Declarative Languages (For Building Systems)

Robert Soulé

New York University

What is a Declarative Language?

We can broadly classify languages into two categories:

- Imperative Languages
  - von Neuman (C, Ada, Fortran)
  - scripting (Perl, Python)
  - object-oriented (C++, Java)
- Declarative Languages
  - functional (Lisp/Scheme, ML, Haskell)
  - dataflow (Id, Val)
  - logic, constraint based (Prolog, spreadsheets)
  - template based (XSLT)

What is a Declarative Language?

- Exact definition is fuzzy.
- Intuitively, they describe the "what" and not the "how".
- If *algorithm* = *logic* + *control* then in declarative programming, programmers give the logic, but not the control.

Types of Declarativity

Languages can be declarative in different ways:

- Logic languages (Prolog, Datalog)
    - Declare relationships
    - Ask questions about the relationships
    - Don't specify how to get the answer
- Data oriented languages (Pig Latin, SQL)
    - Specify criteria for desired data
    - Don't specify how to find the data
- Dataflow languages (nesC, Click)
    - Define relationships between components
    - Encapsulate non-declarative details in the components

Declarative Languages for Systems

- Normally, we think of imperative languages for systems.
- Declarative languages can be surprisingly useful.
  - Free the programmer from implementation details.
  - Provide better abstractions, concise code.
  - Offer correctness guarantees, or increased reliability.
- Benefits come at a performance cost.
  - Compiler can implement optimizations to offset this.

Overview of this talk

- Look at 5 systems that use declarative languages.
- Why these 5 systems?
  - Looking ahead, part 2 of my talk is on my work with Overlog.
  - Discuss the relationship between Overlog and these systems.
  - Discuss how each system uses declarative programming.

Outline

Overlog

Declarative Networking: Language, Execution and Optimization Boon Thau Loo, Tyson Condie, Minos Garofalakis, David A. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe and Ion Stoica. *SIGMOD 2006* [1]

[1]This paper discusses the formal semantics of Overlog, but it is worth examining the background problem first.

P2 Background

- Problem - Difficult to design, build, and deploy overlay networks.
- Goal - Provide a convenient, higher level abstraction for creating overlay networks.
- Solution - Take a cue from databases, and view networking as database query processing.

P2 Overview

- Every node in the network has a runtime:
  - In memory database stores tuples.
  - Messages are sent asynchronously as tuples.
  - Programs are specified in a logic-based language.
- Logic languages are a good fit:
  - Recursion works well for for network algorithms (*shortest path*).
  - Natural to specify routing as a policy.

Logic language

- Datalog[2] is a logical query language.
- Consisting of a series of rules:
  - A single relational atom, called the *head*
  - Followed by the symbol :-, read as “if”
  - Followed by a body consisting of one or more atoms, called subgoals, either relational or arithmetic
- Subgoals are connected by the logical conjunction.

[2] More details available upon request.

Overlog

- Overlog is a logic-based language for networking.
- It maps Datalog to a networked setting.
- Need to define the syntax and semantics.
  - Syntax - slightly modified Datalog syntax
  - Semantics - Datalog semantics in a distributed setting[3]

[3]More details available upon request.

Overlog Optimizations

- Apply traditional Datalog optimizations:
  - Aggregate Selections
  - Magic Sets[4]
- Apply distributed systems optimizations:
  - Query result caching
  - Opportunistic message sharing

[4] More details available upon request.

Overlog as a Declarative Language

- Overlog build distributed systems declaratively:
  - Declaring data relationships.
  - Performing actions as a result of those relationships.
- Let's look at another approach...

Outline

Mace

Mace: Language Support for Building Distributed Systems
Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. *PLDI 2007*

Mace

A different solution to a similar problem:

- Problem - Building distributed applications is difficult and error-prone.
- Goal - Abstract away many of the challenging details.
- Solution - Use a state machine specification for building systems.

Mace: More detail

- Mace is a C++ language extension that structures layered services as state machines.
- The state machine template provides code blocks for specifying the interface, lower layers, messages, state variables, inconsistency detection, and transitions.
- The Mace compiler provides high level validation of the architecture design by analyzing the state machine specifications.

Mace: More detail

Mace offers support for ensuring correctness.

- Uses *aspects* for run time inconsistency and failure detection.
- State machine representation facilitates model checking.
- MaceMC model checker detects liveness violations.

Mace as a Declarative Language

Mace build distributed systems declaratively:

- Encapsulate non-declarative details in the components with well defined interfaces.
  - i.e. state machine templates
- Declaratively specify program state.
- Declaratively specify transitions between states.

Mace vs. P2/Overlog

- Lines of code for Chord:
  - Mace: 250, Overlog: 47, Original: 3400
- Safety
  - Mace SM facilitates model checking.
  - Open question for Overlog.
- Performance
  - Mace has no clear transition to concurrent execution.
  - Static analysis of Overlog for concurrent execution.

Mace vs. P2/Overlog

Qualitative argument:

- "90 percent of the students successfully completed the project and a majority expressed a preference for programming in Mace relative to Java or C++."
- How hard is it to program with Overlog's logic rules?

Imperative/Declarative Balance

- Systems programmers are used to programming imperatively.
- Is it hard for them to think declaratively?
- Let's look at the next system...

Outline

Pig Latin

Pig Latin: A Not-So-Foreign Language for Data Processing
Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins. *SIGMOD 2008*

Pig Latin

- Problem - Need for ad-hoc data analysis on large data sets.
- Goal - Create a data processing language that is natural for systems developers to use.
- Solution - Combine declarative SQL-style queries with procedural programming style.

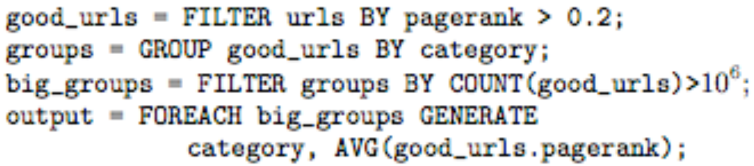
Pig Latin: More detail

- Each line of code performs a single data transformation
    - Stylistic departure from SQL
- Pig Latin offers a nested data model
    - Violating the first normal form restriction enforced by most databases.
    - Allows programmers to use a map-like data structure.
- Pig Latin supports *user-defined functions*.

Pig Latin: More detail

- Programs written in Pig Latin are executed on *Hadoop*.
    - Different execution platforms possible.
- Integrated debugging environment.
    - Generates concise example data.

Pig Latin: Example



```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
                category, AVG(good_urls.pagerank);
```

Pig Latin vs. SQL

Also makes a qualitative argument:

- "I much prefer writing in Pig [Latin] versus SQL. The step-by-step method of creating a program in Pig [Latin] is much cleaner and simpler to use than the single block method of SQL. It is easier to keep track of what your variables are, and where you are in the process of analyzing your data." - Jasmine Novak, Engineer, Yahoo!

Pig Latin: UDFs Revisited

- Pig Latin provides standard operators:
  - grouping, filtering, joining, per-tuple processing
- To accommodate specialized data processing tasks:
  - "extensive support for user-defined functions"
  - UDFs written in Java
  - Plans for arbitrary language interface

Encapsulating Non-Declarative Code

- Both Pig Latin's UDFs and Mace's templates allow developers to encapsulate non-declarative code.
- Lets look at two final systems that emphasize this model.

Outline

Click

The Click modular router Eddie Kohler, Robet Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. *TOCS 2000*

Click

- Problem - Routers are expected to do more than route packets, and routing policies are under active research.
- Goal - Create a framework for building extensible routers.
- Solution - Modularize functionality into re-usable components, and use a data-flow language for connecting components.

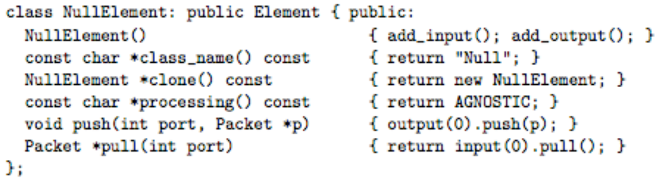
Click: More detail

- Modularize functionality into re-usable components called *elements*.
  - Each *element* performs a simple task, such as packet identification, or queuing.
- Reuters are configured by using a domain specific language.
  - Declare which elements are used, and how they are connected.
- Connections between elements may be either *push* or *pull*.

Click: Element

- Subclass of C++ class *Element*.
- About 20 virtual functions.
- Three (*push*, *pull*, and *run_scheduled*) are used during router operation.
- The others are used for identification, statistics, configuration, etc.

Click: Example Element



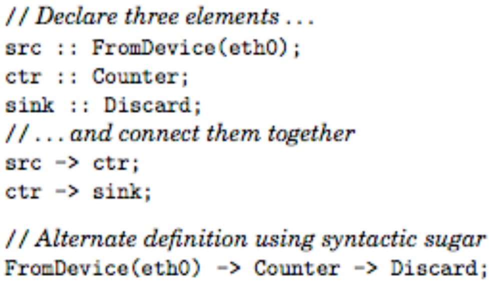
```
class NullElement: public Element { public:
  NullElement()                     { add_input(); add_output(); }
  const char *class_name() const    { return "Null"; }
  NullElement *clone() const        { return new NullElement; }
  const char *processing() const    { return AGNOSTIC; }
  void push(int port, Packet *p)    { output(0).push(p); }
  Packet *pull(int port)            { return input(0).pull(); }
};
```

Fig. 7. The complete implementation of a do-nothing element: *Null* passes packets from its single input to its single output unchanged.

Click: Language

- Declarative language
- Two constructs:
    - *Declarations* create elements
    - *Connections* show how they should be connected

Click: Language Example



```
// Declare three elements . . .
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
// . . . and connect them together
src -> ctr;
ctr -> sink;

// Alternate definition using syntactic sugar
FromDevice(eth0) -> Counter -> Discard;
```

Fig. 6. Two Click-language definitions for the trivial router of Figure 2.

Click: Evaluation

- Overhead for passing packets between elements and unnecessarily general element code.
- The performance evaluation shows that an almost standards compliant IP router built with Click performs only slightly worse than the standard Linux router.

Click: Why is the performance ok?

- A preprocessor can validate configurations.
    - Check that pull elements are connected to pull elements.
    - Ensure every output has a connection.
- A preprocessor can identify optimizations.
    - Output a new configuration file.

Declarative Compilers

- One of the major benefits of declarative languages is that compilers can do the work for you.
  - Click's configuration optimizer.
  - Mace's specification verification.
  - Overlog's query plan optimization.
- Our final system uses this for resource constrained environments.

Outline

nesC

The nesC Language: A Holistic Approach to Networked Embedded Systems David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. *PLDI 2003*

nesC

- Problem - Building applications in resource constrained environments.
- Goal - Create a flexible language for programming sensor networks.
- Solution - Use a component model with an emphasis on concurrency and an event driven architecture.

nesC: More detail

- nesC is a C language extension used to implement TinyOS.
- Applications are built from a set of reusable components.
- There is no dynamic memory allocation.
- Call graph is fully known at compile time.
  - Allows for accurate program analysis.
- Applications are fundamentally event driven.
  - Compiler ensures no race conditions.

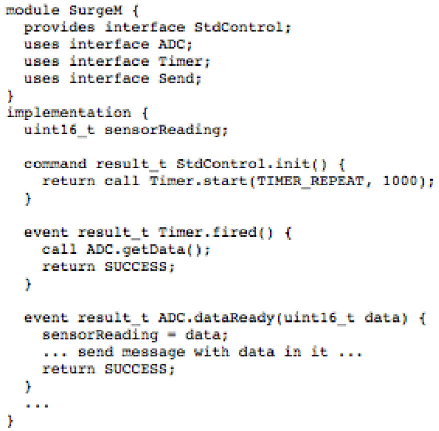
nesC: Components

- TinyOS provides a set of reusable systems *components*.
- Components are either:
  - *Modules* provide application code (implementing an interface).
  - *Configurations* wire components together.
- Most components are software modules, some are wrappers around hardware.
- Unused OS services can be excluded from the application.

nesC: Interfaces

- Components provide and use *interfaces*.
- Interfaces are *bidirectional*: contain both *commands* and *events*.
  - Example: Timer interface defines the *stop* and *start* commands, and *fired* event.
- Similar to Mace's layered state machines.

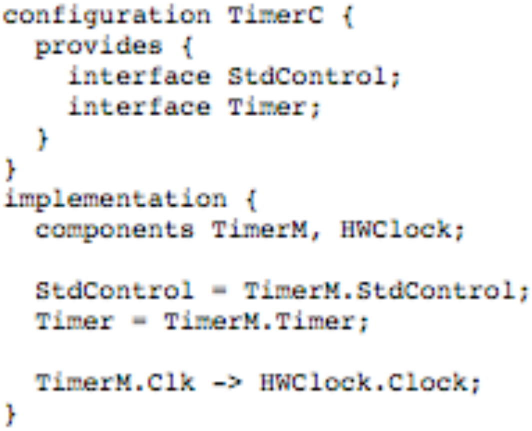
nesC: Example



```
module SurgeM {
  provides interface StdControl;
  uses interface ADC;
  uses interface Timer;
  uses interface Send;
}
implementation {
  uint16_t sensorReading;

  command result_t StdControl.init() {
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  event result_t Timer.fired() {
    call ADC.getData();
    return SUCCESS;
  }

  event result_t ADC.dataReady(uint16_t data) {
    sensorReading = data;
    ... send message with data in it ...
    return SUCCESS;
  }
  ...
}
```

Figure 4: Simplified excerpt from `SurgeM`**.**

nesC: Example



```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}
implementation {
  components TimerM, HWClock;

  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;

  TimerM.Clk -> HWClock.Clock;
}
```

TinyOS: Concurrency

- Two sources of concurrency in TinyOS:
  - *Tasks* are deferred computations that run to completion, and do not preempt each other.
  - *Events* also run to completion, but may preempt other tasks or events.
- No blocking operations
  - Tasks execute non-preemptively.
  - Contention is handled by explicit rejection of concurrent requests.

nesC: Concurrency

- Definition:
  - *Asynchronous Code* is code reachable from at least one interrupt handler.
  - *Synchronous Code* is code only reachable from tasks.
- Note that:
  - Synchronous Code is atomic with respect to other Synchronous Code.
  - Any update to shared state from an AC is a potential race condition.
  - Any update to shared state from an SC that is also update from AC is a potential race condition.

nesC: Compiler Support

- Use the *atomic* key word disables interrupts.
- Compiler enforces that data accessed by AC is in atomic statement.

nesC: Evaluation

- *Safety*
    - Race detection caught 103 race conditions, 53 false positives.
    - Most errors came from non-atomic state transitions.
- *Performance Improvements*
    - Application call graph eliminates unreachable code and module boundary crossings.
    - Inline small functions.
    - Common subexpression elimination, constant propagation.
    - Code reduction about 10%, CPU reduction 15% - 34%

Conclusions

- Declarative languages are effective for building systems.
    - Free the programmer from implementation details.
    - Provide better abstractions, concise code.
    - Compiler can offer correctness guarantees, or increased reliability.
- Trade-off between declarativity and implementation necessity.
    - Need to find the right balance.
- Benefits come at a performance cost.
    - Compiler can implement optimizations to offset this.

The End

Questions?

This slide intentionally left blank

Datalog

Datalog Tutorial

Datalog

A First Course in Database systems Jeff Ullman, and Jennifer Widom. *Prentice Hall, 2007*
Datalog is a logical query language, consisting of a series of rules.

Datalog Atoms

There are two types of *atoms* in Datalog.

1. Relational atoms, are predicates that represent relations.
2. Arithmetic atoms are comparisons between two atomic expressions.

Datalog Rules

1. a single relational atom, called the *head*,
2. followed by the symbol :-, read as “if”,
3. followed by a body consisting of one or more atoms, called subgoals, either relational or arithmetic.

Subgoals are connected by the logical conjunction *AND* and may contain the optional logical operator *NOT*.

Extensional and Intensional Predicates

- *Extensional Predicates* are predicates whose relations are stored in a database
- *Intensional Predicates* are predicates whose relations are computed by applying datalog rules

Safety Condition

Every variable that appears anywhere in the rule must appear in some non-negated, relational subgoal of the body.

Naive Evaluation

Given an EDB:

1. Start with all IDB relations empty.
2. Instantiate variables of all rules in all possible ways. If all subgoals become true, infer that the head is true.
3. Repeat step 2 as long as new IDB facts can be inferred

Note that:

- Step 2 is finite as long as all rules are *safe*
- The limit of 1-3 is the *Least fixed point* of the rules and EDB.

Semi Naive Evaluation

1. Initialize IDB relations by using only those rules without IDB subgoals.
2. Initialize the Δ-IDB relations to be equal to the corresponding IDB relations.
3. In one round, for each IDB predicate p:
    1. Compute new Δ-P by applying each rule for p, but with *one* subgoal treated as a Δ-IDB relation and the others treated as the correct IDB or EDB relation. (Do for *all* possible choices of the Δ-subgoal
    2. Remove from new Δ-P all facts that are already in P
    3. $P := P \cup \Delta P$
4. Repeat step 3 until no changes to any IDB relation.

Overlog

Overlog Semantics

Overlog: More detail

- A *location specifier* is an attribute of type address in a predicate that indicates the network storage location of each tuple.
- A *link relation* is a stored ("extensional") relation ($link(@src, @dst, ...)$) representing the connectivity information of the network being queried.
- *Local rules* are rules that have the same location specifier in each predicate, including the head.

Overlog: More detail

- A *link literal* is a relation that appears in the body of a rule prepended with the "#" sign.
- A *link restricted rule* is either a local rule, or a rule with the following properties:
    - There is exactly one link literal in the body
    - All other literals have their location specifier set to either the source or destination field in the link literal.

Overlog: More detail

An NDlog program is a Datalog program with the following constraints:

1. *Location Specificity*: Each predicate has a location specifier as its first attribute.
2. *Address Type Safety*: A variable that appears once in a rule as an address type must not appear elsewhere as a non address type.
3. *Stored Link Relations*: Link relations never appear in the head of a rule with a non-empty body. (i.e. they are stored, not derived).
4. *Link Restriction*: Any non-local rules in a program are link restricted by some link relation.

The semantics of NDlog are those of Datalog.

Overlog: More detail

What are those semantics?

- Semi-naive fixpoint evaluation.

But applied to a distributed setting:

- *Buffered Semi-naive* Same as SN but a node can start a local SN iteration at any time its its input queue is non-empty.
- *Pipelined Semi-naive* Same as SN but a node can start a local SN iteration as soon as a tuple is received.

Overlog

Magic Set

Magic Set

Cousins of the Same Generation Relation

- sg(U,V) means that U and V are cousins, i.e. have a common ancestor W, and the lines of descent from U to W and U to V are the same number of generations.
- Anyone is their own cousin.
- No assumption that parenthood is well organized by levels (someone can marry their grandchild).
- No assumption of unique number of generations between individuals.
- No assumption of acyclic graph.

Magic Set

```
r1:  sg(X,X).
r2:  sg(X,Y) :- par(X,X1), par(Y,Y1), sg(X1,Y1).

sg(a,W)?
```

Magic Set

Reorder goals so Prolog can terminate:

```
r2':  sg(X,Y) :- par(X,X1), sg(X1,Y1), par(Y,Y1).

sg(a,W)?
```

Magic Set

Top down, "backward chaining", Prolog evaluation:

- Consider each parent of a (say b and c).
- Recursively find all b's cousins, then c's cousins, and children of both.
- Since b and c may have ancestors in common, there may be repeated work.
- The running time is exponential to the number of individuals in the database.
- Discovers all "proofs", not all "answers".

Magic Set

Bottom up, "forward chaining", Datalog evaluation:

- Start assuming only facts in the database (par relation).
- sg is initially an empty set.
- Apply the rules, compute the join and union, repeat.
- Runs in polynomial time.
- Better, but generates many useless facts.

Magic Set

Intuition for optimization:

- We only want the "relevant facts".
- Relevant facts are those that are essential to establishing a fact in the answer.
- If "d" and "f" are not an ancestors of "a", we don't care about sg(d,f).
- Thing of a's cone.

Magic Set

If we encounter a goal sg(b, W), where b is an ancestor of a, and apply r2, we get the new goal:

```
par(b,X1), par(W,Y1), sg(X1,Y1).
```

X1 is a parent of b, and an ancestor of a

Magic Set

In rules, that looks like:

```
r3:  magic(a).
r4:   magic(U) :-  magic(V), par(V,U).
```

Magic Set

We can re-write r1 and r2 to insist that values of the first argument of sg are in the magic set.

```
r5:  sg(X,X) :- magic(X).
r6:  sg(X,Y) :- magic(X), par(X,X1),
                         par(Y,Y1), sg(X1,Y1).
```

Magic Set

- Not quite "magic".
- There is a general algorithm for finding these sets, and performing the transformation.
- Sometimes, no transformation is desirable.

Relational Algebra

Relational Algebra

Relational Algebra

The operations of relational algebra can be categorized in four groups:

1. *Set Operations* i.e. union, intersection, and difference
2. *Operations to remove part of a relation* i.e. selection (σ) and projection (π)
3. *Combine tuple of a relation* such as Cartesian product and join
4. *Renaming* Changes the names of the attributes or the name of the relation

Other Operations

There are other operations which do not fit into the categories states above.

1. *Group By* Group tuples by an attribute
2. *Aggregations* aggregation operations (MIN, MAX, COUNT, AVG, SUM) are included in most query languages.

Sets and Bags

We tend to think of the set operations as acting on *sets*. For example $R \cup S$ is the *union* of R and S. That is, its the set of elements that are in either R or S. However, if an element is present in both R and S, it appears only once in the *union*. Often, though, in databases, we use *bags*, which like a set has unordered elements, but allows more than one occurrence of an element. One reason for using bags is performance, as it is extra processing to remove duplicates.

Relational Algebra and Datalog

Given the schema:

$Movies(title,\ year,\ length,\ genre,\ studioName,$
$producer)$

the Datalog rule

$LongMovie\ (t,y)\ \text{:-}\ Movie(t,y,l,g,s,p)\ AND\ l \geq 100$

is equivalent to the relational algebra expression

$LongMovie := \pi_{title,year}(\sigma_{length \geq 100}(Movies))$

Relational Algebra and Datalog

Note that each of the relational-algebra operators can be expressed by one or more Datalog rules.
For example, union $A \cup B$ can be expressed as two rules

$$U(x,y,z) \text{ :- } A(x,y,z)$$
$$U(x,y,z) \text{ :- } B(x,y,z)$$

Relational Algebra and Datalog

In standard Datalog, there are no aggregation functions, or group by functions. Also, there is no support to remove duplicates. One significant difference between Datalog and relational algebra is that Datalog can express recursion, while relational algebra cannot.

PATH(X,Y) :- EDGE(X,Y)
PATH(X,Y) :- EDGE(X,Z) and PATH(X,Y)