

## Brewer/Hellerstein CS262 Spring 2008: Declarative Networking

- Big idea here: declarative languages have much broader applicability than previously realized!
  - Gray's Turing Lecture: Automatic Programming as a Grand Challenge
    - Lampson's response in JACM 50'th anniversary
    - Can the "brittle" declarative languages of data management extend elsewhere?
    - Declarative Domain-Specific Languages
      - How many domains must we knock off before we decide it's no longer a DSL?
        - Overlay Nets, Sensornets Distributed Systems, Compilers, ML, Robotics, Security ...
- The SOSP 2005 paper on P2
  - The first shot at making declarative networking real
  - We had earlier done Declarative Routing using PIER, our p2p query engine, but programs were basically just illustrations of a concept
- Set ourselves the challenge to do complex overlays, not just toy examples: Chord!
  - Click meets Datalog
  - Much work since then
    - Clean up semantic messes: bottom-up design of a declarative system had pros/cons.
    - Realize the promise: automatically optimize, prove properties, show more stuff we can easily do
- Background: Datalog
  - Post-dates Prolog. An attempt to do 2 things to Prolog:
    - remove all operational semantics
    - allow for batch-oriented (dataflow) processing, a la database engines
    - basic SQL plus recursively defined views
  - Basics:
    - Relational structures
    - Extensional Data Base (**EDB**): stored "facts". Think of tables in SQL. Expressed via "facts".
      - parent("George HW", "George W").
    - Intensional Data Base (**IDB**): derived facts (deduction). Think of "views" in SQL. Strictly logical entities -- keep lazy evaluation in mind, it will keep you from getting confused.
    - scalars: variables (capital letters), constants
    - atoms: p(a1, ..., an) where p (lowercase letters) is a relation(predicate) name, a1...an are variables or constants
    - rules: "head" and "body"
      - ancestor(X, Y) :- parent(X,Y).
      - ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).
  - Things to keep in mind
    - 1 all uses of a predicate have same arity
    - 2 multiple appearances of predicate in head = disjunction (UNION)
    - 3 body predicates implicitly involved in conjunction (AND)
    - 4 **unification** across body predicates of a rule via variable-matching (equi-join)
    - 5 reordering rules in a program does not change meaning
    - 6 reordering predicates in a body does not change meaning
  - Now, the syntax is mathematical logic, meaning that it's just a grammar defining legal sentences. We need a "model" to define what we mean by this grammar.
    - want model to be consistent with existence/non-existence of facts
    - what is a good model?
      - Minimal models are good (can't remove any fact and stay consistent)
      - Unique Minimal Model would be nice!
    - simple semantics that "work" (i.e. provide a unique minimal model):
      - an atom is true for "bindings" of its variables iff the arguments form a tuple of the relation
      - whenever a binding of values to each variable makes all subgoals in the body true, the rule asserts that the head is also true
      - Nice: clean, intuitive declarative semantics that has attractive logical properties
  - Even better: this model arises naturally from a simple dataflow implementation!
    - a.k.a. "forward chaining" or "bottom up" (as opposed to Prolog's "backward chaining" or "top down")
    - and the implementation is amenable to much optimization. (yay!)
    - **Semi-Naive Evaluation**
      - basic idea is to keep "joining in" newly derived tuples against previously derived tuples until you hit a "fixpoint".

## Brewer/Hellerstein CS262 Spring 2008: Declarative Networking

- Visualize descending the tree in "levels": find children of root. Then grandchildren. Then great-grandchildren. (In a cyclic relation, remove duplicates as you go.)
- Isn't the ancestor example dorky?
  - But look:
    - $\text{path}(X, Y) :- \text{link}(X, Y).$
    - $\text{path}(X, Y) :- \text{path}(X, Z), \text{link}(Z, Y).$
  - Or even better:
    - $\text{path}(X, Y, Y, \text{Cost}) :- \text{link}(X, Y, \text{Cost}).$
    - $\text{path}(X, Y, Z, C1+C2) :- \text{path}(X, Z, \text{NextHop}, C1), \text{link}(Z, Y, C2).$
  - Now add this:
    - $\text{minCost}(X, Y, \text{min}\langle C \rangle) :- \text{path}(X, Y, \text{NextHop}, C).$
    - $\text{bestPath}(X, Y, \text{NextHop}, \text{Cost}) :- \text{path}(X, Y, \text{NextHop}, \text{Cost}), \text{minCost}(X, Y, \text{Cost}).$
    - Hey, that's shortest-path routing -- a network protocol! (sort of.. still centralized)
- A side note: aggregation, negation & stratified versions
  - The "min" above complicates our semantics
    - so does NOT p() -- i.e. negation (same as  $\text{count}\langle * \rangle = 0$ )
    - "non-monotonic": i.e. the answer (or its subsequent use) may not grow as we deduce more facts, recursively
  - the min is computed over what set exactly?
    - idea: identify the minimal model over path, and define minCost on that!
    - draw a **rule dependency graph**.
    - partition the graph into **strata**, such that each stratum contains no negation/aggregation edges internally
      - if the resulting graph of strata is acyclic, the program is **stratifiable**
    - stratified semantics define each stratum with respect to the (unique) minimal model of the preceding strata
    - i.e. compute fixpoints following the partial order on strata
- So far we have seen...
  - Datalog
  - With stratified negation/aggregation
  - It can represent an important networking task: routing on shortest paths
    - But in a centralized implementation. Kind of silly!
    - How can we get a distributed version?
- Datalog + Horizontal partitioning = Networking.
  - partitioning a la classical parallel DBs, a.k.a. "Map".
  - the trick: partition on a field of type "address" -- then "Map" is the declarative version of "Send", says where data shall live.
    - $\text{link}(@X, Y).$
    - $\text{path}(@X, Y, Y, P, C) :- \text{link}(@X, Y, C), P = \text{f\_concatPath}(\text{link}(X, Y, C), \text{nil}).$
    - $\text{path}(@X, Y, Z, P, C1+C2) :- \text{link}(@X, Z, C1), \text{path}(@Z, Y, N, P2, C2), P = \text{f\_concatPath}(\text{link}(X, Z, C1), P2).$
  - localization
    - the recursive rule above has to do a distributed join
    - use a syntactic rewrite to describe that. replace the last rule with:
      - $\text{link\_d}(X, @Z, C) :- \text{link}(@X, Z, C).$
      - $\text{path}(@X, Y, Z, P, C1+C2) :- \text{link\_d}(X, @Z, C1), \text{path}(@Z, Y, N, P2, C2), P = \text{f\_concatPath}(\text{link}(X, Z, C1), P2).$
      - OMG! That is Path Vector routing, a classic internet protocol used in BGP (among other places).
    - Wait, it gets better! Could have done join reordering on the recursive rule, and then localized.
      - $\text{path}(@X, Y, Z, C1+C2) :- \text{path}(@X, Z, P1, C1), \text{link}(@Z, Y, C2).$
      - rewrite to:
        - $\text{path\_d}(X, @Z, P1, C1) :- \text{path}(@X, Z, P1, C1).$
        - $\text{path}(@X, Y, Z, C1+C2) :- \text{path\_d}(X, @Z, P1, C1), \text{link}(@Z, Y, C2), P = \text{f\_concatPath}(P1, \text{link}(Z, Y, C2)).$
      - This is the key to what's called Dynamic Source Routing, which is widely used in wireless networking.
    - This is more than mere conciseness!
      - We have found an avenue to treat protocol design as a query optimization problem
        - Just say what routes you want, and pick the protocol to find them by understanding performance issues
        - Critical, because the world is getting more complicated than 2 classes of connections (wireless vs. Internet!)
  - Some Declarative Networking details we're glossing over

## Brewer/Hellerstein CS262 Spring 2008: Declarative Networking

- Time, state, network delays, and effects on semantics
  - Tricky! Much of this is confused in the early papers. Coming clearer recently.
- **Soft state**: a key persistence model in networking
  - data items have expiry as of some time after *arrival at the destination*
  - persistence is achieved through periodic sender retransmission
  - passive failure detection -- handles disconnection elegantly
  - soft-state IDB?
    - soft-state heads? head/body differ in their lifetimes?
    - events are soft-state with an "instantaneous" expiry!
- Time in P2: One local event per clock "tick"
  - clock events from **periodic**: an infinite clock eventstream
  - network arrival events
  - take the current tick's single event as a one-fact table
  - expire stale soft state
  - treat the result as EDB, run datalog to (local) fixpoint
  - expire the event
  - How about distributed time??
- How do we detect distributed fixpoint? Esp for strata?
  - Can make this crisp via consensus (Paxos), but may not want to.
- Why does Datalog fit so well for so many things?
  - networks are graphs, want recursive queries on graphs
    - routing example above
  - (repeated) asynchronous communication is basically join: match pending requests to stream of responses
    - `pending_request(@X, Y, Args) :- request(@X, Y, Args).`
    - `request(X, @Y, Args) :- request(@X, Y, Args).`
    - `response(@X, Y, Result) :- request(X, @Y, Args), Result = f_doit(@Y, Args).`
    - `answer(@X, Args, Result) :- pending_request(@X, Y, Args), response(@X, Y, Result).`
  - forward chaining is essentially dynamic programming
    - and D.P. optimizations can be simply expressed in datalog with stratified aggregation
    - note that D.P. is used lots of places including database query optimization, various machine learning algorithms, etc.
  - Note that monotonic Datalog can express PTime computations
    - Do you really want to do more than that over a network??
- Datalog variants exploding in the last 5 years!
  - Declarative networking@Berkeley/Intel/Texas/Rice/etc.
    - Overlays, sensornet protocols
    - Distributed systems protocols (Paxos, Chandy-Lamport Snapshots, various forms of replication)
  - Machine Learning algorithms (JHU, Berkeley)
    - Inference (Junction trees, loopy BP) & Distributed Inference
    - Natural Language Processing (Dyna@JHU)
  - Compiler Analysis@Stanford
    - again, recursive queries, this time on call graphs
    - bddbdb
  - Security/Access protocols
    - Stanford, MSR
  - Modular Robotics
    - Meld@CMU
  - Datalog metacompilation@Berkeley
    - Evita Raced
      - e.g., simple datalog programs to build a dependency graph and check for stratification
      - e.g. Selinger-style dynamic programming
      - e.g. magic sets
  - OK, is this a set of DSLs?

**Brewer/Hellerstein CS262 Spring 2008: Declarative Networking**

- Well, they're mostly yucky for traditional programmers
- Not unified
- How much of this is a syntax problem? how much is about domain specific optimizations?
  - Does metacompilation help with the latter?