Computational Logic and Applications

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

ICCCNT'2016

Research supported by NSF grant **1423324**

Outline

1. Programming in Logic
2. Horn Clause Prolog in four slides
3. First class logic engines and their applications
4. Agent programming with logic engines
5. Combinatorial generation and type inference: λ-terms in Prolog
6. Binary tree arithmetic
7. Size-proportionate ranking/unranking for lambda terms
8. Logic Programming and circuit synthesis
9. Conclusions

Prolog: Programming in Logic

- a (very) old language: originating in the late 70's - but built on a mathematically well-understood foundation ⇒ slower at aging :-)
- Robinson: unification algorithm - for better theorem proving
- motivations: Colmeraurer: NLP, Kowalski: algorithms = logic + control
- ⇒ a computationally well-behaved subset of predicate logic
- Horn clauses: `a :- b,c,d.`
- all variables universally quantified ⇒ we do not have put quantifiers
- multiple answers returned on demand (a possibly infinite stream)
- newer derivatives: constraint programming, SAT-solvers, answer set programming: exploit fast execution of propositional logic
- like FP, and relational DB languages: a from of "declarative programming"

Prolog: raising again?

Programming Language Ratings - from the Tiobe index

- 29 Lisp 0.630 %
- 30 Lua 0.593 %
- 31 Ada 0.552 %
- 32 Scala 0.550 %
- 33 OpenEdge ABL 0.467 %
- 34 Logo 0.432 %
- 35 Prolog 0.406 %
- 36 F# 0.391 %
- 37 RPG (OS/400) 0.375 %
- 38 LabVIEW 0.340 %
- 39 Haskell 0.287 %

a year or two ago: Prolog not on the list (for being above 50)

Horn Clause Prolog in four slides

Prolog: unification, backtracking, clause selection

```
?- X=a,Y=X. % variables uppercase, constants lower
X = Y, Y = a.

?- X=a,X=b.
false.

?- f(X,b)=f(a,Y). % compound terms unify recursively
X = a, Y = b.

% clauses
a(1). a(2). a(3).    % facts for a/1
b(2). b(3). b(4).    % facts for b/1

c(0).
c(X):-a(X),b(X).     % a/1 and b/1 must agree on X

?-c(R).              % the goal at the Prolog REPL
R=0; R=2; R=3.       % the stream of answers
```

Unification: a few more examples

```
?- X=Y,Y=a.
X = a ,
Y = a

?- f(X,g(X,X))=f(h(Z,Z),U),Z=a.
X = h(a,a) ,
Z = a ,
U = g(h(a,a),h(a,a))

?- [X,Y,Z]=[f(Y,Y),g(Z,Z),h(a,b)].
X = f(g(h(a,b),h(a,b)),g(h(a,b),h(a,b))) ,
Y = g(h(a,b),h(a,b)) ,
Z = h(a,b)
```

Prolog: Definite Clause Grammars

Prolog's DCG preprocessor transforms a clause defined with "-->" like

```
a0 --> a1,a2,...,an.
```

into a clause where predicates have two extra arguments expressing a chain of state changes as in

```
a0(S0,Sn):-a1(S0,S1),a2(S1,S2),...,an(Sn-1,Sn).
```

- work like "non-directional" attribute grammars/rewriting systems
- they can used to compose relations (functions in particular)
- with compound terms (e.g. lists) as arguments they form a Turing-complete embedded language

```
f -- > g,h.
f(In,Out):-g(In,Temp),h(Temp,Out).
```

Some extra notation: `{...}` calls to Prolog, `[...]` terminal symbols

Prolog: the two-clause meta-interpreter

The meta-interpreter `metaint/1` uses a (difference)-list view of prolog clauses.

```
metaint([]).          % no more goals left, succeed
metaint([G|Gs]):-     % unify the first goal with the head of a clause
   cls([G|Bs],Gs),    % build a new list of goals from the body of the
                      % clause extended with the remaining goals as tail
   metaint(Bs).       % interpret the extended body
```

- clauses are represented as facts of the form `cls/2`
- the first argument representing the head of the clause + a list of body goals
- clauses are terminated with a variable, also the second argument of `cls/2`.

```
cls([  add(0,X,X)                          |Tail],Tail).
cls([  add(s(X),Y,s(Z)), add(X,Y,Z)        |Tail],Tail).
cls([  goal(R), add(s(s(0)),s(s(0)),R)     |Tail],Tail).
```

```
?- metaint([goal(R)]).
R = s(s(s(s(0)))) .
```

First class logic engines and their applications

First class logic engines

a richer API then what streams provided can be used

- a *logic engine* is a Prolog language processor reflected through an API that allows its computations to be controlled interactively from another *engine*
- very much the same thing as a programmer controlling Prolog's interactive toplevel loop:
  - launch a new goal
  - ask for a new answer
  - interpret it
  - react to it
- logic engines can create other logic engines as well as external objects
- logic engines can be controlled cooperatively or preemptively

Interactors (a richer logic engine API, beyond streams): new_engine/3

new_engine(AnswerPattern, Goal, Interactor):

- creates a new instance of the Prolog interpreter, uniquely identified by `Interactor`
- shares code with the currently running program
- initialized with `Goal` as a starting point
- `AnswerPattern` : answers returned by the engine will be instances of the pattern

Interactors: get/2, stop/1

get(Interactor, AnswerInstance):

- tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`
- if an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned
- is used to retrieve successive answers generated by an Interactor, on demand
- it is responsible for actually triggering computations in the engine
- one can see this as transforming Prolog's backtracking over all answers into a deterministic stream of lazily generated answers

stop(Interactor):

- stops the Interactor
- `no` is returned for new queries

The `return` operation: a key co-routining primitive

return(Term)

- will save the state of the engine and transfer *control* and a *result* `Term` to its client
- the client will receive a copy of `Term` simply by using its `get/2` operation
- an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available

Application: exceptions

```
throw(E):-return(exception(E)).
```

Exchanging Data with an Interactor

to_engine(Engine,Term):

- used to send a client's data to an Engine

from_engine(Term):

- used by the engine to receive a client's Data

Typical use of the Interactor API

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*:
   - the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
   - the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   - the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

What can we do with first-class engines?

- define the complete set of ISO-Prolog operations at source level
- implement (at source level) Erlang-style messaging - with millions of engines
- implement Linda blackboards
- implement Prolog's dynamic database at source level
- build an algebra for composing engines and their answer streams
- implement "tabling" a from of dynamic programming that avoids recomputation

Agent programming with logic engines

Cooperative coordination - concurrency without threads

- `new_coordinator(Db)` uses a database parameter `Db` to store the state of the Linda blackboard
- the state of the blackboard is described by the dynamic predicates
  - `available/1` keeps track of terms posted by `out` operations
  - `waiting/2` collects pending `in` operations waiting for matching terms
  - `running/1` helps passing control from one engine to the next

```
new_coordinator(Db):-
  db_dynamic(Db, available/1),
  db_dynamic(Db, waiting/2),
  db_dynamic(Db, running/1).
```

Agents as cooperative Linda tasks

```
new_task(Db, G):-
  new_engine(nothing, (G, fail), E),
  db_assert(Db, running(E)).
```

Three cooperative Linda operations are available to an agent. They are all expressed by returning a specific pattern to the `Coordinator`.

```
coop_in(T):-return(in(T)), from_engine(X), T=X.

coop_out(T):-return(out(T)).

coop_all(T, Ts):-return(all(T, Ts)), from_engine(Ts).
```

A Bird's view of our Lightweight Prolog Agent Layer

- *agents* are implemented as *named* Prolog dynamic databases
- each agent has a process where its *home* is located - called an *agent space*
- they share code using a simple "Twitter-style" mechanism that allows their *followers* to access their predicates
- an agent can *visit* other spaces located on local or remote machines - where other agents might decide to follow its replicated "avatars"
- the state of an agent's avatar is dynamically updated when a state change occurs in the agent's code space
- communication between agents, including avatar updates, is supported by a remote predicate call mechanism between agent spaces, designed in a way that each call is atomic and guaranteed to terminate

Agent Spaces

- an *agent space* is seen as a container for a group of agents usually associated with a Prolog process and an RLI server
- we assume that the name of the space is nothing but the name of the RLI port
- we make sure that on each host, a “broker”, keeping track of various agents and their homes, is started, when needed
- `start_space(BrokerHost,ThisHost,Port)` starts, if needed, the unique RLI service associated to a space and registers it with the broker (that it starts as well, if needed!)
- communication with agents inhabiting an agent space happens through this unique port - typically one per process
- $\Rightarrow$ all RLI calls to a given port are atomic and terminating

Visiting an Agent Space

- an agent can *visit* one or more agent spaces at a given time
- when calling the predicate `visit(Agent,Host,Port)` an agent broadcasts its database and promises to broadcast its future updates
- “avatar”: an agent is represented at a remote space by a replica of its set of clauses
- the predicate `take_my_clauses(Agent,Host,Port)` remotely asserts the agent’s clauses to the database of the agent’s “avatar”
- only the agent’s *own code* goes and not the code that the agent inherits locally

Propagation of Updates

- as the agent keeps track of all the locations where it has dispatched avatars, it will be able to propagate updates to its database using atomic, guaranteed to terminate remote calls
- an agent is also able to `unvisit` a given space - in which case the code of the avatar is completely removed and broadcasts of updates to the unvisited space are disabled

Remote Followers

- an agent can have followers in various spaces that it visits
- followers inherit the code of the avatar - and therefore all their calls stay local
- why this makes sense:
    - for instance, an agent asked to find neighboring gas stations should do it based on the GPS location of the agent space it is visiting
    - execution is local - possible non-termination or lengthy execution does no block communication ports

Combinatorial generation and type inference: λ-terms in Prolog

Lambda Terms in Prolog

- logic variables can be used in Prolog for connecting a lambda binder and its related variable occurrences
- this representation can be made canonical by ensuring that each lambda binder is marked with a distinct logic variable
- the term $\lambda a.((\lambda b.(a(b\,b)))(\lambda c.(a(c\,c))))$ is represented as
- `l(A,a(l(B, a(A,a(B,B))), l(C, a(A,a(C,C)))))`
- "canonical" names - each lambda binder is mapped to a distinct logic variable
- scoping of logic variables is "global" to a clause - they are all universally quantified

De Bruijn Indices

- *de Bruijn Indices* provide a name-free representation of lambda terms
- terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation
  - variables following lambda abstractions are omitted
  - their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* on the way up to the root of the term
- term with canonical names: l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C))))) $\Rightarrow$
- de Bruijn term: l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1),a(v(0),v(0))))))
- note: we start counting up from 0
- closed terms: every variable occurrence belongs to a binder
- open terms: otherwise

Generating Motzkin trees: the skeletons of lambda terms

- Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2
- ⇒ like lambda term trees, for which we ignore the de Bruijn indices that label their leaves

```
motzkinTree(L,T):-motzkinTree(T,L,0).

motzkinTree(u)-->down.
motzkinTree(l(A))-->down,
  motzkinTree(A).
motzkinTree(a(A,B))-->down,
  motzkinTree(A),
  motzkinTree(B).

down(S1,S2):-S1>0,S2 is S1-1.
```

Generating closed de Bruijn terms

- we can derive a generator for closed lambda terms in de Bruijn form by extending the Motzkin-tree generator to keep track of the lambda binders
- when reaching a leaf `v/1`, one of the available binders (expressed as a de Bruijn index) will be assigned to it nondeterministically

```
genDBterm(v(X),V)--> {down(V,V0),between(0,V0,X)}.
genDBterm(l(A),V)--> down, {up(V,NewV)},
  genDBterm(A,NewV).
genDBterm(a(A,B),V)--> down,
  genDBterm(A,V),
  genDBterm(B,V).
```

Generating closed de Bruijn terms – continued

```
genDB(L,T):-genDB(T,0,L,0).    % terms of size L
genDBs(L,T):-genDB(T,0,L,_).   % terms of size up to L
```

Generation of terms with up to 2 internal nodes.

```
?- genDBterms(2,T).
T = l(v(0)) ;
T = l(l(v(0))) ;
T = l(l(v(1))) ;
T = l(a(v(0), v(0))).
```

Generating simply typed de Bruijn terms of a given size

- we can interleave generation and type inference in one program
- DCG grammars control size of the terms with predicate `down/2`

```
genTypedTerm(v(I),V,Vs)--> {
    nth0(I,Vs,V0),      % pick binder and ensure types match
    unify_with_occurs_check(V,V0)
  }.
genTypedTerm(a(A,B),Y,Vs)-->down, % application node
  genTypedTerm(A,(X->Y),Vs),
  genTypedTerm(B,X,Vs).
genTypedTerm(l(A),(X->Y),Vs)-->down, % lambda node
  genTypedTerm(A,Y,[X|Vs]).
```

- 3 orders of magnitude faster than existing algorithms

Binary tree arithmetic

Blocks of digits in the binary representation of natural numbers

The (big-endian) binary representation of a natural number can be written as a concatenation of binary digits of the form

$$n = b_0^{k_0} b_1^{k_1} \ldots b_i^{k_i} \ldots b_m^{k_m} \quad (1)$$

with $b_i \in \{0,1\}$, $b_i \neq b_{i+1}$ and the highest digit $b_m = 1$.

Proposition

An even number of the form $0^i j$ corresponds to the operation $2^i j$ and an odd number of the form $1^i j$ corresponds to the operation $2^i (j+1) - 1$.

Proposition

A number n is even if and only if it contains an even number of blocks of the form $b_i^{k_i}$ in equation (1). A number n is odd if and only if it contains an odd number of blocks of the form $b_i^{k_i}$ in equation (1).

The constructor c: prepending a new block of digits

$$c(i,j) = \begin{cases} 2^{i+1}j & \text{if } j \text{ is odd,} \\ 2^{i+1}(j+1)-1 & \text{if } j \text{ is even.} \end{cases} \quad (2)$$

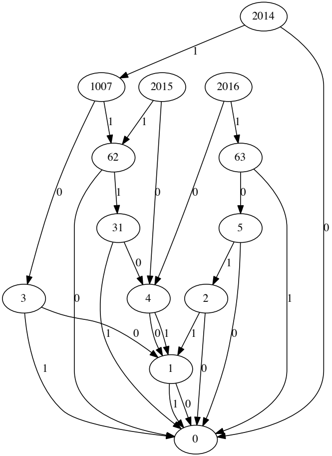
- the exponents are $i+1$ instead of i as we start counting at 0
- $c(i,j)$ will be even when j is odd and odd when j is even

Proposition

The equation (2) defines a bijection $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^{+} = \mathbb{N} - \{0\}$.

The DAG representation of 2014,2015 and 2016

- a more compact representation is obtained by folding together shared nodes in one or more trees
- integers labeling the edges are used to indicate their order



Binary tree arithmetic

- parity (inferred from from assumption that largest bloc is made of 1s)
- as blocks alternate, parity is the same as that of the number of blocks
- several arithmetic operations, with Haskell type classes at `http://arxiv.org/pdf/1406.1796.pdf`
- complete code at: `http://www.cse.unt.edu/~tarau/research/2014/Cats.hs`

Proposition

Assuming parity information is kept explicitly, the operations `s` *and* `p` *work on a binary tree of size N in time constant on average and and $O(log^*(N))$ in the worst case*

Successor (s) and predecessor (p)

```
s(x,x>x).
s(X>x,X>(x>x)):-!.
s(X>Xs,Z):-parity(X>Xs,P),s1(P,X,Xs,Z).

s1(0,x,X>Xs,SX>Xs):-s(X,SX).
s1(0,X>Ys,Xs,x>(PX>Xs)):-p(X>Ys,PX).
s1(1,X,x>(Y>Xs),X>(SY>Xs)):-s(Y,SY).
s1(1,X,Y>Xs,X>(x>(PY>Xs))):-p(Y,PY).

p(x>x,x).
p(X>(x>x),X>x):-!.
p(X>Xs,Z):-parity(X>Xs,P),p1(P,X,Xs,Z).

p1(0,X,x>(Y>Xs),X>(SY>Xs)):-s(Y,SY).
p1(0,X,(Y>Ys)>Xs,X>(x>(PY>Xs))):-p(Y>Ys,PY).
p1(1,x,X>Xs,SX>Xs):-s(X,SX).
p1(1,X>Ys,Xs, x>(PX>Xs)):-p(X>Ys,PX).
```

Size-proportionate ranking/unranking for lambda terms

A size-proportionate bijection from λ-terms to tree-based natural numbers

- injective encodings are easy: encode each symbol as a small integer and use a separator
- in the presence of a bijection between two infinite sets of data objects, it is possible that representation sizes on one side are exponentially larger than on the other side
- e.g., Ackerman's bijection from hereditarily finite sets to natural numbers $f(\{\}) = 0, f(x) = \sum_{a \in x} 2^{f(a)}$
- however, *if natural numbers are represented as binary trees*, size-proportionate bijections from them to "tree-like" data types (including λ-terms) is (un)surprisingly easy!
- some terminology: "bijective Gödel numbering" (for logicians), same as "ranking/unranking" (for combinatorialists)

Ranking and unranking de Bruijn terms to binary-tree represented natural numbers

- variables `v/1`: as trees with `x` as their left branch
- lambdas `l/1`: as trees with `x` as their right branch
- to avoid ambiguity, the rank for application nodes will be incremented by one, using the successor predicate `s/2`

```
rank(v(0),x).
rank(l(A),x>T):-rank(A,T).
rank(v(K),T>x):-K>0,t(K,T).
rank(a(A,B),X1>Y1):-rank(A,X),s(X,X1),rank(B,Y),s(Y,Y1).
```

- unrank simply reverses the operations – note the use of predecessor `p/2`

```
unrank(x,v(0)).
unrank(x>T,l(A)):-!,unrank(T,A).
unrank(T>x,v(N)):-!,n(T,N).
unrank(X>Y,a(A,B)):-p(X,X1),unrank(X1,A),p(Y,Y1),unrank(Y1,B).
```

What can we do with this bijection?

- a size proportional bijection between de Bruijn terms and binary trees with empty leaves
- random generation of binary tree-algorithms are directly applicable to lambda terms
- a different but possibly interesting distribution
- “plain” natural number codes

```
?- t(666,T),unrank(T,LT),rank(LT,T1),n(T1,N).
T = T1, T1 =  (x> (x> (x> ((x>x)> ((x>x)> (x> (x> (x>x)))))))),
LT = l(l(l(a(v(0), a(v(0), v(1)))))),
N = 666 .
```

Logic Programming and circuit synthesis

Exact Circuit Synthesis

Given a library of universal gates, the exact synthesis of boolean circuits consists of finding a minimal representation using only gates of the library.

- a recurring topic of interest in circuit design, complexity theory, boolean logic, combinatorics and graph theory
- extreme intractability (typically, single digit number of gates for most problems)
- exact synthesis is usable in combination with heuristic methods

Exact synthesis - things to put together

Our exact synthesis algorithm uses depth-first backtracking to find minimal N-input, M-output circuits representing boolean functions, based on a given library of operators and constants.

Needed for an efficient implementation:

- Combinatorial Generation
- Minimization by Design: smallest circuits first
- Constraint Propagation
- Efficient bitstring algebra for evaluation
- Sharing of gates between multiple outputs

The synthesis algorithm

- First, obtain an output specification from a symbolic formula and compute a conservative upper limit (in terms of a cost function, for instance the number of gates) on the size of the synthesized expression.
- Next, enumerate candidate circuits (represented as directed acyclic graphs) *in increasing cost order*, to ensure that minimal circuits are generated first.
- Until a maximum number of gates is reached, connect a new gate's inputs to the previously constructed gates'
- On success, the resulting circuit is decoded into a symbolic expression consisting of a list of primary input variables, a list of gates describing the operators and their input and output arguments, and a list of primary output variables.

outputs.

Expressiveness

Expressiveness = Performance on Exact Synthesis Tasks

Fig. 2 compares a few libraries used in synthesis with respect to the total gates needed to express all the 16 2-argument boolean operations.

Library	Total	Library	Total	Library	Total
`*,=,0`	23	`+,^,1`	23	`<,=>`	24
`*,^,1`	25	`+,=,0`	25	`*,=,^`	26
`+,=,^`	26	`<,=`	28	`=>,^`	28
`<,1`	28	`=>,0`	28	`<,nhead`	30
`=>,nhead`	30	`nand`	36	`nor`	36

Figure: Total gates for minimal libraries

Expressiveness Indicators

- This comparison provides our first indicator for the relative expressiveness of libraries.
- Surprisingly, $(\Rightarrow, 0)$ and its dual $(<, 1)$ do clearly better than `nand` and `nor`: they can express all 16 operators with only 28 gates.
- The overall “winner" of the comparison, expressing the 16 operators with only 20 gates is the library $<, \Rightarrow, 0, 1$.
- both of its operators have small transistor count implementations

Applications

- full automation of exact synthesis tasks
- discovery of minimal universal libraries
- quantitative expressiveness comparison for library components
  - the first based on how many gates are used to synthesize all binary operators
  - the second based on how many *N*-variable truth table values are covered by combining up to *M* gates from the library
- extension to reversible logic needed for quantum computing

Conclusions

- Logic (and constraint) programming languages are an ideal tool for combinatorial search algorithms (e.g. circuit synthesis)
- this is especially the case when *unification* is involved (e.g. type inference)
- $\Rightarrow$ test generation for λ-calculus based language compilers and proof assistants
- ranking/unranking to natural numbers represented as binary trees is naturally size-proportionate - it can be extended with other data structures
- exact circuit synthesis reveals new things about the relative expressiveness of boolean function libraries
- by decoupling logic engines and threads, programming language constructs for coordination can be kept simple and scalable
- first class engines bring additional flexibility needed for practical applications (e.g. agent programming)