The Design of j-DREW: a Deductive Reasoning Engine for the Semantic Web

Bruce Spencer
National Research Council Canada
and
University of New Brunswick
Fredericton, New Brunswick

Original Idea: train people to build the Semantic Web

- Courses on systems employing rule engines an Internet applications
  - Writing deduction engines in Java/C/C++
  - Interfacing with Internet API
  - Old techniques (Prolog 30 years ago)
  - New techniques from CADE System Competition
- Meier and Warren's book: Programming in Logic, 1988
  - Updated in Java?
  - Specific to Prolog at low level

Will such a course work?

- No
  - Guts of Prolog, Internet API’s, how to program in logic
  - At least three courses here
- Yes
  - Students understand recursion
  - How to build a tree
    how to search a space
    - Propositional theorem prover
  - how to interface to Internet

This talk

- Architecture for building deduction systems
  - first order
  - easily configured
    - forward or backward
  - embedded
  - supports calls to and from rest of system
- Tour of internals
  - backward & forward engines
  - tree/proof
  - terms
  - bindings
  - discrimination tree
- Prototypes

Choose the right abstractions

- Goal, Unifier, ProofTree
- use Java iterators: pay as you go
  - for finding the next proof
- Make every Goal responsible for its list o matching clauses
  - hasNextMatchingClause()
  - attachNextMatchingClause()
- Place Goals in stack of backtrack points
  - popped in reverse chronological order

Propositional Prover: 1

```
initially proofTree has an open Goal
loop
   if(proofTree.hasNoOpenGoal())
       halt('success');
   else
      Goal g = proofTree.selectOpenGoal();
      g.createMatchingClauseList();
      if(g.hasMoreMatchingClauses())
          g.attachNextMatchingClause();
          choicePoints.push(g);
      else
          chronologicalBacktrack();
```

Propositional Prover: 2

```
chronologicalBacktrack
    while(not choicePoints.empty())
        Goal g = choicePoints.pop();
        g.removeAttachedClause();
        if(g.hasMoreMatchingClauses())
            return
    halt('failure');
```

Moving to First Order Logic

- Students struggle with variables
  - Unification
  - Composition of substitutions
  - Unbinding on backtrackin

- Can we hide the hard stuff?
  - Powerful abstractio

Hiding the hard stuff

- When attaching a clause to a Goal
  - Matching clause must be a instance of input clause
    - Semi-unification creates the instance
  - Bindings to variables in goal may be propagated through tree now or later

- When removing the clause
  - relax any propagated variable bindings

proof tree goal	p(a, Y)
input clause	p(X, b) :- …
clause instance	p(a, b) :- …
propagated binding	Y←b

Shallow or deep variables?

- Shallow
  - variable binding is a list of replacements
  - traverse list for each lookup
  - undoing: remove most recent replacements
    {X ← f(Y)} • {Y ← a}

- Deep
  - an array of (all) variables and their curren values
    [X ← f(a)
     Y ← a
     …]
  - undoing: pop stack of previous values (trail)

Choosing between shallow and deep

- Shallow
  - pay for each lookup
  - unbinding is cheap
- Deep
  - lookup is cheap
  - may need many large arrays of possible variables
- *j*-DREW uses *local* deep
  - each clause has own array of just local variables, named –1, -2, …
  - scope is clause-wide
  - so propagation necessary

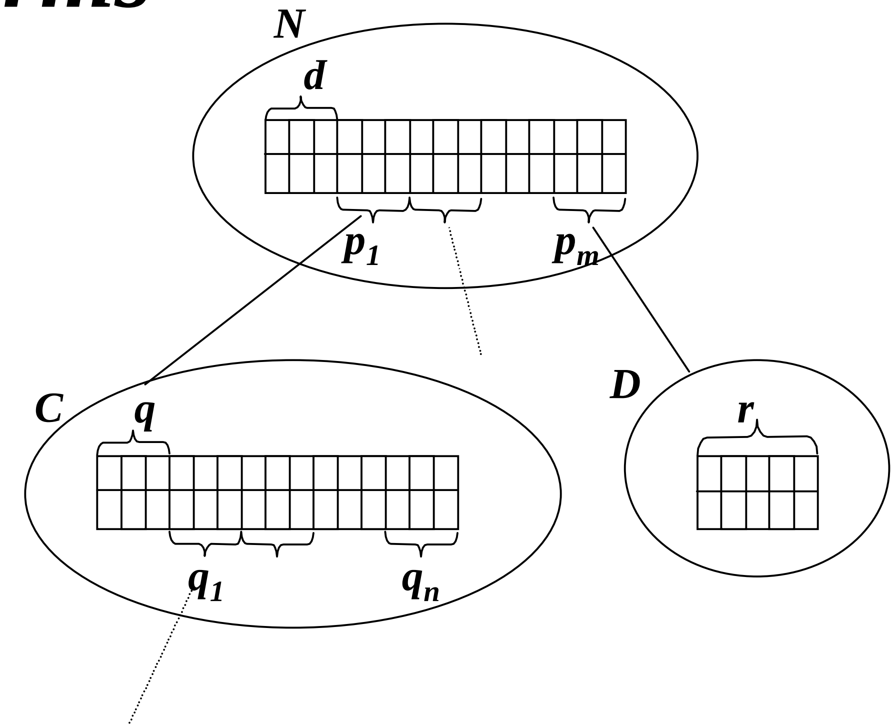
Goal Tree and flatterms

- **Each node has head and body atoms**
- **Body atoms form goals**
  - **attach children**
- **resolved p_1 from**

  $$d \leftarrow p_1, \ldots, p_m$$

  against q from

  $$q \leftarrow q_1, \ldots, q_n$$
- **resolved p_m against $r \leftarrow$.**



Flatterms to represent atoms

- ***j*-DREW uses flatterms**
  - **Array of pairs:**
    - **symbol table ref**
    - **length of subterm**
  - **Not structure sharing**
- **Flatterms save theorem provers time and space (de Nivelle, 1999)**
- **Data transfer between deduction engine and rest of application**

Symbol Table

	name	arity
1	p	2
2	f	1
3	g_1	0
4	g_2	0
5	h	0
6	h	1

Flatterm for $p(f(g_1), h, h(g_2), g_1)$

	symbol	length
1	1	7
2	2	2
3	3	1
4	5	1
5	6	2
6	4	1
7	3	1

Variables are clause-specific

- **Variables use negative indexes**
- **Bindings are references to flatterm & position**
- **Unifier**
  $X \leftarrow g_2$
  $Y \leftarrow f(g_2)$
  $W \leftarrow h(g_2)$
  $Z \leftarrow f(g_2)$

Flatterm for left
$p(f(h(X)), h(Y), f(X), Y)$

	symbol	length
1	1	9
2	2	3
3	6	2
4	-1	1
5	6	2
6	-2	1
7	2	2
8	-1	1
9	-2	1

	position	side
-1	6	right
-2	5	right

Flatterm for right
$p(f(W), h(f(g_2)), Z, Z)$

	symbol	length
1	1	8
2	2	2
3	-1	1
4	6	3
5	2	2
6	4	1
7	-2	1
8	-2	1

	position	side
-1	3	left
-2	5	right

Composing and undoing Bindings

- **Local deep bindings currently do not allow composition**
  - **bindings must be done to a flatterm**
  - **new binding on a ne   flatterm**

- **Backtracking is integrated with unbinding**
  - **for quick unbinding, we use a stack of flatterms for each goal.**

Evaluation of local deep bindings

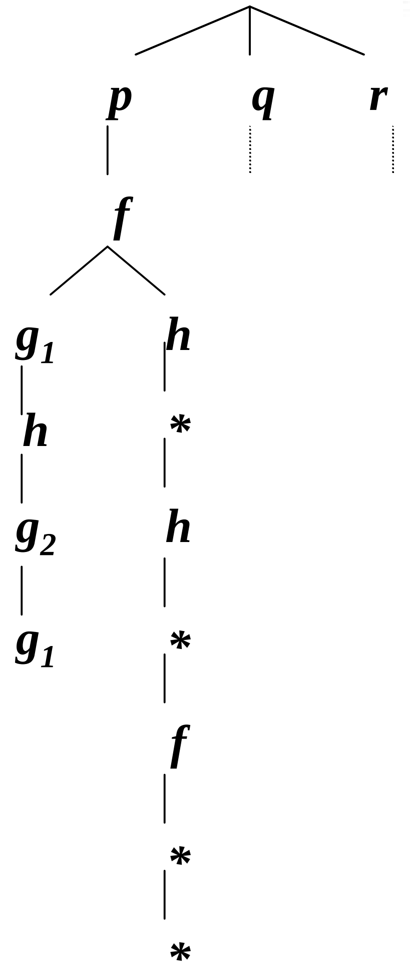
- Disadvantage for backtrackin
  - must propagate bindings to other nodes
- Advantage
  - fast interaction with rest of system
  - simple, no environments to pass aroun
  - compact, no large arrays
- Appropriate
  - for forward chaining
    - no backtracking, no propagation
  - Probably appropriate when backward chaining function-free logic
- Design decision to revisit

Discrimination trees

- Given a goal we want to access matching clauses quickl
- Every-argument addressin
  - unlike Prolog's first argument addressing
- Useful for RDF triples
  - a pattern may have variable in first argument
  - rdf(X, ownedb , 'Ora Lassila')

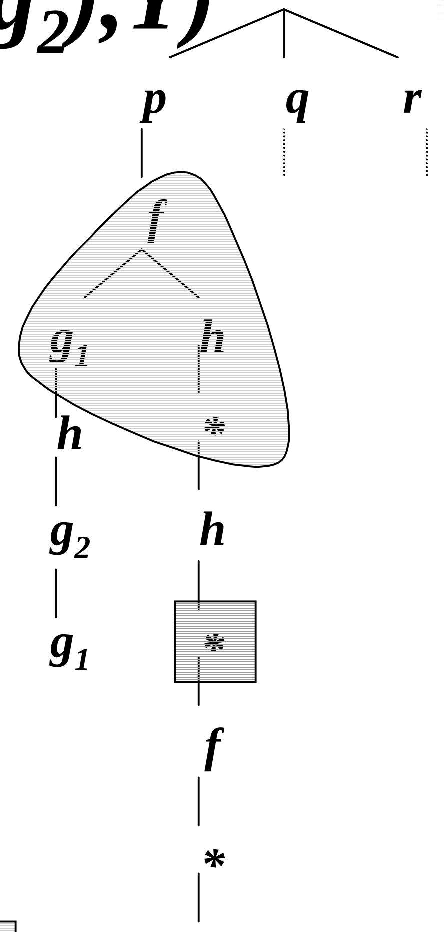
Discrimination trees

- Given a goal, want to access input clauses with matching heads quickly
- Index into clauses via a structure built from heads
- Replace vars by *
  - imperfect discrimination
- merge prefixes as much as possible
  - a tree arises
- We adde $p(f(g_1),h(g_2),g_1)$ $p(f(h(X)),h(Y),f(Z, Z))$



Finding heads for goal $p(X,h(g_2),Y)$

- replace vars in goal by *
  - $p(*,h(g_2),*)$
- Find instances of goal
  - * in goal, skip subtree
- Find generalizations of goal
  - * in tree, skip term in goal
- Find unifiable
  - combination of both



$p(f(g_1),h(g_2),g_1)$

$p(f(h(X)),h(Y),f(Z, Z))$

Iterator for matching clauses

- We use Java idioms where possible
- Java's iterators give access to sequence
  - next()
  - hasNext()
- Used for access to sequence of matchi clauses
  - used in discrimination tree for access to roots leaves of skipped tree (McCune's term: jump-list)

Working Prototypes:

- Basic Prolog Engine
  - Accepts RuleML, or Prolog, or mixture
  - Iterator for instances of the top goal
  - Main loop is same code as propositional theorem prover (shown earlier)
  - Builds, displays deduction tree
    - available to rest of system
  - Negation as failure

More working prototypes: Variants of Top-Down Engine

- User directe
  - User selects goals
  - User chooses clauses
    - keeps track of clauses still left to try
  - Good teaching tool
- Bounded search
  - iteratively increase bound
  - every resolution in search space will eventually be tried
  - a fair selection strateg
- Original variable names supplied
  - particularly important for RuleML

When to propagate bindings?

- When all subgoals closed (1)
  - best option if selecting deepest goal
- When new clause is attached
  - to all delayed goals (2)
    - best option if sound negation or delaying goals
  - to all open goals (3)
    - best option if user selects
- Propagation on demand (4)
  - lazy propagation
- Currently (1) and (3) working

proof tree goal	p(a, Y)
input clause	p(X, b) :- …
clause instance	p(a, b) :- …
propagated binding	$Y \leftarrow b$

Not-yet-working: Calls to user's Java code

- Want this to incur little overhead
- Java programmer uses flatterms
- Interface to symbol table
  - symbol lookup
  - add new symbols
- Argument list: an array of symbols
- Works with backtracking
  - User's Java procedure is an iterator
- Works with forward reasonin

Dynamic additions

- Some asynchronous process loads new rules
  - push technolog
- Backward chaining
  - additions are unnatural
  - Using iterative bounds
    - look for additions between bounds
- Forward chaining (next)

Bottom-Up / Forward Chaining

- **Set of support prover for definite clauses**
- **Facts are supports**
- **Theorem: Completeness preserved when definite clause resolutions are only between first negative literal and fact.**
  - **Proof: completeness of lock resolution (Boyer's PhD)**
- **Use standard search procedure to reduce redundant checking (next)**
- **Unlike OPS/Rete, returns proofs and uses first order syntax for atoms**

Theorem Prover's Search Procedure

- 3 Definite Clause Lists:
  - new facts (priority queue)
  - old facts
  - mixed
- 2 Discrimination trees:
  - used facts
  - rules, indexed on first goal

```
loop
    select new fact
    for each matching rule
        resolve
    process new result
```

```
process new result(C)
   if C is rule
      for each old fact matching first
         resolve
         process new result
      add C to rules
   else
      add C to new facts
```

Event – Condition - Action

- Suppose theorem prover saturates
  - may need datalog, subsumption
  - new facts added from
    - push process
    - Java event listener
  - adding a fact restarts saturation
    - could generate new Java events
- ECA interaction with Java events

j-DREW sound and complete

- Sound unification
- Search complete variant
  - fair search procedure rather than depth-first
  - uses increasing bounds
- Sound negation
  - delay negation-as-failure subgoals
  - until ground or until only NAF goals remain

Related Work

- Prolog
  - Not compiled
  - More flexible
    - Dynamic additions
    - Web-ized
    - Programmer's API
  - Performance requirements different
    - *j*-DREW unlikely to yield megalips

Related Work

- Mandarax
  - easy to use RuleML editor and engine
- CommonRules
  - compiles priorities
  - Datalog
  - also top-down, bottom up
    - shares view of single semantics for bot

Summary

- Architecture for Java-based reasoning engines
  - forward & backward
- Backward: variable binding/unbinding automatic
  - tied with choicepoints
  - configurable
- Integrated with other Java APIs
- Small footprint
  - Depolyed as thread, on server, on client, mobile
- Dynamic additions to rules
  - Integration of RuleML and Prolog rules in same proofs
- Proofs available

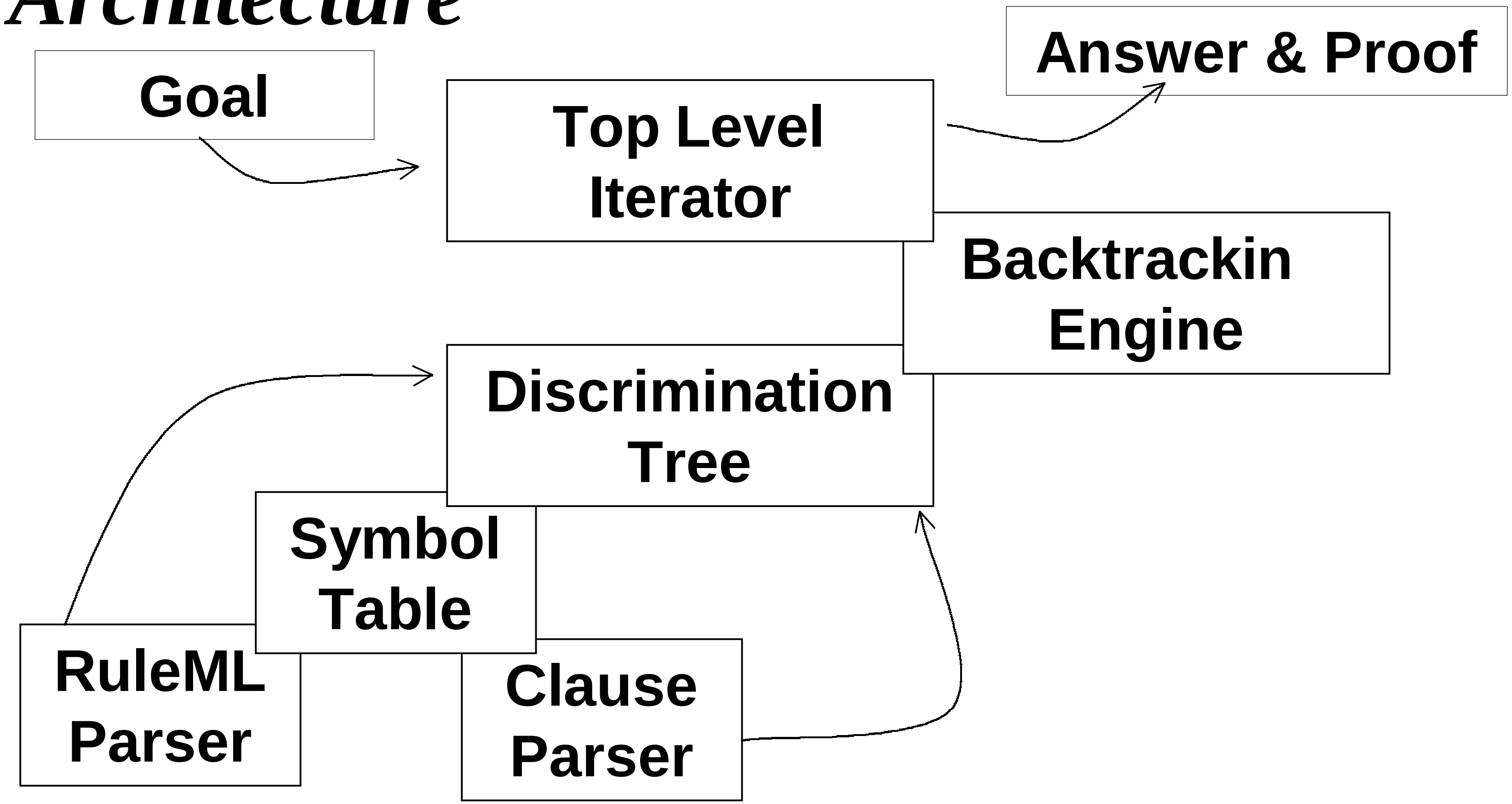
Canada's new e-Business national lab

- New Brunswick
  - over 90 people planned, about 20 so far
  - $38 million over 5 years
  - 3 locations
    - Fredericton 27 staff researchers, 13 support, 40 visitors, new building on UNB campus
    - Moncton and Saint John 14 more
- http://www.iit.nrc.ca
  - then follow"E-Business link"
  - semantic web, e-procurement, interactive voice, telehealth, e-learning, CRM, security
- recruiting no

j-DREW Demo

Bruce Spencer
National Research Council Canada
and
University of New Brunswick
Fredericton

Architecture



Demo

- **1 combining RuleML and Prolog**
  2 User interaction