Extending Pattern Directed Inference Systems

EECS 344
Winter 2008
Extending the basic PDIS model

• How to increase the convenience of the system
• How to increase the logical power of the system
• How to increase the efficiency of the system
FTRE Changes

• Syntax changes
  – Triggers are now a list, interpreted conjunctively
  – `rassert!` to remove backquotes

• Examples

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

becomes

```
(rule ((show ?q) (implies ?p ?q))
  (rassert! (show ?p)))
```
FTRE Syntax: Trigger keywords

- :\texttt{var} = next item is a variable to be bound to entire pattern
- :\texttt{test} = next item is lisp code executed in environment so far. If \texttt{nil}, match fails.

Example:

```
(rule ((show ?q)
   :test (not (fetch ?q))
   (implies ?p ?q) :var ?imp)
(debug-nd
 "~% BC-CE: Looking for ~A to use ~A.."
 ?p ?imp)
(rassert! (show ?p)))
```
Implementation notes on syntax changes

- For `rassert!`, look at the procedure `quotize` in `funify.lsp`
- For `:var`, `:test`, also look at `funify.lsp`
Problem: We need more inferential power

- Implementing full KM* requires the ability to make and retract assumptions
  - Indirect proof, negation introduction, conditional introduction, etc.

- Implementing search procedures more generally requires ability to make and retract assumptions
  - Example: N-queens problem
Key idea: Logical Environment

- The *logical environment* of a computation is the set of assumptions upon which it rests.
- Every program has a logical environment.
- In AI programs, a substantial fraction of that logical environment is represented explicitly in the program’s data structures.
- How to represent and manipulate logical environments is a key issue in problem solver design.
Stack model of logical environments

- Each operation that makes an assumption pushes a new stack frame.
- When the operation is finished, the stack is popped.
- Supports depth-first exploration of set of assumptions
Initial state of database

(show P)
(not Q)
(implies (not P) Q)
(implies (not Q) R)
Start of indirect proof attempt

(show P)
(not Q)
(implies (not P) Q)
(implies (not Q) R)

(not P)
After CE on assumption

(show P)
(not Q)
(implies (not P) Q)
(implies (not Q) R)

(not P)  Q
Contradiction
After successful indirect proof

(show P)
(not Q)
(implies (not P) Q)
(implies (not Q) R)
P
Constraints on Assumption Stack

• Every assertion must be made in the newest context.
  – Guarantees retraction when assumption is retracted.

• Conclusions must be drawn in simplest possible logical environment
  – Prevents inappropriate retraction

• Every operation that requires an assumption must push a new context.
  – Otherwise could over-retract
Implementation of assumption stack

• Use redundant rule, fact databases to store contexts
  – Implement as linear lists so that pushing = lambda binding and popping = returning.
  – Associate an integer index with each level of context to provide a mechanism for bounding resources used.
    • No assumptions = level 0
  – Must modify database interface procedures to store facts and rules in appropriate place.
    • Context = 0 means store in global database, otherwise in context database.
  – Must provide mechanism for “lifting” results out of a context to where they are needed.
Let’s look at finter.lsp
Improving efficiency

• In TRE, alignment of logical variable bindings with Lisp variable bindings achieved via `eval` on a constructed `let` statement.
  – Inefficient. Should be able to compile rules.
  – Trick: Turn rules into procedures

• In TRE, pattern-matching was handled via calls to `unify`
  – Observation: We know one of the patterns already.
  – Trick: *Open code* unification
Making rules compilable

• Body of rule is lisp code -- should compile it
  – (Always compile your code! The error-checking alone is worth it.)
• Implementation in TRE won’t do
• Define a separate procedure for the rule body
  – Arguments are the pattern variables used
  – Do it automatically, user just writes rules
• Issues
  – Must create the appropriate environment
  – Must arrange contents of files so that procedures are defined before they are used, because matching facts can already be in the database.
Open code unification

• At compile time we know:
  – Structure of trigger pattern
  – What variables will already be bound
  – Any additional tests required.

• Idea: Create special-purpose procedure which does what unify would do on the trigger pattern

• Rule = match procedure and body procedure
Implementation issues for efficient rules & open-coded unification

• At rule expansion time, must process all subrules.
• Must keep track of variables that will be bound at run-time, to ensure that the appropriate lexical scoping is enforced.
• Tests performed by the unifier must be “unrolled” into a procedure.
• The rule and body procedures must have their argument lists set up correctly.
Fully implementing KM*

• Now we can implement all the inference rules of the KM* system
  – Most rules are normal rules
  – Rules which make assumptions are a-rules

• Let’s look at fnd.lsp…
Homework 2

- Problems 17 and 18, page 148
  - Remember, a few minutes of thought can save days of hacking