ELEC 875
Design Recovery
and
Automated Evolution

Grok and Sgrep
Relational Databases

• On Disk Data Structures
  ◊ optimized for huge databases
    - many millions of records
  ◊ optimized for IT based queries
  ◊ `select avg(sales) from employee where commission > 0.5`
  ◊ `select manager from employee where name = “James Higgins”`
  ◊ allows update to single records
• Spectacular for these types of queries
Program Analysis Queries

- **example**
  - Common Ancestor Subsystem of Two modules
    - equivalent IT query:
      - common boss of two employees
    - requires recursive SQL (in latest version)
  - requires multiple queries to the same table
- updates to single records are rare
- often add entire derived relations to the database
- some individual queries
- Queries often need to use every record in the relation
- Relational DBs not optimized for these types of queries
  - not surprising, very minuscule portion of database use.
Grok

- Initial Version in 1995, Ric Holt
- Optimized for large Databases
  ◊ hundreds of thousands of facts
- Heinlein - Stranger in a Strange Land

- Relational Algebra Calculator
  ◊ Discrete Math
  ◊ Sets and Relations

- Ram Based
  ◊ Queries tend to use entire relations at a time
  ◊ Recursive Queries
Grokh - Input of Relations

- RSF - Rigi Standard Format
  - triple format
  - `funcdef main main.c`
  - `defloc main "main.c:10"
  - `include main.c stdio.h`
  - `calls main foo`
  - `sets foo x`
  - `parameter foo y`

- Automatic discovery of domain and range sets
  - just use names in relations
- Attributes are just another relation
Grok - Input of Relations

- TA - Tuple Attribute format
  ◊ ER based notation
  ◊ Definition of instances
  ◊ Attributes instead of relations

```c
funcdef main main.c
defloc main "main.c:10"

$INSTANCE main func {defloc="main.c:10"}
```

◊ Relations can also be extended

◊ translated to RSF internally
Grok - Input of Relations

- TA -Schema Definition
  ◊ Allows the user to specify the schema of the data
  ◊ Not explicitly checked
  ◊ Schema is also compiled into relations
  ◊ Can write a grok program that checks the data against the schema
    - already done
Grok - Operators

- Sets
  - construction
    ```
    functions = { "main", "foo", "bar", "bat" }
    vars = { "m", "x", "y" }
    refs = {"x", "z"}
    ```
  - union/intersection/complement
    ```
    ents = functions + vars
    vrefs = vars ^ refs
    vnrefs = vars - refs
    ```
  - cardinality
    ```
    numvars = #vars
    ```
  - sets can be read and written to files, one entity per line
Grok - Operators

- Relations
  - Cross Product
    \[ \text{foo} = \text{functions} \times \text{refs} \]
  - Relations are sets of tuples, so all set operators work on relations in the obvious way
  - Domain/range (codomain)
    \[ \text{f} = \text{dom} \text{ foo} \]
    \[ \text{r} = \text{rng} \text{ refs} \]
  - Relation composition
    \[ \text{h} = \text{f} \circ \text{g} = \{ (x,y) \mid y = g(f(x)) \} \]
Grok - Operators

- Relations
  - Id constructor (S is a set)
    \[ r = \text{id }S \quad \Rightarrow \quad \{(x,x)\} \text{ for all } x \text{ in } S \]
  - inverse (n is a relation)
    \[ m = \text{inv }n \]
  - transitive closure
    \[ R^+ \]
  - Transitive, reflexive closure
    \[ R^* \]
Grok - Operators

- Sets and Relations

◊ projection (s is set, R is relation)
  s.R = { y | x in S and (x,y) in R}
  R.s = s . inv R

{“f”,”g”} . invokes == all functions invoked by f and g
{“f”,”g”} . invokes+ = all functions invoked directly or indirectly by f and g
{“f”,”g”} . invokes* = all functions invoked directly or indirectly by f and g including f and g.
Grok - Scripting

- Grok also has a scripting language:
  - conditionals (if)
  - looping
  - arguments
  - file io

- Other numerous options including options to ask for names of sets, relations and variables, string operations, id operations, file I/O
Grep

• problems with grep
  ◊ no syntax awareness
  ◊ grep “date” *.c gets:
    – all variables with date the name
    – all functions with date in the name
    – all comments with date in them
  ◊ scans code line by line. Fast for small file, slow for big systems (limited by I/O speed).

• advantages of grep
  ◊ simple Regular Expression notation, easy for developers to understand
sgrep

- lets grep run on TA database
  - run fact extractor to get TA from code
  - contains an arbitrary model
    - they use the software landscape model
    - could be a Datrix or DMM model too.
- regular expressions can be limited to particular entities
  - variables containing “date” in the name
- regular expressions can be applied to results of queries.
  - all methods from class A that are overridden by class B and contain the “f.*bar” in the return type.
• combination allows us to mix structural (grok queries) and lexical patterns.
  – key relation is the contains relation which is given by the 'in' query verb.
  – need a mapping from the equivalent of contains in the extracted model.
  – similar to the Holt, Fahmy and Cordy paper.
• Contrast back to Lethbridge and Singer
sgrep

- Implementation:
  - Front end for a grok server
  - translates to grok and executes
  - applies pattern matching to result

- Grok is a complex language, sgrep attempts to simplify

- assumes some relation names (contains)
sgrep

Queries:
- pattern is entity --- result is a set
  pattern is run against projection of $INSTANCE
  $INSTANCE x entity
  get* is function right projection
  getChar is * left projection

Start by right projection:
$INSTANCE.{'function'}

then do a regular expression match on result
  simplest query to implement
  Can also returns attributes
sgrep

• Queries:
  - pattern is entity in pattern --- result is a set
    First part is the same as before, but constrained by
    the contains+ relation
    get* is function in parser.c

not clear if
  get* is function in pars*.c
is supported
- clear extension if not.
sgrep

- Queries:
  - pattern is entity <relation> pattern is entity
    --- result is a relation
    find sets for the left and right is and then find those tuples in relation that match..
    * is function <calls> getc is *

  Two sets (based on first query)
  Match against relation
sgrep

- Queries:
  - pattern is entity in pattern <relation> pattern is entity in pattern
    -- result is relation
  
Find sets for left and right and relation
* is function in parser.c <calls> * is function in scanner.c
**sgrep**

- Queries:
  - pattern is entity `<relation+>` pattern is entity
    - result is a relation
  - find sets for the left and right is and then find those tuples in transitive closure of relation that match..

  * is function `<calls+>` getc is *
sgrep

• Does not handle composite relation queries
  – what variables are modified when I call this function?
  – composes *calls*+ and *sets*
the types of all fields of subclasses of the class 'transact'