< -.name=¬,
  -.fnctr=null,
  -.arity=1,
  -.type=1   >.     NOTE .type
```

How this works is revealed by code listing later in this section.

Example 4

```
match, apply(p)(a), apply(P)(A) => P(A).
```

This example illustrates more *forcing*

```
Match fact= apply(p)(a) with literal= apply(P)(A), and
then substitute match bindings into
result = P(A).
Match table indices:
   P:0 A:1 => P:0 A:1

bindings:
   0:p
   1:a

Match attempt completed.
After substitution from match,
result is p(a).
Structure of substituted return:
```

```
< -.name=p,
  -.fnctr=null,
  -.arity=1,
  -.type=0   >.  NOTE .type
```

This illustrates how the parametric modulator

```
apply(P)(A) = P(A).
```

is intended to behave. The MatchTest is a very handy tool for inspecting match/substitute problems.

Example 5

```
match, apply(apply(p)(a))(b), apply(apply(P)(A))(B) => P(A)(B).
```

Using the previous example as a pattern, first guess what what the match test will do, and then run the match test to check your answer. What parametric rewrite modulator has its action modeled by this match test?

Example 6

```
match, a•b, F(A,B) => F(B,A).
```

Guess the results of a match test, presuming that the functor name would be forced. Then check what the `MatchTest` actually reports. (There might be a warning regarding the forced functor name).

——————

The `MatchTest` development code, which includes the `match` method and the `substitute` method, is located at

```
sam/machine/engine/MatchTest.java
```

in the development code archive.

6.3 `Rule` literal entanglement methods and `TangleTool` testing ⁂

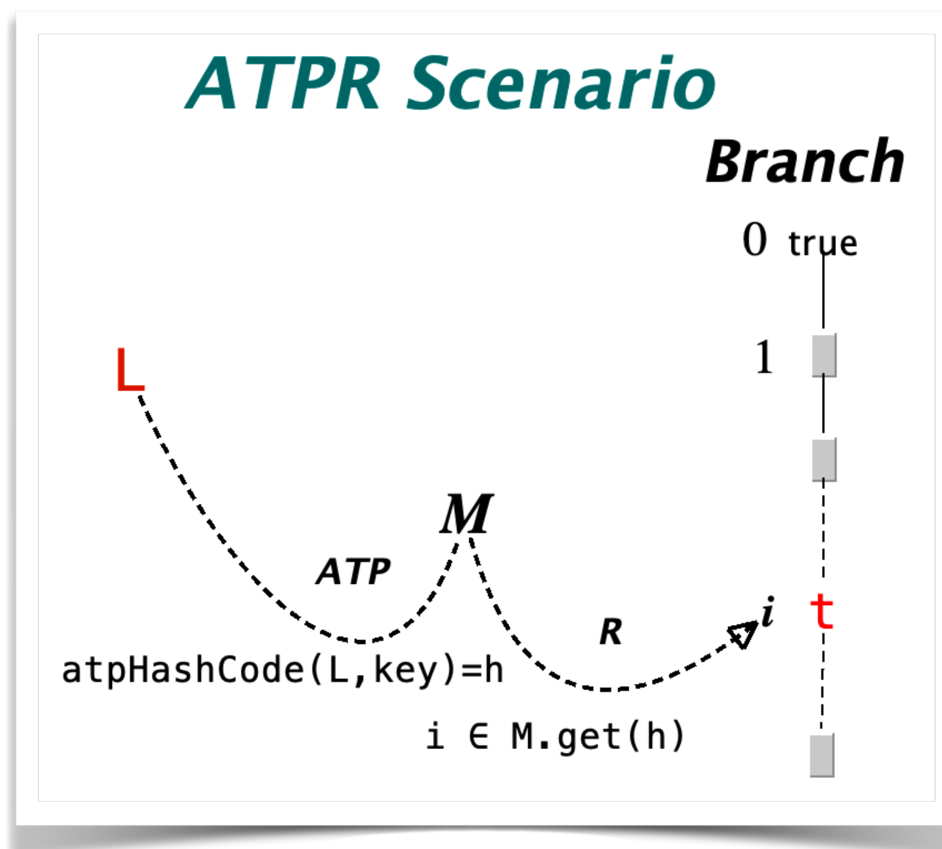
The specific topic of this section is `sam.machine.lang.Rule` literal entanglements with search branch facts. At runtime the corresponding sam.machine.engine.ActiveRule encapsulates methods to find matches for the antecedent of the rule with current branch facts in order to activate (fire) a new inference to conclude new facts in the consequent of the rule. The active rule also marshals the addition of only new facts not currently on the branch.

The overall methodology proposed is similar to that used by the legacy prover `colog14I` (2014, with minor Improvements or corrections since then).

The ***ATPR*** methodology uses an algebraic term structure calculation previously used in the colog14I system (for coherent predicative logic). That method (or design pattern) seems like a good place to start for an AUTOLOG23 adaptation. The overall integration of ATPR methodology with the distributive choices algorithm and inference saturation techniques seems appropriate for Skolem machine implementations where efficient computation of current branch locations for possible matching terms are desired.

The *ATP (algebraic term profile)* part of the *ATPR* computations refers to both the *algebraic* literal antecedent or consequent term profile of an Autolog input rule or the algebraic term profile of a current branch fact contained in the list of the current branch facts of the active Skolem Machine. The *R* — ***retrieval*** — part of *ATPR* refers to algorithms that describe how to *retrieve* the branch facts whose profile could match the inference rule literal profile.

This abstract diagram below illustrates components that are relevant for the ATPR design specifications.



In the diagram, term L refers to a rule literal in an autolog `Rule` having a key profile of bound arguments, as determined by the AutoLogAnalyzer statically on input of the autolog program. The branch fact `t` is a candidate match for L using the ATPR scenario.

The general intention suggested by the diagram is a desire to find a possible or likely match for rule literal L with a branch fact `t` occurring on the current active Autolog branch ***Branch***. This is implemented via a hash map **M** that uses an `atpHashCode(Term,boolean[])` method to compute a hash code value that can be employed to retrieve a list of branch positions sharing the same algebraic term profile.

```
atpHashCode(L,key)=h
i ∈ M.get(h)
```

`M.get(h) is a list` of branch indices (locations) so that `t` is effectively located (retrieved) at branch position i ∈ `M.get(h), where M.get(h)` is the list which contains branch fact candidates having the same algebraic term profile as literal L.

Conversely, the branch fact ***storage*** process that enables retrieval of possible matches for Autolog rule literals involves the assertion of new facts to branch using all possible atpHashCode values that will subsequently be required for retrieval during the next stage of rule inference processing. For `colog14I`, which only allowed (named) predicative rule literals, all of the possible key patterns for the `atpHashCode` computations were determined by the Analyzer for the input program. If an Autolog program does have only predicative rules, then the storage regimen can be the same as that used by colog14I. But, more

generally, Autolog must use some *adaptive* (but less efficient) device to maintain a complete ATPR scenario for nonpredicative rule literals .

Suppose that new fact F is asserted to the current branch at position index p and that K is a list of all possible key patterns (boolean[]) associated with rule literals in the program that could match F. Then for each argument key pattern k in K the following storage operations are performed, assuming that M.get(h) is not empty (otherwise initialize that list first) …

```
atpHashCode(F,k)=h

M.put(h,M.get(h).add(p))
```

which, in effect, adds p to the locations of terms having atpHashCode h at the time when the fact F is asserted to the branch, for the purpose of subsequent retrieval using p ∈ M.get(h).

Examples Using TangleTool to test ATPR methodology

The test tool component sam.machine.engine.TangleTool.java is used to test implementation ideas for both the ATPR methodology describe above, and *active locate, match and replace* technology for active rewrite modulators described in the next section, §6.4.

The format for an Autolog FactMapTest() script has the following *2-rule* appearance

```
true => b1,b2, … ,bn.   The branch
L => fact.              The query
```

The b_i in the consequent of the first rule are used as current branch facts BRANCH[i], and the indexical literal L antecedent of the second rule is the term whose atpHashCode(F,k) is used to retrieve the relevant possible matches in BRANCH. When the script is compiled the Analyzer will detect the factual b_i and the indexical L— otherwise no FactMapTest will be performed.

The FactMapTest() part of TangleTool uses a java.util.concurrent.ConcurrentHashMap

```
protected ConcurrentHashMap<Integer,Vector<Integer>> FactMap ;
```

to store each fact b_i using all possible key patterns k in FactMap :

```
atpHashCode(bi,k)=h
FactMap.put(h,FactMap.get(h).add(i))
```

where care is used to be sure that FactMap.get(h) is initialized.

The code for the atpHashCode is provide here. The shift-left (<<) OFFSET can be modified in the source code in order to test separation of facts in the hash map. Note that .fnctr.index (index assigned by AutoLogAnalyzer for record keeping purposes only) is **NOT** used to compute atpHashCode(s).

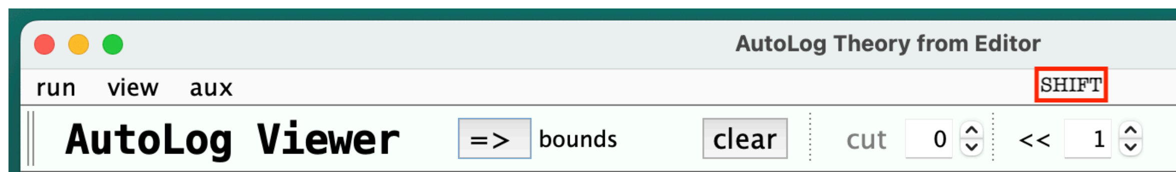
```
/**
*   atpHashCode calculations for a GROUNDED branch fact
*   using a particular bound key pattern.
*   N.B., args terms are Functor terms.
*   Use KEY list as key accelerator in rememberFact.
 */
public static int atpHashCode(Functor f, boolean[] key) {
   int h ;
   if (f.fnctr != null)   // f parametric
      h = atpHashCode(f.fnctr) ;
   else h = f.index ;     // f named
   for (int i = 0 ; i < f.arity ; i++) // use key pattern
      h = (h<< SHIFT) + (key[i] ? atpHashCode(f.args[i]) : 0) ;
   return h ;        //   ^ N.B. NOT f.key[i]
}
/** For embedded ground terms */
public static int atpHashCode(Term t) {
   Functor f = (Functor)t ;
   int hc ;
   if (f.fnctr != null)   // f parametric
      hc = atpHashCode(f.fnctr) ;
   else hc = f.index ;     // f named
   for (int i = 0 ; i < f.arity ; i++) { // can accelerate?
      hc = (hc << SHIFT) + atpHashCode(f.args[i]) ;
   }
   return hc ;
}
```

An algebraic key pattern (`boolean[]`) for a Functor determines which arguments of the Functor are used in the top-level computation of the `atpHashCode` (true means use argument's code, false means use 0 instead). For example, If the fact b_1= `p(a,b,c)` is stored to FactMap then each of the following key patterns is used

```
[false,false,false]
[false,false,true]
[false,true,false]
[false,true,true]
[true,false,false]
[true,false,true]
[true,true,false]
[true,true,true]
```

The reason that this exhaustive key pattern collection is used for TangleTool's FactMapTest() is to *test* completeness (or restriction) issues for FactMap and to study how much overlap there might be (clashing). We will see later in §6.5 that key Patterns can be more restricted by program analysis such as was the case for colog14l using predicative (named) input rules.

The TangleTool is deployed in a fashion similar to the `MatchTest` discussed in §6.2. In order to demonstrate some effects of using different `Shift` values, a java `JSpinner` selector widget has been added to the `ToolBar` of the AutoLog Viewer (marked with the annotation in the snapshot)



If the user selects the SHIFT value on the ToolBar before executing the TangleTest then the selected value will be used for the test. Later in the Autolog development effort, we will make an attempt to automate SHIFT decisions as much as possible but leave the manual decision device for further testing.

Example 1

```
true => p(a,b,c),
        p(c,b,a).

p(a,b,C) => fact.
```

After compiling this test, select SHIFT=3, execute `TangleTest` from the aux menu (or shift command T) in the `Viewer` and then click OK in the dialog. The following report appears in the `aux` window of the `Viewer` ...

```
TangleTool Test Problem
BRANCH
   0 true
   1 p(a,b,c)
   2 p(c,b,a)
maxArity=3
TANGLE TEST:
Find matches for literal p(a,b,C) with branch facts.

atpHashCode SHIFT (<<) value =3
FactMap mappings ↘
{2496=[2], 2368=[1], 2048=[1, 2], 0=[0], 2053=[2], 2501=[2], 2375=[1],
2055=[1], 2416=[1], 2096=[1, 2], 2544=[2], 2101=[2], 2549=[2], 2103=[1],
2423=[1]}

FactMap.keyset().size()=15 (overlap)
Query result:
p(a,b,C)↦2416=[1]
```

Notice that the answer to the query says that BRANCH[1] is a possible match based and the hash code mapping. And, of course with C=c, a match is obtained. Notice that the argument key pattern

`[false,true,false]` would compute the same `atpHashCode` value (an overlap) for both BRANCH[1] and BRANCH[2].

As an exercise, formulate other query literals of SHIFT values for Example 1 and inspect the TangleTool report. E.g., Which (*no match*) `FactMap` candidates are returned for query `p(X,X,X)` and *why* ?

Example 2

```
true => p(a)(b),
        p(b)(c).
p(b)(Y) => fact.
```

The `FactMapTest()` report for `TangleTool, using SHIFT=2` is

```
TangleTool Test Problem
BRANCH
   0 true
   1 p(a)(b)
   2 p(b)(c)
maxArity=2
TANGLE TEST:
Find matches for literal p(b)(Y) with branch facts.

atpHashCode SHIFT (<<) value =2
FactMap mappings ⇲
{0=[0], 117=[2], 104=[1], 108=[2], 111=[1]}

FactMap.keyset().size()=5 ✔ (no overlap)
Query result:
p(b)(Y)↦108=[2]
```

The *argument* key patterns used for both `BRANCH` facts is `[true],[false].`

This tangle test is relevant to much more general situations for indexical programs. For example, consider the following indexical Autolog program (a *real* program and not just a tangle test simulation).

```
true => p(a)(b),
        p(b)(c).
true => p:t→prop, b:t.
P:t→prop, B:t, P(B)(Y) => goal.
```

Now, for this alternate program the active rule component for the 3rd rule will determine that, at match time for `P(B)(Y)`, that P is p and that B is b are choices for `P` and `B` respectively, as was the forced case for the query of example 2, and then proceed to retrieve the possible match facts in the same way that example 2 did ! Thus, the `FactMapTest()` is a significant tool to test what will happen for more general Autolog programs. As mentioned above §6.5, (active inference component designs) will delve into these issues in more detail.

Example 3

```
true => a=b, b=c.
X=Y => fact.
```

This example illustrates *equality literals* in a Rule (and as active rewrite modulators in the next section §6.4). The `FactMapTest()` report for `TangleTool using SHIFT=5` is

```
angleTool Test Problem
BRANCH
   0 true
   1 a=b
   2 b=c
maxArity=2
TANGLE TEST:
Find matches for literal X=Y with branch facts.

atpHashCode SHIFT (<<) value =5
FactMap mappings ⇲
{0=[0], 3072=[1, 2], 3200=[1], 3232=[2], 3077=[1], 3205=[1], 3078=[2],
3238=[2]}

FactMap.keyset().size()=8 (overlap)
Query result:
X=Y↦3072=[1, 2]
```

This example illustrates equality literal entanglement. A realistic example of this would be an Autolog rule for transitivity of equality

```
X=Y, Y=Z => X=Z.
```

Using the test test facts 1 and 2 of example 3 would actively infer `a=c`. In practice Autolog would make the inference to branch as a new fact but also thereafter use `a=c` as an active rewrite modulator in subsequent stages of inference. That specific design topic will be covered in more detail in later sections, §6.4 and §6.5. Again, the reader should try different

Example 4

```
true => p(a,b,c,d,e),
        p(f,g,h,i,j),
        p(1,2,3,4,5),
        p(6,7,8,9,10),
        p(11,12,13,14,15).

p(U,V,W,9,Y) => fact.
```

This example involves how to use the FactMapTest() to explore the robustness of the HashMap storage system with extensive storage density. Note that the example has predicative (named) literals so that extensive key pattern storage of the facts (using all boolean[5]) patterns would not be required in practice (see the comments before example 1 above). But using all possible key patterns allows one to inspect the `FactMap`'s stability. The `FactMapTest()` report for `TangleTool using SHIFT=1` is

```
TangleTool Test Problem
BRANCH
   0 true
   1 p(a,b,c,d,e)
   2 p(f,g,h,i,j)
   3 p(1,2,3,4,5)
   4 p(6,7,8,9,10)
   5 p(11,12,13,14,15)
maxArity=5
TANGLE TEST:
Find matches for literal p(U,V,W,9,Y) with branch facts.


atpHashCode SHIFT (<<) value =1
FactMap mappings ↘
{0=[0], 256=[1, 2, 3], 515=[3], 261=[1], 262=[4], 518=[4], 774=[4], 264=[2], 265=[1, 5], 272=[1], 528=[5], 529=[5], 275=[3], 532=[3], 278=[2], 536=[4], 792=[5], 281=[1], 284=[1], 286=[4], 288=[2], 290=[2], 292=[3, 5], 293=[1], 551=[3], 296=[4], 300=[1], 557=[5], 302=[2], 304=[2], 560=[4], 564=[3], 309=[1], 821=[5], 311=[3], 314=[2], 320=[4], 321=[5], 324=[3], 582=[4], 583=[3], 328=[2], 584=[5], 844=[5], 336=[2, 5], 342=[4], 343=[3], 600=[3], 350=[2], 606=[4], 613=[5], 360=[3], 616=[4], 873=[5], 362=[2], 619=[3], 365=[5], 366=[4], 368=[3], 376=[2], 379=[3], 636=[5], 
128=[1, 2, 3, 4, 5], 384=[4], 640=[4], 387=[3], 900=[5], 390=[2], 392=[5], 137=[1], 142=[2], 144=[1], 402=[2], 147=[3], 404=[3], 662=[4], 152=[4], 408=[4], 153=[1], 665=[5], 154=[2], 156=[1], 157=[5], 416=[2], 929=[5], 164=[3], 165=[1], 421=[5], 423=[3], 168=[2], 424=[2], 172=[1], 174=[4], 430=[4], 686=[4], 176=[1, 2], 436=[3], 692=[5], 181=[1], 438=[2], 183=[3], 184=[5], 185=[1], 444=[5], 190=[2], 192=[1], 448=[4], 704=[4], 450=[2], 196=[3], 198=[4], 454=[4], 455=[3], 201=[1], 202=[2], 204=[1], 208=[1], 464=[2], 721=[5], 213=[1, 5], 215=[3], 472=[3, 4], 216=[2, 4], 728=[4], 217=[1], 473=[5], 220=[1], 224=[1], 736=[5], 229=[1], 230=[2], 232=[3], 233=[1], 491=[3], 236=[1, 5], 494=[4], 750=[4], 496=[3], 240=[4], 242=[2], 500=[5], 245=[1], 251=[3], 252=[1], 765=[5]}

FactMap.keyset().size()=145 (overlap)
Query result:
p(U,V,W,9,Y)↦174=[4]
```

Try other queries and all SHIFT values for this example, comparing results of the tests.

Example 5

```
true => q(1,2,3,4,5,6,7,8,9,10),
        q(1,2,3,4,x,6,7,8,9,10).

q(A,B,C,D,E,F,G,H,I,J) => fact.
```

Try the TangleTool test using SHIFT=1, and we get

```
TangleTool Test Problem
BRANCH
   0 true
   1 q(1,2,3,4,5,6,7,8,9,10)
   2 q(1,2,3,4,x,6,7,8,9,10)
maxArity=10
```

```
TANGLE TEST:
Find matches for literal q(A,B,C,D,E,F,G,H,I,J) with branch facts.


atpHashCode SHIFT (<<) value =3
FactMap mappings ⇲  … 0=[0, 1, 2] …

FactMap.keyset().size()=1536 (overlap)
Query result:
q(A,B,C,D,E,F,G,H,I,J)↦0=[0, 1, 2]
```

What we are witnessing is overflow using `SHIFT=3 (4,5).` Using either `SHIFT=2` or `SHIFT=1` one would have gotten

```
q(A,B,C,D,E,F,G,H,I,J)↦4194304=[1, 2]   with SHIFT=2
q(A,B,C,D,E,F,G,H,I,J)↦4096=[1, 2]      with SHIFT=1
```

Challenge: explain *exactly* why we got the result `0` with `SHIFT=3,4,5.` Hint: look at the `Analyzer` report and notice that index 4 is assigned to the predicate 'q' and then compute atpHashCode value by hand.

—————

The `FactMapTest(AutoLog viewer)` method is located in `sam.machine.engine.TangleTool.java` {line#553} and uses the `FactMap` for fact storage and retrieval actions. Alternate methods for non-predicative theories are described in §6.7.

6.4 `Rewrite` entanglement methods and `TangleTool` testing ⁎

The topic of this section is `Rewrite Lterm` entanglements with branch fact subterms and evaluation of of the resulting rewrite modulations. An *algebraic term profile locator* representation for subterm indexing is proposed and tested.

Three significant historical *references* involving subterm location methods are the following.

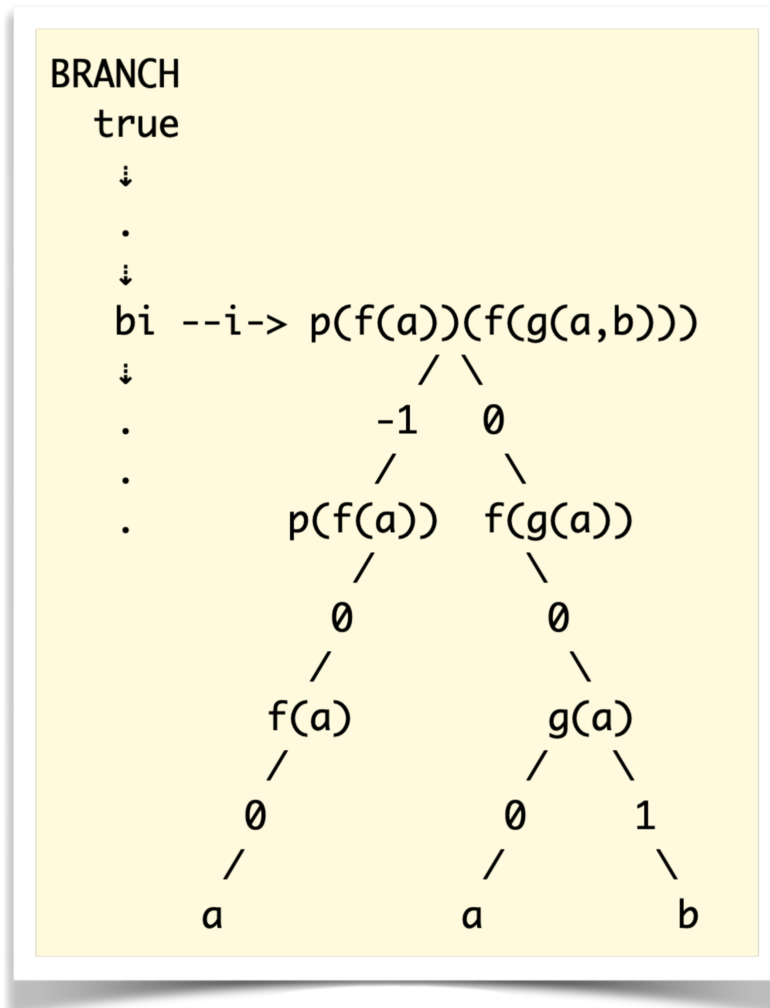
> Bundy, A., Wallen, L. (1984). Discrimination Net. In: Bundy, A., Wallen, L. (eds)
> Catalogue of Artificial Intelligence Tools. Symbolic Computation. Springer, Berlin, Heidelberg.
> https://doi.org/10.1007/978-3-642-96868-6_59
>
> Mark E. Stickel, *The Path-Indexing Method for Indexing Terms*,
> Technical Note 473, SRI International, Oct. 1989.
> (term structure path conceptual description of indexing/retrieval methods)
>
> William McCune, *Experiments with Discrimination-Tree Indexing for Term Retrieval*,
> Journal of Automated Reasoning 9: 147-167, 1992.
> (Otter, Prover9 relevant information)

Although the concepts in all three of these references are intimately involved with the methods presented in this section for Autolog active rewriting, Autolog requires for adaptations that conforms to active inference methods suitable for Skolem Machine scenarios.

The following diagram depicts an *algebra term profile locator* tree (ATPL tree) for the branch fact

```
BRANCH[i] = p(f(a))(f(g(a,b)))
```

The tree shows the locations of the sub terms corresponding to algebraic subterm *locations* `[i,…]` computed as a list of indices: the branch index `i` followed by either a prefix functor (-1) location index or the argument index for the `args[]` component of a subterm.



ATPL Scenario

The list of all subterm locations (`Vector<Integer>`) is as follows

```
@[i]          p(f(a))(f(g(a,b)))
@[i,-1]       p(f(a))
@[i,-1,0]     f(a)
@[i,-1,0,0]   a
@[i,0]        f(g(a))
@[i,0,0]      g(a)
@[i,0,0,0]    a
@[i,0,0,1]    b
```

This scenario is the essential detail needed to establish an *entanglement* of a branch fact's sub terms with a program active rewrite modulator (component `sam.machine.engine.ActiveRewrite` in progress). In this section, we only prototype algorithms for using a rewrite modulator to modulate (rewrite) the branch fact using a sam.machine.lang.Rewrite equation (Lterm = Term) and leave most of the complicated runtime control issues for later sections §8.5-6.7.

Continuing, in the first example we use the hypothetical branch fact to formulate a ModtTest example for TangleTool (sam.machine.engine.TangleTool).

Example 1

```
true => p(f(a))(f(g(a,b))).
f(X)=X.
```

The first rule presents a fact, presumed to be `BRANCH[1]` (following previous tangle tool presumptions). The second inference form should be read as a `sam.machine.lang.Rewrite component`. As previously, compile the little program into the viewer and fire the test in the aux menu (SHIFT COMMAND T), and the `ModTest` method for `TangleTool` reports to the aux window as follows …

```
TangleTool Test Problem
BRANCH
   0 true
   1 p(f(a))(f(g(a,b)))
maxArity=2
TANGLE TEST:
Apply rewrite f(X)=X to BRANCH[1].

Describe subterm locations for [1]p(f(a))(f(g(a,b)))
p(f(a))(f(g(a,b)))  @[1]
p(f(a))  @[1, -1]
f(a)  @[1, -1, 0]
a  @[1, -1, 0, 0]
f(g(a,b))  @[1, 0]
g(a,b)  @[1, 0, 0]
a  @[1, 0, 0, 0]
b  @[1, 0, 0, 1]
-----------------

Lterm key profile=[false]
Rewrite Variable index: X:0 = X:0
# match locations=2
Locations, bindings and modulants:
@[1, -1, 0]:
  f(a) = a
modulant= p(a)(f(g(a,b)))
{ BRANCH[1]= p(f(a))(f(g(a,b))) }    ✓ was copied
----------
@[1, 0]:
  f(g(a,b)) = g(a,b)
modulant= p(f(a))(g(a,b))
{ BRANCH[1]= p(f(a))(f(g(a,b))) }    ✓ was copied
----------
```

The first calculation computes all of the possible subterms and their locations for `BRANCH[1]`. The second group of calculations locates all possible matches for the `Lterm` of the modulator and then modulates `BRANCH[1]` producing a sufficiently copied modulant so as not to alter `BRANCH[1]` for later inferences as runtime inferences continue. The relevant method codes are given after some more examples.

Example 2

```
true => p(f(a)(b)(c)).
f(X)(Y)(Z) = g(X,Y,Z).
```

The argument for `BRANCH[1]` is predicative, but the `Lterm` for the modulator is `paraPredicative`. A `Rewrite` r is parapredicative (`r.parapredicative == true`) provided that `paraPredicative(r.Lterm)` returns true. A `Term` is `paraPredicative` provided that it is a `Functor` and that it is either (named) `predicative` or its parametric part is a `paraPredicative Term` (recursively via `.fnctr`). The code specifications are given in `sam/machine/lang/Rewrite.java` (added 5/12/2022).

The `ModTest` for `TangleTool` reports as follows ….

```
TangleTool Test Problem
BRANCH
   0 true
   1 p(f(a)(b)(c))
maxArity=3
TANGLE TEST:
Apply rewrite f(X)(Y)(Z)=g(X,Y,Z) to BRANCH[1].

Describe subterm locations for [1]p(f(a)(b)(c))
p(f(a)(b)(c))  @[1]
f(a)(b)(c)  @[1, 0]
f(a)(b)  @[1, 0, -1]
f(a)  @[1, 0, -1, -1]
a  @[1, 0, -1, -1, 0]
b  @[1, 0, -1, 0]
c  @[1, 0, 0]
-----------------

Lterm key profile=[false]
Rewrite Variable index: X:0 Y:1 Z:2 = X:0 Y:1 Z:2
# match locations=1
Locations, bindings and modulants:
@[1, 0]:
  f(a)(b)(c) = g(a,b,c)
modulant= p(g(a,b,c))
{ BRANCH[1]= p(f(a)(b)(c)) }
----------
```

Example 3

```
true =>apply(q)(a).
apply(Q)(A)=Q(A).  // application casting

TangleTool Test Problem
BRANCH
   0 true
```

```
   1 apply(q)(a)
maxArity=2
TANGLE TEST:
Apply rewrite apply(Q)(A)=Q(A) to BRANCH[1].

Describe subterm locations for [1]apply(q)(a)
apply(q)(a)  @[1]
apply(q)  @[1, -1]
q  @[1, -1, 0]
a  @[1, 0]
-----------------

Lterm key profile=[false]
Rewrite Variable index: Q:0 A:1 = Q:0 A:1
# match locations=1
Locations, bindings and modulants:
@[1]:
  apply(q)(a) = q(a)
modulant= q(a)
{ BRANCH[1]= apply(q)(a) }
----------
```

Example 4

```
true => apply(apply(p)(a))(b).
apply(apply(P)(A))(B)=P(A)(B).  // parametric casting

TangleTool Test Problem
BRANCH
   0 true
   1 apply(apply(p)(a))(b)
maxArity=2
TANGLE TEST:
Apply rewrite apply(apply(P)(A))(B)=P(A)(B) to BRANCH[1].

Describe subterm locations for [1]apply(apply(p)(a))(b)
apply(apply(p)(a))(b)  @[1]
apply(apply(p)(a))  @[1, -1]
apply(p)(a)  @[1, -1, 0]
apply(p)  @[1, -1, 0, -1]
p  @[1, -1, 0, -1, 0]
a  @[1, -1, 0, 0]
b  @[1, 0]
-----------------
Lterm key profile=[false]
Rewrite Variable index: P:0 A:1 B:2 = P:0 A:1 B:2
# match locations=1
Locations, bindings and modulants:
@[1]:
```

```
  apply(apply(p)(a))(b) = p(a)(b)
modulant= p(a)(b)
{ BRANCH[1]= apply(apply(p)(a))(b) }
----------
```

Example 5

```
true => a∧b=c.
X∧Y=Y∧X.

TangleTool Test Problem
BRANCH
   0 true
   1 (a∧b)=c
maxArity=2
TANGLE TEST:
Apply rewrite X∧Y=Y∧X to BRANCH[1].

Describe subterm locations for [1](a∧b)=c
(a∧b)=c  @[1]
a∧b  @[1, 0]
a  @[1, 0, 0]
b  @[1, 0, 1]
c  @[1, 1]
-----------------

Lterm key profile=[false,false]
Rewrite Variable index: X:0 Y:1 = X:0 Y:1
# match locations=1
Locations, bindings and modulants:
@[1, 0]:
  a∧b = b∧a
modulant= (b∧a)=c
{ BRANCH[1]= (a∧b)=c }
----------
```

Example 6

```
true => a=b, a=d.
b=c.

TangleTool Test Problem
BRANCH
   0 true
   1 a=b
   2 a=d
maxArity=2
TANGLE TEST:
Apply rewrite b=c to BRANCH[1].

Describe subterm locations for [1]a=b
```

```
a=b  @[1]
a  @[1, 0]
b  @[1, 1]
-----------------

Lterm key profile=[]
Rewrite Variable index:  =
# match locations=1
Locations, bindings and modulants:
@[1, 1]:
  b = c
modulant= a=c
{ BRANCH[1]= a=b }
----------
```

Notice that the new fact (modulant) `a=c` should also create a new `ActiveRewrite` which would in turn apply to BRANCH[2] in order to generate another new fact `c=b` and another new `ActiveRewrite`. This action is in addition to the direct modulator actions discussed in this section and one of the topics for §6.5 & 6.6.

—————

The `ModTest(AutoLog viewer)` method is located in

`sam.machine.engine.TangleTool.java` (line#581)

which also contains method `modulate(BRANCH[1])` {line#614} , and methods `locate`, `match` and `substitute` (same codes as in `MatchTest.java`).
See `sam/machine/engine/TangleTool.java` in the development code archive.

6.5 ActiveRewrite completeness and soundness issues ⁎

An `ActiveRewrite` is a Callable runtime rewrite inference modulator process. Any sam.machine.lang.Rewrite component which is specified in the autolog program generates an an `ActiveRewrite` used to modulate inference branch facts (as specified and tested in §6.4). In addition, ground equality literals inferred in the consequent of an auto log Rule also generates an `ActiveRewrite` after it is asserted to the current inference branch. These actions will be discussed in more detail after an introduction to an important `ActiveRewrite` modulator issue, namely …

ActiveRewrite *equality completeness issues*

In order to explain the intended behaviors for an `ActiveRewrite`, the following simple examples are provided as an explanation regarding how rewrite actions are able to "complete equality facts" in order to effect *symmetry* and *transitivity* of equality, without any need for the inclusion of corresponding *rules* in an autolog program/theory.

This rewrite strategy does however require some way to introduce *reflexivity* in an effective and indexical manner.

We use `colog14I` , appropriately adapted, to illustrate the special rewrite behaviors. In order to do that, the rewrite modulations in `colog14I` are put into effect using `colog` rules rather than a rewrite modulation inference action that will actually be used with autolog. In addition to that detail, the

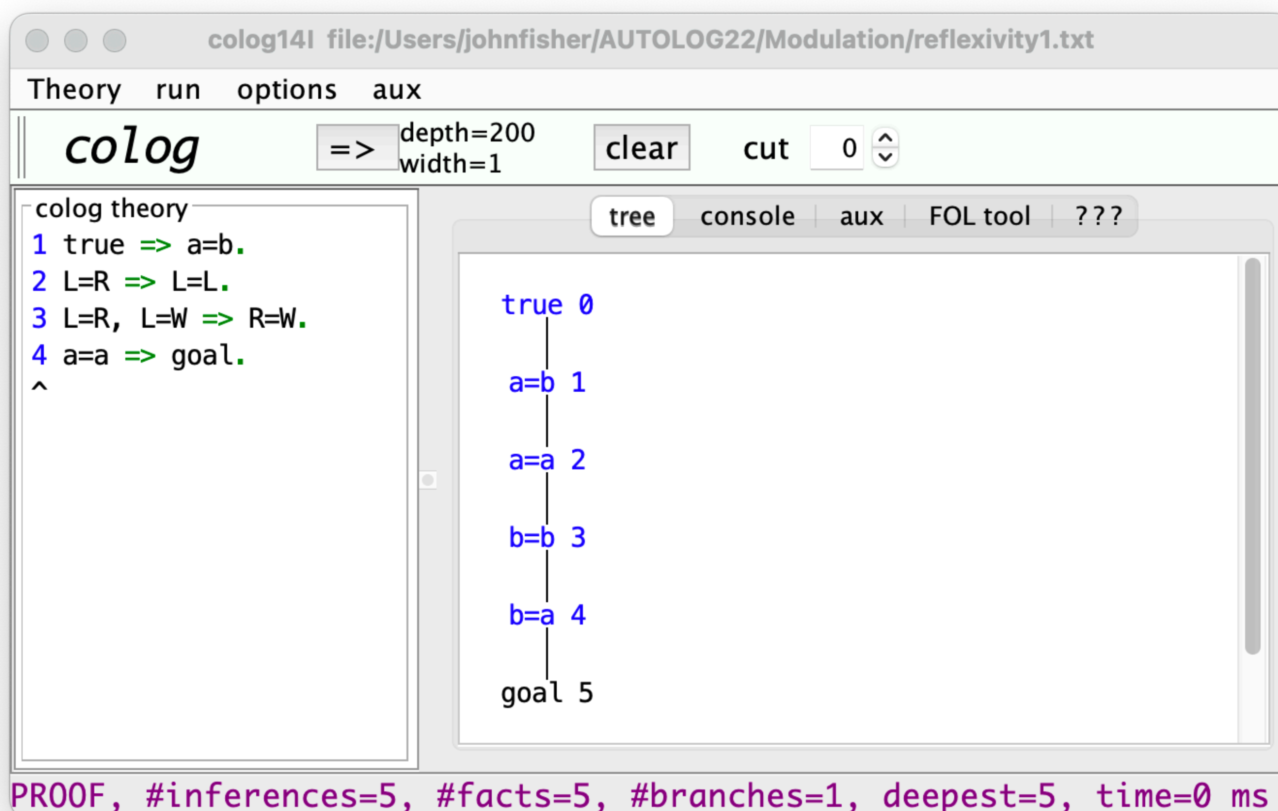
colog14I simulations illustrate how the parent prover can help to model development of the new autolog prover system.

Reflexivity examples

Consider the colog theory reflexivity1.txt

```
true => a=b.
L=R => L=L.          // reflexivity of L rule
L=R, L=W => R=W.  // L=R modulation of L=W
a=a => goal.         // reflexivity of a
```

When this program is loaded into colog14I and a proof is attempted, one sees the following display



The proof extracted via the aux menu looks like this (read it from bottom to top) …

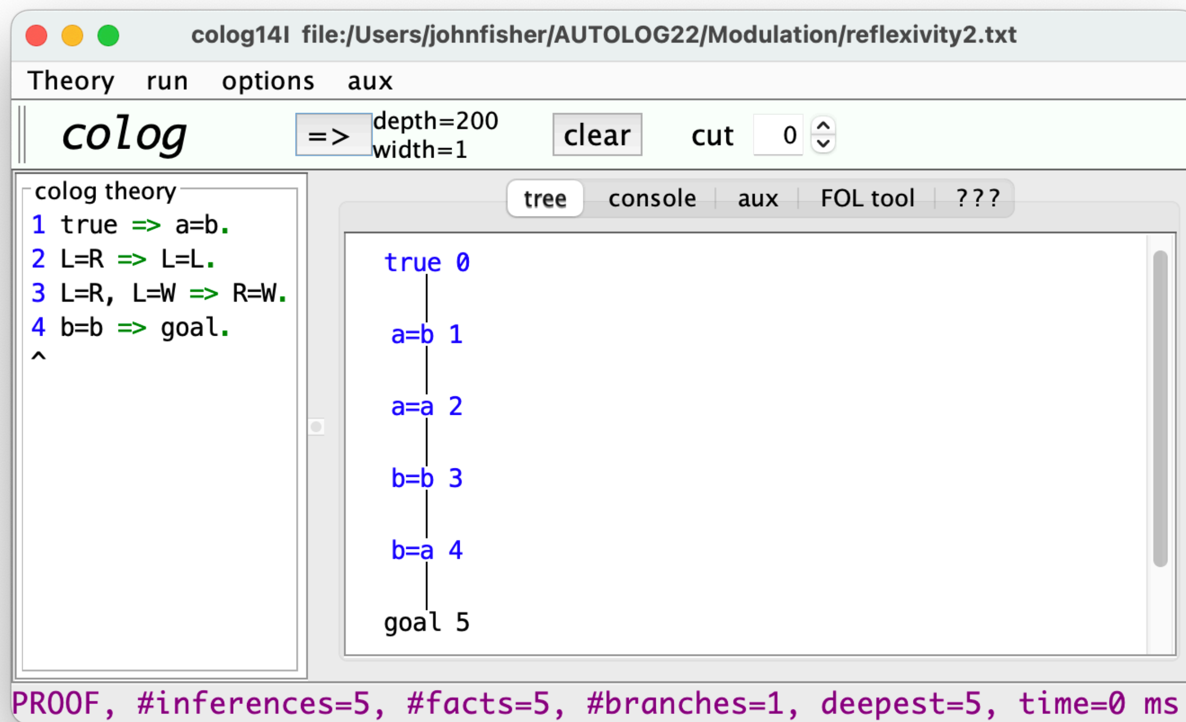
```
LEAF 5.
@4, rule4: a=a => goal
@1, rule2: a=b => a=a
@0, rule1: true => a=b
```

The reflexivity rule#2 infers reflexivity of the Lterm of the Rewrite, with no modulation rewrite required.

Now Consider the colog theory reflexivity2.txt

```
true => a=b.
L=R => L=L.       // reflexivity of L rule
L=R, L=W => R=W.  // L=R modulation of L=W
b=b => goal.      // reflexivity for b
```

When this program is loaded into colog14I and a proof is attempted, one sees the following display



The proof extracted via the aux menu looks like this (read it from bottom to top) …

```
LEAF 5.
@4, rule4: b=b => goal
@2, rule3: a=b, a=b => b=b
@0, rule1: true => a=b
```

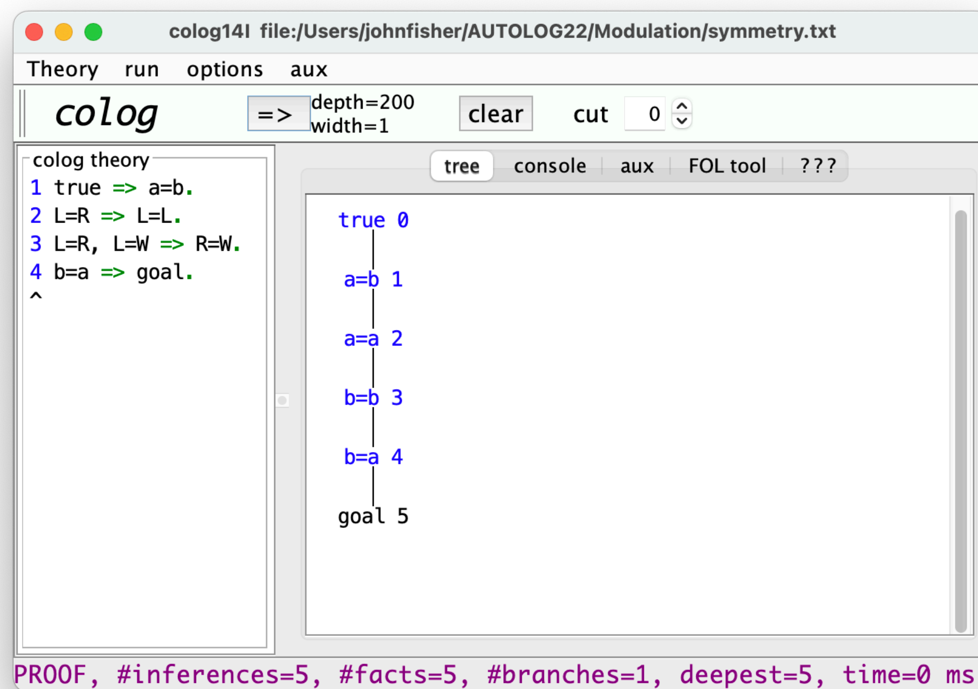
Notice that only the modulation rule#3, but not the reflexivity rule#2, is required for the proof. That is, when the rewrite modulator a=b rewrites the fact a=b, one gets b=b. In this `colog14I` simulation, the rewrite rule was applied to the initially asseredt fact. For autolog the corresponding ActiveRewrite will be *generated* when the fact is asserted in order to be available for next-stage inferencing!

Symmetry example

Consider the colog program `symmetry.txt`

```
true => a=b.
L=R => L=L.          // reflexivity of L rule
L=R, L=W => R=W.     // L=R modulation rewrite of L=W as rule
b=a => goal.
```

When this program is loaded into colog14I and a proof is attempted one sees the following display



The proof extracted via the aux menu looks like this (read it from bottom to top) …

```
LEAF 5.
@4, rule4: b=a => goal
@3, rule3: a=b, a=a => b=a
@1, rule2: a=b => a=a
@0, rule1: true => a=b
```

Note that both the reflexivity rule and the modulation rule are required for the proof.

Transitivity example

Consider the colog program `transitivity.txt`

```
true => a=b, b=c.
L=R => L=L.        // reflexivity of L rule
L=R, L=W => R=W.   // L=R modulation rewrite of L=W as rule
a=c => goal.
```

As the reader can hopefully anticipate, the proof will be achieved via two rewrite modulations …

colog14l file:/Users/johnfisher/AUTOLOG22/Modulation/transitivity.txt

Theory run options aux

colog => depth=200 width=1 clear cut 0

colog theory

```
1 true => a=b, b=c.
2 L=R => L=L.
3 L=R, L=W => R=W.
4 a=c => goal.
^
```

tree console aux FOL tool ???

```
true 0
a=b 1
b=c 2
a=a 3
b=b 4
b=a 5
c=c 6
c=b 7
c=a 8
a=c 9
goal 10
```

PROOF, #inferences=9, #facts=10, #branches=1, deepest=10, time=0...

The proof achieved is

```
LEAF 10.
@9, rule4: a=c => goal
@8, rule3: b=a, b=c => a=c
@4, rule3: a=b, a=a => b=a
@2, rule2: a=b => a=a
@0, rule1: true => a=b, b=c
```

using both reflexivity rule#1 and rewrite rule#3 twice.

The examples so far illustrate the possible need to effectively employ some kind of reflexivity rule. The one used in the previous examples is sort of a neutral approach: `L=R => L=L.` There could be other versions of such a rule, for example a rule passed upon a type specification: `X:T => X=X.` Another scenario could use some predication: `p(X) => X=X.` That issue is not expanded upon in this section. Suffice it so say here, that an ActiveRewrite is a prototype for implementation of equality types using rewrite modulators. For now, a witness for an equality type would be the application of an active rule or the application of an active rewrite which produces equality facts on the current inference branch, but at

this time no autolog language specification for such witnesses is proposed, that being a somewhat tricky issue (so later with that).
The following additional examples explore some proof complexity issues that can arise from the use of active rewrite modulators together with cooperating rules.

Proof complexity examples

Consider the colog program `mod1.txt`

```
true => a=b, b=c, p(a).
L=R => L=L.          // reflexivity for L
L=R, L=Z => R=Z.     // modulate equality
p(L), L=R => p(R).  // modulate p(L)
p(c) => goal.
```

Notice that the *modulate p(L)* 4th rule applied twice should effect a quick proof. However, the program has two other rules (2nd and 3rd) which colog14I will attempt first. The proof generated via the aux menu of `colog14I` actually looks like this …

```
colog14I  file:/Users/johnfisher/SkolemMachines.ORG/adn_pages/§6.5Modulation/mod1.txt
Theory   run   options   aux
colog   =>   depth=200 width=1   clear   cut 0
colog theory
1 true => a=b, b=c, p(a).
2 L=R => L=L.
3 L=R, L=Z => R=Z.
4 p(L), L=R => p(R).
5 p(c) => goal.
^
tree   console   aux   FOL tool   ???
true 0
a=b 1
b=c 2
p(a) 3
a=a 4
b=b 5
b=a 6
c=c 7
c=b 8
c=a 9
a=c 10
p(b) 11
p(c) 12
goal 13
PROOF, #inferences=11, #facts=13, #branches=1, deepest=13, time=0 ms
```

The proof achieved is …

```
LEAF 13.
@12, rule5: p(c) => goal
@11, rule4: p(a), a=c => p(c)
@9, rule3: b=a, b=c => a=c
@5, rule3: a=b, a=a => b=a
@3, rule2: a=b => a=a
@0, rule1: true => a=b, b=c, p(a)
```

… which is not the quick substitution proof possible via the 4th rule. The proof we got threads first through a lemma showing that a=c, and then substitutes c for a in p(a).

EXERCISE: Reorder the rules of mod1.txt so that we get the shorter proof that deduces the goal in two applications of the 4th rule of mod1.txt. (Alternatively, eliminate rule 2.)

The issue of the effects of inference orders for autolog are discussed again later in the notebook.

Now Consider the colog program mod2.txt

```
true => a=b, p(b).
p(a) => goal.
L=R => L=L.             // reflexivity for L
L=R, L=Z => R=Z.        // modulate equality
p(L), L=R => p(R).      // modulate p(L)
```

Since the b=a (symmetry) is a required lemma to prove the goal, we expect that lemma to be part of the proof, which is indeed the case.

colog14l file:/Users/johnfisher/SkolemMachines.ORG/adn_pages/§6.5Modulation/mod2.txt

Theory run options aux

colog => depth=200 width=1 clear cut 0

colog theory
```
1 true => a=b, p(b).
2 p(a) => goal.
3 L=R => L=L.
4 L=R, L=Z => R=Z.
5 p(L), L=R => p(R).
^
```

tree console aux FOL tool ???

```
true 0
a=b 1
p(b) 2
a=a 3
b=b 4
b=a 5
p(a) 6
goal 7
```

PROOF, #inferences=6, #facts=7, #branches=1, deepest=7, time=0 ms

And the extracted proof is

```
LEAF 7.
@6, rule2: p(a) => goal
@5, rule5: p(b), b=a => p(a)
@4, rule4: a=b, a=a => b=a     << NB
@2, rule3: a=b => a=a
@0, rule1: true => a=b, p(b)
```

Note that symmetry (b=a) is computed first (lemma) and then used to rewrite p(b) as p(a).

As last example illustrating some aspects for proof complexities, consider mod3.txt

```
true => f(a,b)=c, c=d.
true => f(a,b)=g(b,a).
g(b,a)=d => goal.
X=Y=> X=X.        // reflexivity rule for L
X=Y, X=W => Y=W.  L=R modulation rewrite of L=W as rule
```

Here is a snapshot of a colog14l inference tree

```
colog14l  file:/Users/johnfisher/SkolemMachines.ORG/adn_pages/§6.5Modulation/mod3.txt
Theory   run   options   aux
colog   =>   depth=200 width=1   clear   cut 0

colog theory
[1] true => f(a,b)=c, c=d.
[2] true => f(a,b)=g(b,a).
3 g(b,a)=d => goal.
4 X=Y => X=X.
5 X=Y, X=W => Y=W.
^

tree   console   aux   FOL tool   ???

true 0
f(a,b)=c 1
c=d 2
f(a,b)=f(a,b) 3
c=c 4
c=f(a,b) 5
d=d 6
d=c 7
d=f(a,b) 8
f(a,b)=d 9
f(a,b)=g(b,a) 10
c=g(b,a) 11
d=g(b,a) 12
g(b,a)=c 13
g(b,a)=g(b,a) 14
g(b,a)=f(a,b) 15
g(b,a)=d 16
goal 17

PROOF, #inferences=16, #facts=17, #branches=1, deepest=17, time=0 ms
```

And the corresponding colog14l proof

```
LEAF 17.
@16, rule3: g(b,a)=d => goal
@15, rule5: f(a,b)=g(b,a), f(a,b)=d => g(b,a)=d
@9, rule2: true => f(a,b)=g(b,a)
@8, rule5: c=f(a,b), c=d => f(a,b)=d
@4, rule5: f(a,b)=c, f(a,b)=f(a,b) => c=f(a,b)
@2, rule4: f(a,b)=c => f(a,b)=f(a,b)
@0, rule1: true => f(a,b)=c, c=d
```

EXERCISE: Find a colog14l proof that does not use rule4, the reflexivity rule.

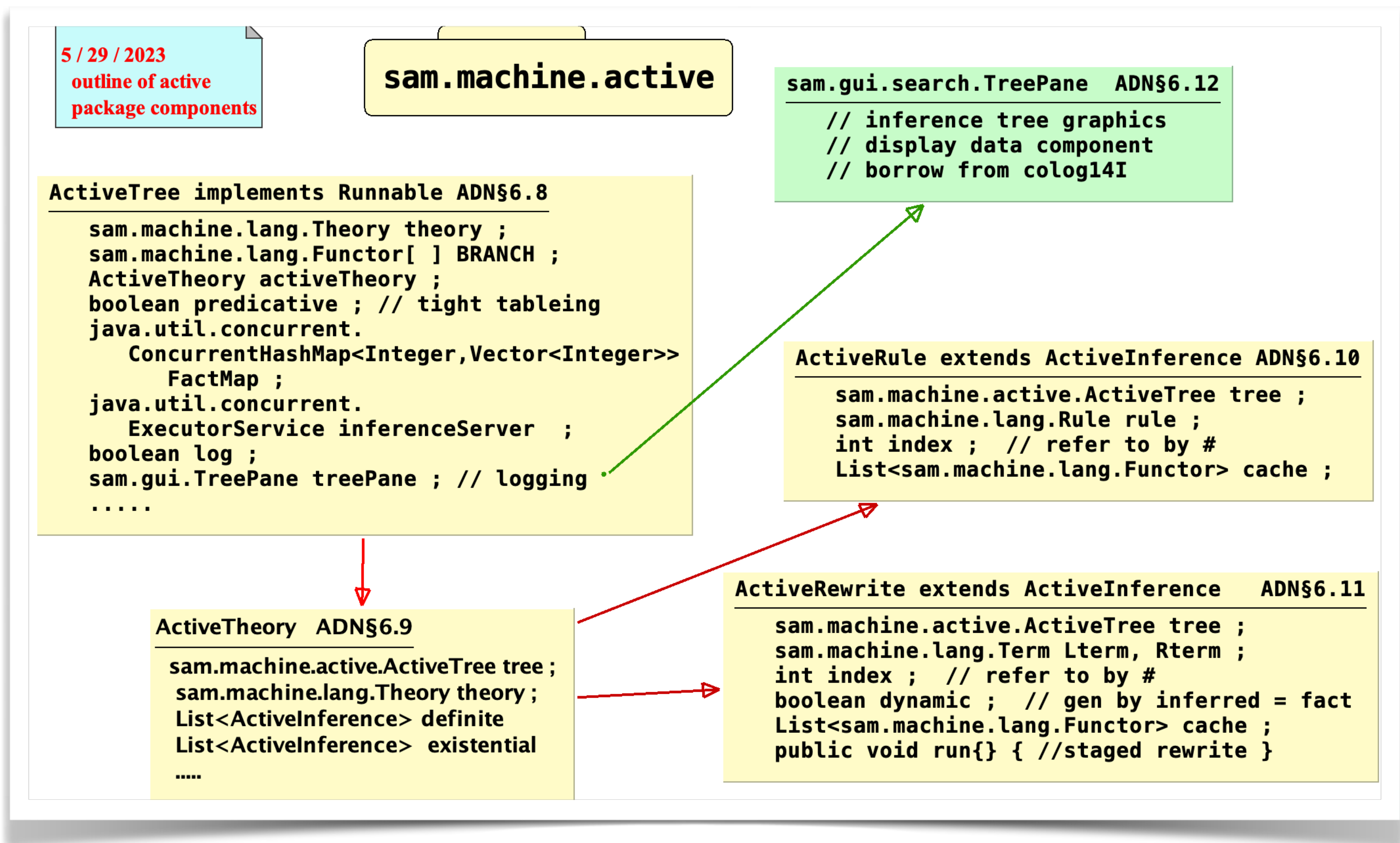
These colog14l simulations for some behaviors for active rewrite modulations intended for autolog will be revisited later when the autolog active rewrite components, and support for same, are finished.

ActiveRewrite *equality soundness issues*

<< to be continued >>

6.6 Active Inference Design Scenarios ※

The following active package class diagram shows the proposed active components specified in the `sam.machine.active` package and shows some general relationships between the classes.



Some details shown by the diagram will likely change over time as the implementation codes are written, tested, and refined. The diagram is `dated` for the current version.

The `ActiveTree` is the primary active component for staged inference control, as characterized abstractly in §5.7. The details for inference control will be described in §6.8. The `ActiveTree` decides when an inference branch terminates, based upon the inferences of `ActiveRule`(s) or `ActiveRewrite`(s), and controls the pursuit of a new branch initiated by a disjunctive consequence for an `ActiveRule` earlier along the terminating branch. The depth-first branching pursuit control is also decided by the `ActiveTree inference manager`.

The `ActiveTheory` component stores the `ActiveRule` and `ActiveRewrite` components that perform the rule inference actions or rewrite modulation actions during each active inference stage. The code details for `ActiveTheory` design are described in §6.9.

The `ActiveRule` and `ActiveRewrite Runnable` components act concurrently during each stage of inference. Details for `ActiveRule` action design are described in §6.10 and details for `ActiveRewrite` action design are described in §6.11. Both of these active components use ***internal caches*** to hold the results of their actions. Between stages, these caches are used to add ***new*** generated facts to the current branch. An existential ActiveRule component is activated when definite components have saturated the current branch. The strategy mimics the *round-robin strategy* used in colog14I, in order to assure fairness. (An existential rule has a free variable in its `consequent`.)

The inference tree data computed by these scenarios is encapsulated by logging results in a sam.gui.TreePane §6.12, which maintains a colog14l kind of graphic tree data model for the inference actions.

The five aspects of action described above are similar to those implemented in colog14l, except that colog14l had no active rewrites and used inference rules for equality reasoning.The primary action differences involve the use of ***staged concurrent inference*** actions intended for AUTOLOG24.

6.7 sam.machine.engine.ActiveMod concurrency testing tool ⁂

The general concurrency relationships in the diagram were outlined in early development of autolog (circa 2018), at which time it was presumed that java.util.concurrent components available at that time would be employed to compute concurrent inferences by ActiveRule(s) and ActiveRewrite(s) using the InferenceManager via the java.util.concurrent.ExecutorService.

In this section a *test* component sam.machine.engine.ActiveMod is provided to illustrate the concurrent computation of many ActiveRewrite(s) (fact modulations) .

The *OpenJDKs Project Loom* perspective on virtual threads has created a good opportunity to test the Java implementation of virtual threads afforded by using *preview features* of Java19. This means that the ExecutorService component inference_server could be an incarnation of newVirtualThreadPerTaskExecutor() which could be adapted for use as a kind of *staged virtual inference mechanism*.

The following links provide a convenient overview for the new virtual threads and Java19:

Virtual Threads: New Foundations for High-Scale Java Applications

OpenJDK JEP 425: Virtual Threads (Preview)

——————

We show the file listing for sam.machine.engine.ActiveMod.java below. The ActiveMod class itself borrows the TangleTool's Modtest components discussed in §6.4. The main method for ActiveMod constructs and then concurrently executes many ActiveMod class instances in order to gather run times.

```
package sam.machine.engine ;

import sam.machine.lang.* ;
import java.util.* ;
import java.util.concurrent.* ;
import java.time.Duration;
import java.time.Instant;

/**
  *  ActiveMod
  *     dev tests for ActiveRewrite component
  *  using Java 19 preview
  *   --release 19 --enable-preview < compile switches
  *   --enable-preview             < run switch
  *  Runnable w/internal cache of modulants
  *  @version 1/1/2022
   */
```

```
public class ActiveMod implements Runnable {
   // Use rewrite to modulate fact
   public Rewrite rewrite ;
   public Functor fact ;
   public Vector<Functor> cache ;
   public Functor left  ;
   public Term    right ;

   /**
     * Construct an ActiveMod task from string, e.g.:
     * "true => p(f(1,2),f(2,3)). f(X,Y) = X." ;
     *             ^fact                ^rewrite
     *                             ^left   ^right
     * A TangleTool test form.
      */
   public ActiveMod(String modtest) {
      Theory theory = Theory.translate(modtest) ;
      AutoLogAnalyzer.analyze(theory) ; // indexing
      this.cache = new Vector<Functor>() ;
      this.fact =
         ((((Rule)theory.elementAt(0)).consequent).elementAt(0)).elementAt(0) ;
      this.rewrite = (Rewrite)theory.elementAt(1) ;
      this.left = (Functor)rewrite.Lterm ; // cast
      this.right= rewrite.Rterm ;
   } // end constructor

   /**
     * Employ TangleTool (✓) static methods
     * to create modulants.
     */
   public void run() { // perform the modulations
      Vector<Integer> path = new Vector<Integer>() ;
      path.add(1) ;
      Functor[] varbind = new Functor[rewrite.U] ;
      Vector<Vector<Integer>> locs = new Vector<Vector<Integer>>() ;
      Vector<Functor[]> bindings = new Vector<Functor[]>() ;
      TangleTool.locate(left,rewrite.U,fact,path,locs,bindings) ;  // ✓
      for (int i=0 ; i<locs.size() ; i++) {
         //Functor lterm =
         // (Functor)TangleTool.substitute(left,bindings.elementAt(i)) ; // ✓
         Functor rterm =
            (Functor)TangleTool.substitute(right,bindings.elementAt(i)) ; // ✓
         Vector<Integer> rpath = locs.elementAt(i) ; // replacement path
         /////  modulate // ✓
         Functor modulant =(Functor)TangleTool.modulate(fact,0,rpath,rterm) ;
         this.cache.add(modulant) ;
      }
   } // end run

   ///////////////////////////////////////////////
   /////      Concurrency Executors examples
   /////    Boilerplate executor code for tests
   ///////////////////////////////////////////////
   /**  -- MAIN --
     * Compare   Executors.newFixedThreadPool(-)
     *      vs   Executors.newVirtualThreadPerTaskExecutor()
     * running concurrent list of ActiveMOD tasks
      */
   public static void main(String[] args) {
      ///// create list of independent ActiveMod tasks
      int N = 10_000 ;  // number of ActiveMod tasks to time
```

```
      Vector<ActiveMod> tasks = new Vector<ActiveMod>() ;
      for (int n=0 ; n<N ; n++)   // USER EDITS REWRITE TEST PROBLEM
         tasks.add(new ActiveMod("true => p(f(1,2),f(2,3)). f(X,Y) = X.")) ;
      System.out.println(tasks.size()+" tasks") ;

      ///// construct executor, CHOOSE:
      ExecutorService executor = Executors.newWorkStealingPool() ;
//      ExecutorService executor = Executors.newFixedThreadPool(10) ;
//      ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor() ;
//             --release 19 --enable-preview

      ///// submit all tasks to executor
      Instant start = Instant.now();
      try{
         for(int n=0 ; n<N ; n++)
            executor.submit((ActiveMod)(tasks.elementAt(n))) ;
      } catch(Exception e) { System.out.println(e) ; }

      ///// shutdown and await termination
      executor.shutdown() ;    // needs to finish, so
      try {          // added for virtual thread version
           if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
               executor.shutdownNow();
          }
      } catch (InterruptedException ex) {
           executor.shutdownNow();
           Thread.currentThread().interrupt();
      }

      ///// report
      Instant finish = Instant.now();
      System.out.println("shutdown " + executor.isShutdown()) ;
      System.out.println("terminated " +executor.isTerminated()) ;
      long timeElapsed = Duration.between(start, finish).toMillis();
      System.out.println("Total elapsed time : " + timeElapsed+"ms");
      ///// spy on some task cache
      Vector<Functor> cache = tasks.elementAt(5656).cache ;
      for (int i=0 ; i < cache.size() ; i++)
         System.out.println("cache 5656@"+i+": "+ cache.elementAt(i)) ;
      int p = Runtime.getRuntime().availableProcessors() ;
      System.out.println(p+" #processors") ;

   } // end main
} // end ActiveMOD
```

Some sample test results are listed as *end notes* in the code file …

/* end notes
ActiveMod does not locate multiple possible matches on a long branch
stage nor assert new facts -- which is what the
autolog sam.machine.engine.ActiveRewrite component would also do.

iMacPro 10_000   36 #processors   (18 core Xeon W)
24ms   Executors.newWorkStealingPool()   WSP
33ms   newFixedThreadPool(10)   FTP(10)
51ms   newVirtualThreadPerTaskExecutor() VTPT

For WorkStealingPool with N=1_000_000, Total elapsed time : 767ms --- AMAZING.
Seems like perhaps WSP takes a few seconds to terminate(?)

```
MacBookAir 10_000   4 #processors    (2core i5)
51ms  WSP
64ms  FTP(10)
175ms VTPT

MacBookPro 10_000    12 #processors  (6core i7)
19ms  WSP
27ms  FTP(10)
47ms  VTPT
*/
```

The test output from a run looks like this …

```
10000 tasks
shutdown true
terminated true
Total elapsed time : 31ms
cache 5656@0: p(1,f(2,3))
cache 5656@1: p(f(1,2),2)
36 #processors
```

The `ActiveMod` test is somewhat limited, running MANY constructed instances of the same rewrite problem. One can run different rewrite problem, by editing the test problem. One can include other `ExecutorServices` to the test routines. All tests so far seem very quick for what amounts to very many concurrent *primitive recursive algebraic logic calculations.* Of course, one gets slight timing variations over multiple executions.

6.8 sam.machine.active.ActiveTree code notes ⁂

>>> Runnable runtime Rule inference process
UNDER CONSTRUCTION <<<

6.9 sam.machine.active.ActiveTheory code notes ⁂

>>> Inference stage manager for active rules and rewrites
UNDER CONSTRUCTION <<<…

6.10 sam.machine.active.ActiveRule code notes ⁂

>>> Inference stage manager for active rules and rewrites
UNDER CONSTRUCTION <<<…

6.11 sam.machine.active.ActiveRewrite code notes ⁂

>>> The inference tree built by ActiveTheory
UNDER CONSTRUCTION <<<…

6.12 sam.gui.TreePane code notes ⁂

>>> The Viewer graphics representation for sam.machine.engine.Tree
UNDER CONSTRUCTION <<<…

7 Linked References ⋇

The following linked references are relevant to various aspects of the computational theory in the Autolog Design Notebook. These links provide some general background information, and some historical context for those aspects. The plan is to include additional references later as well as some notes regarding the relevance of the references to the various intended aspects of Autolog (after there exists an *executable* prototype of Autolog).

1. Soloman Fefferman, *Predicativity* (2002),
   https://math.stanford.edu/~feferman/papers/predicativity.pdf

2. Wikipedia, Impredicativity,
   https://en.wikipedia.org/wiki/Impredicativity

3. Wikipedia, Indexicality,
   https://en.wikipedia.org/wiki/Indexicality

4. Mark van Atten, *Predicativity and parametric polymorphism of Brouwerian implication*, (2018)
   https://arxiv.org/pdf/1710.07704.pdf