Pattern-Directed Inference Systems

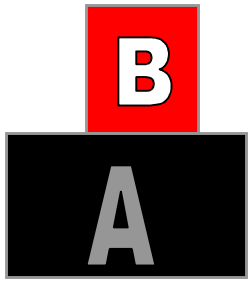
EECS 344

Winter, 2008

Pattern-Directed Inference Systems

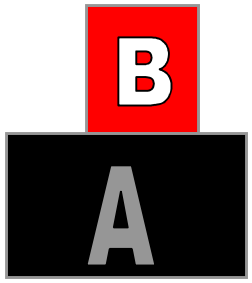
- One of the most popular categories of reasoning systems
  - Procedural deduction systems
  - OPS-like production rule systems
  - Mycin-like backward-chaining systems
  - Prolog-like system
- Focus on *antecedent reasoning model*
- Program: Tiny Rule Engine (TRE)

Example



Assertions	Rules
(on blockA table) (on blockB blockA) (red blockB) (black blockA)	(rule (red ?b1) (assert! (small ?b1))) (rule (on ?b1 ?b2) (rule (on ?b2 table) (assert! (tower ?b1 ?b2)))) . . .

Example



Assertion matches trigger pattern of rule

Assertions

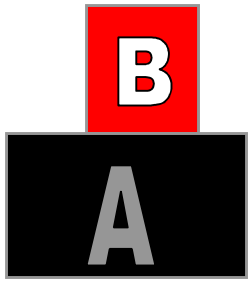
```
(on blockA table)
(on blockB blockA)
(red blockB)
(white blockA)
.
.
.
.
.
.
```

Rules

```
(rule (red ?b1)
(assert! (small ?b1)))

(rule (on ?b1 ?b2)
  (rule (on ?b2 table)
    (assert!
     (tower ?b1 ?b2))))
.
.
.
```

Example



Execution of rule causes new assertion to be added

Assertions

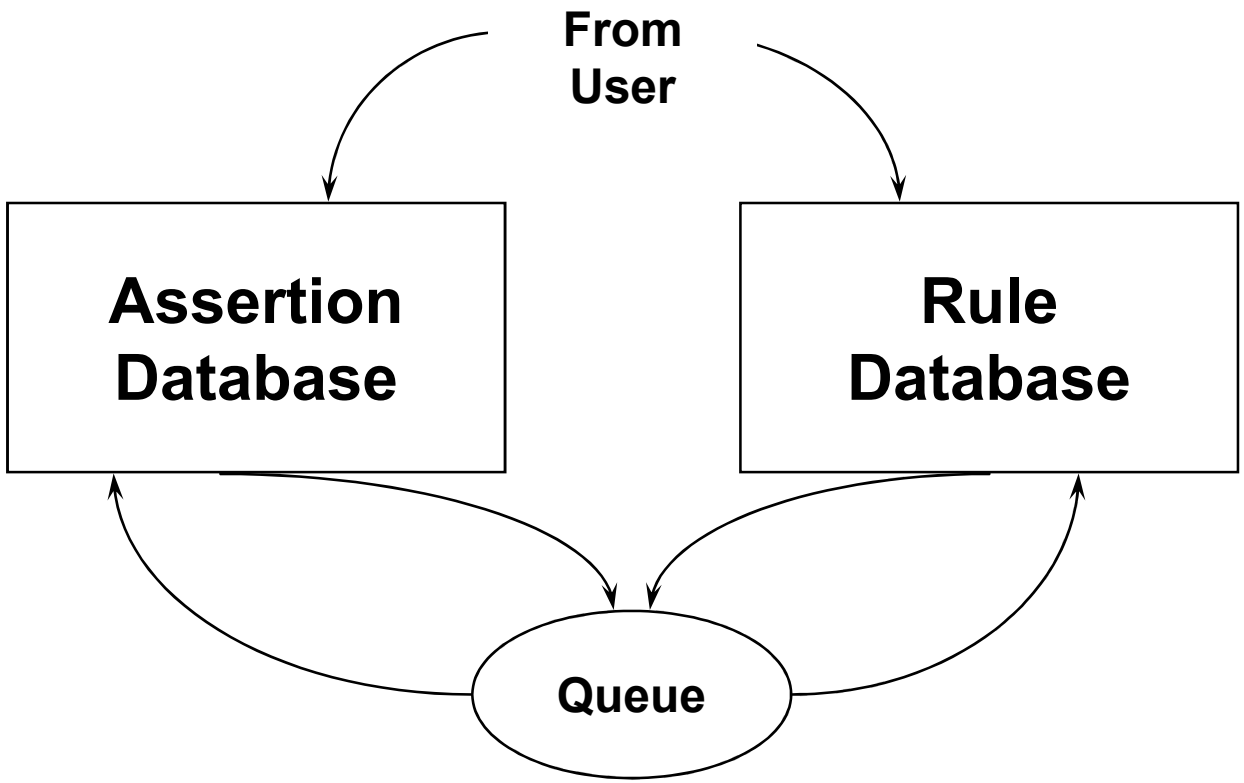
```
(on blockA table)
(on blockB blockA)
(red blockB)
(white blockA)
(small blockB)
.
.
.
.
.
.
```

Rules

```
(rule (red ?b1)
  (assert! (small ?b1)))

(rule (on ?b1 ?b2)
  (rule (on ?b2 table)
    (assert!
     (tower ?b1 ?b2))))
.
.
.
```

Simplest PDIS architecture:



Assertions as Representation Medium

- Assertions can be used to represent knowledge in a variety of ways
  - `(this statement is false)`
  - `(implies (kiss john mary)`
    `(slap mary john))`
  - `(thigh-bone connected-to`
    `knee-bone)`

Encoding frames as assertions

```
(atrans (actor (person (name (john))))
        (object (book))
        (to (person (name (mary))))
        (from (person (name (john)))))
```

```
(act23 instance-of atrans)
(act23 actor person3)
(act23 object book8)
(act23 to person4)
(act23 from person3)
```

TRE Simplifications

- No retraction
  - What to do about multiple derivations?
  - What to do about retracting derived values?
  - We'll see a solution to this later
- Pure antecedent rule model
  - Very simple
  - Generally used as component in larger systems
- Won't be overly concerned about efficiency
  - But we won't completely ignore it either
  - Goal here is simplicity

Pattern Matching: Unification

- `Unify` takes two patterns and a list of bindings
  - Both patterns can have variables
  - Bindings can be empty
- Produces a new set of bindings which suffice to make the two patterns identical.
- Example:
  `(foo ?x ?y)` with `(foo a b)` yields
  `((?x . a)(?y . b))`
- Example: `?x` doesn't unify with `(F ?x)`

Examples of rules

```
(rule (student ?x)
      (assert! `(broke ,?x))

(rule (implies ?p ?q)
      (rule ?p
            (assert! ?q)))
```

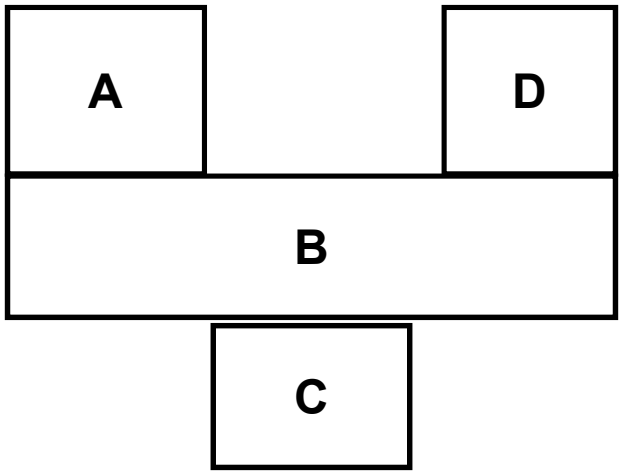
Recall: Scope and extent from programming language semantics

- Scope -- area of code over which a variable is defined.
  - *Lexical scope:* bound only within the lambda expression (e.g., let or defun)
  - *Indefinite scope:* bound everywhere
- Extent -- time period over which it is defined
  - *Dynamic extent:* bound during execution of the procedure
  - *Indefinite extent:* once bound, remains bound forever
- Analogous concepts are useful in thinking about logical environments in problem-solvers

Environments in Rules

- **Rule bodies are lexically scoped**
- **The bindings of pattern variables previous made form the environment of the body**
- **Example:**

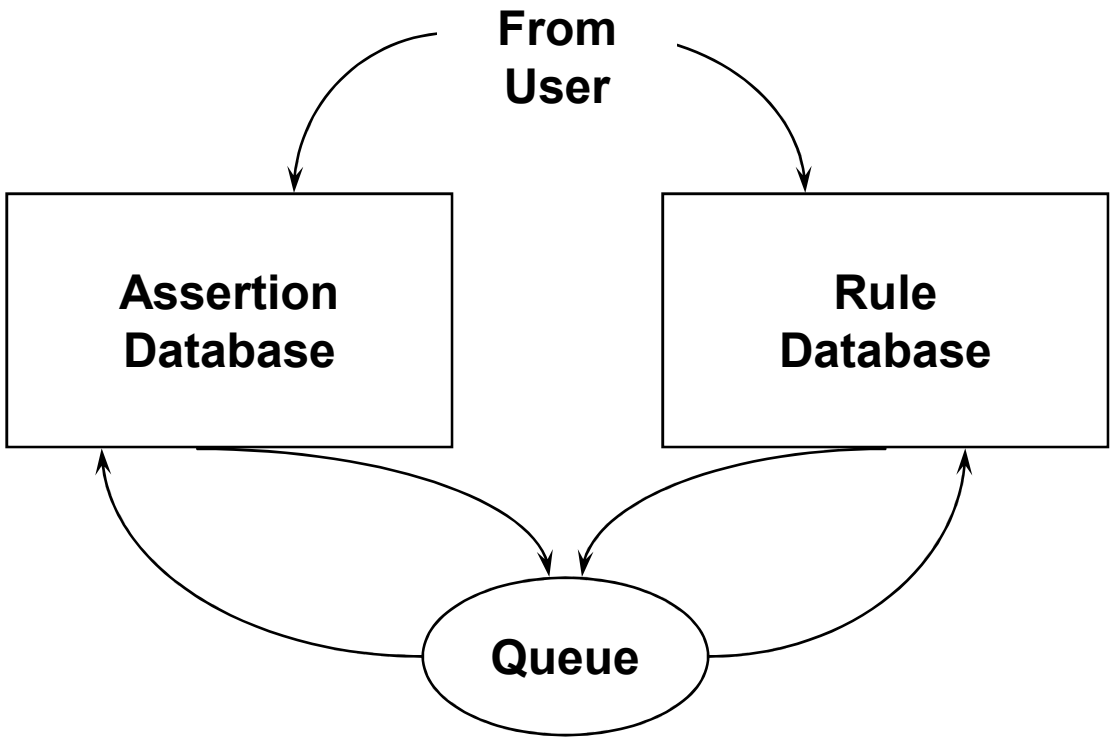
```
(rule (on ?a ?b)
      (rule (on ?b ?c)
            (assert! `(tower ,?a ,?b ,?c))))
```



Properties of rules

- Indefinite extent: Once spawned, a rule lives forever.
  - Example:
    `(assert! '(on a b), (assert! '(on b c))`
    yields same result as
    `(assert! '(on a b)), <N random assertions>, (assert! '(on b c))`
- *Order-independence:* Results are the same no matter when you add the same set of assertions and rules
  - Example:
    `(assert! '(on a b)), (assert! '(on b c))`
    yields same result as
    `(assert! '(on b c)), (assert! '(on a b))`

Architecture of TRE



Design Constraints on TRE

- Order-independence ⇒ restrictions
  - All rules are executed eventually.
  - No deletion
  - Evil: Using `fetch` in the body of a rule.
- Order-independence ⇒ freedom
  - We can execute rules in any order.
  - We can interleave adding assertions and triggering rules as convenient

Design choices for TRE

- Assertions implemented as list structure
  - Present in database $\Rightarrow$ belief
- Pattern matching implemented as unification
  - Simplification: No variables in database of facts
- Database: Must efficiently bring rules and assertions together
- Rules: Must pass around execution environments
- Queue: What sort of things are put there?

How do we choose which rules to run against which assertions?

- Given a rule trigger, which assertions in the database might match it?
- Given an assertion in the database, which rules might be applicable
- Basic problem: *Pattern-based retrieval*

Pattern-based retrieval

- Task: Given a pattern, return matching patterns from large set of assertions
- Straw man approach: run `unify` between pattern and each datum
  - Too expensive
- Two-stage approach: cheap filter for plausible candidates, followed by unify to produce bindings
  - What should the filter be?

Possible filter: Discrimination trees

- Pluses:
  - General over all patterns
  - Time efficient, scales reasonably well
- Minuses:
  - Complex bookkeeping
  - More space used by indexes than simple lists

Possible filter: CAR indexing

- Also called “class indexing”
  - Store rule by first symbol in trigger
  - Store assertion by first symbol in list
- Pluses:
  - Simple and efficient (time and space)
- Minuses:
  - Can’t handle patterns with variable as first symbol
  - May store many items under single index
- One fix:
  - CADR indices for entries that contain many items
    - » Classic trick in Prolog implementations

Possible filter: Generalized hashing

- Multiple hash tables, for indices such as:
  - CAR of expression
  - CADR of expression
  - Number of items in expression
- Retrieve all entries, take intersection
- Pluses:
  - Works on all patterns
- Minuses
  - Really inefficient (time and space)
  - Seldom used

What TRE uses: CAR indexing

- Simplest to implement
- For small to medium-scale systems, almost as efficient as discrimination trees

Some programming conventions

- `tinter.lisp` describes system "glue"
- Things to notice:
  - Use of globals as registers (`*TRE*`)
  - `With-TRE`, `in-TRE`
  - Macro to encapsulate debugging: `debugging-tre`

TRE Database highlights

- Contained in the file `data.lisp`
- One database for rules and assertions
- Organized around classes of assertions *(dbclasses)*
- Key procedures to understand:
  - `get-dbclass`
  - `fetch`

TRE Rule highlights

- Contained in the file `rules.lisp`
- Environment defined as alist of (*variable . value*)
- Global register `*env*` used to pass bindings around.
- Queue
  - Since order-independent, exhaustive, details of queue aren't important.
  - Use LIFO for simplicity
- Key procedures to understand:
  - `run-rule`

Tradeoffs in TRE's rule mechanism

- Advantages
  - Simple to implement
    - » Pattern match single trigger
    - » Treat rule body as code to execute
    - » Rewrite bindings as let statement
  - Flexible
- Disadvantages
  - Slow (due to `eval` and `unify`)
  - Overly flexible
    - » Rule body could be anything, e.g., web crawler, Quicken, …

Unification

- Unification is a particular form of pattern-matching
  - Unification can handle variables in both pattern and datum
  - Produces: Set of bindings that makes two patterns identical
- Basic algorithm:
  - tree-walk the pattern and datum
  - When variable encountered, check if bound
    - » If so, bindings must be *consistent*
      - Unification FAILS if they aren't
    - » If not, bind variable to corresponding piece of expression

TRE's Unifier

- Contained in `unify.lisp`
- Unlike most Prologs, performs occurs-check (see `free-in?`).
- Not a full unifier.