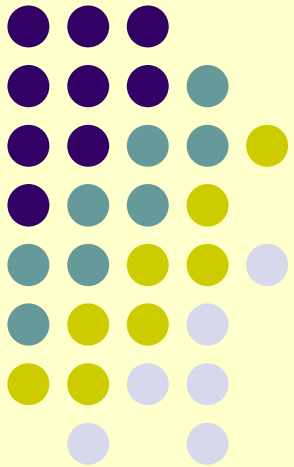
My Foray Into Declarative Languages

...and what I learned about language design

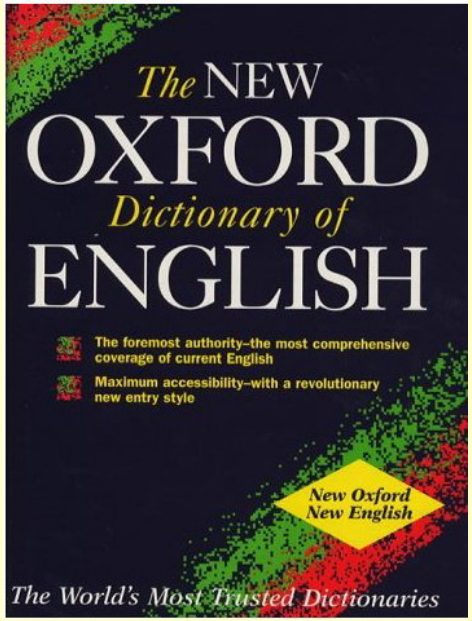
Yannis Smaragdakis
University of Athens



European Research Council
Established by the European Commission

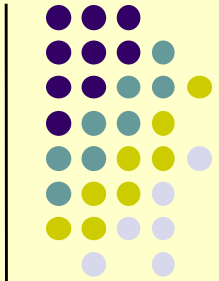
“Declarative”?

“denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed.”



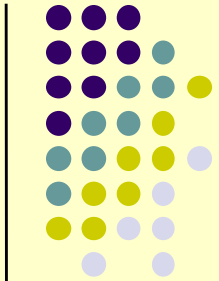
- high-level
- what, not how
- no control-flow, no side-effects
- specifications, not algorithms

What Am I Doing in This Space?

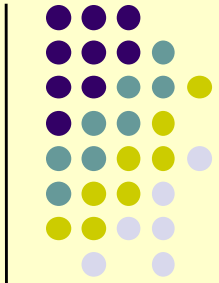


- Before 2008: nearly nothing
  - mixin layers, generics and meta-programming, domain-specific languages, virtual memory, caching algorithms, FC++, automatic partitioning, middleware semantics, automatic testing, symbolic execution, …
- Very little to do with declarative languages
  - barring minor consulting for LogicBlox Inc.

Since Then…

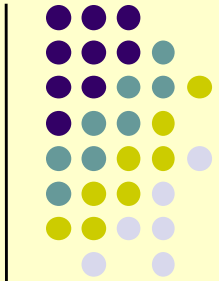


- Doop: declarative static analysis (for Java and now C/C++)
- DeAL: logic-based language for computation over heap structures during GC time
- PQL: declarative, fully parallelizable language over a Java heap
- Academic liaison for LogicBlox
- Lots of other research expressed declaratively
  - also domain-specific work



Sample of Declarative Data Points

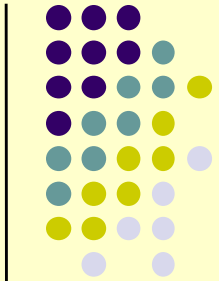
DeAL[OOPSLA'10]



- First-order language for heap computations
  - full connectives, no quantifier nesting, ***R*** predicates (reachable)
  - disjointness assertion
- Guarantee: all programs executable within a single traversal of the heap, side-effect of GC!
- Example: all objects in 'HttpSession' are serializable, none are threads

```
forall Object x: R(HttpSession)[x] ->
        (x instanceof Serializable) &&
        !(x instanceof Thread)
```

PQL[ECOOP'12, CC'15]

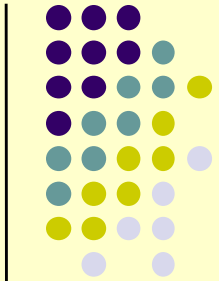


- Unrestricted first-order logic over the Java heap
- Guarantee: automatically parallelizable
- Example: set intersection with filtering

```
Set<Item> intersection =
  query (Set.contains(Item e)):
        s0.contains(e) && s1.contains(e)
                       && !e.is_dead;
```

- Order of evaluation?

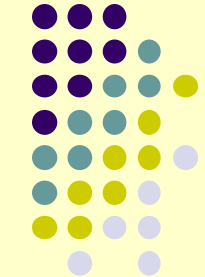
LogicBlox



- Company developing Datalog(-uesque) platform
  - language, optimizer (think: JIT), DB
  - all applications developed declaratively (even UI)
- Datalog: first-order logic + recursion
  - expressiveness-wise: superset of all prior
  - captures PTIME complexity, Turing-complete with simple extensions
  - declarative: order of rules or clauses irrelevant (!Prolog)
- LogicBlox recently sold for ~$150M
  - most value in applications: majority of top retailers worldwide have deployed LogicBlox apps

Static Analysis in Datalog

[OOPSLA'09, PLDI'10, POPL'11, OOPSLA'13, PLDI'13, PLDI'14, SAS'16, …]

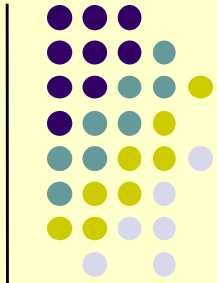


- Datalog-based analysis frameworks for Java, C, C++

DOOP

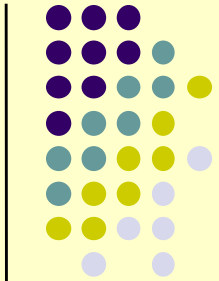
- 2-3K logical rules (20-25KLoC)
- Very high performance (often 10x over prior work)
- Sophisticated, very rich set of analyses
  - subset-based analysis, fully on-the-fly call graph discovery, field-sensitivity, context-sensitivity, call-site sensitive, object sensitive, thread sensitive, context-sensitive heap, abstraction, type filtering, precise exception analysis
- High completeness: full semantic complexity of Java
  - jvm initialization, reflection analysis, threads, reference queues, native methods, class initialization, finalization, cast checking, assignment compatibility

http://doop.program-analysis.org



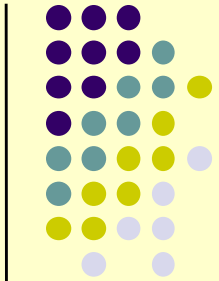
Back To Our Group (Language Design) …

Quotes From "Blue. No! Yellow!"

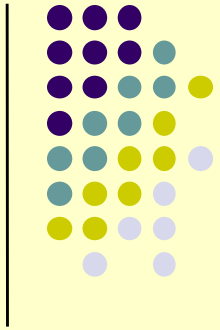


- *"[W]e've passed the point of diminishing returns. No future language will give us the factor of 10 advantage that assembler gave us over binary. No future language will give us 50%, or 20%, or even 10% reduction in workload"*
  - **Question 1**: can we get large productivity increases?
  - Also "assembler over binary"???
    Sorry, I don't buy it.

Quotes From "Blue. No! Yellow!"



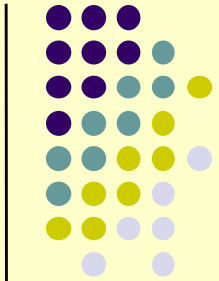
- *"it is difficult to see past the rut that we seem to be in today. … research takes 10/20 years to hit practice"*
  - **Question 2**: are there designs that offer large productivity gains **now**?
- *"all programming languages seem very similar to each other. They all have variables, and arrays, a few loop constructs, functions, and some arithmetic constructs. Sure, some languages have fancier features like first-class functions or coroutines…"*
  - **Question 3**: are there useful languages that have no loop constructs, no arrays, and no functions?



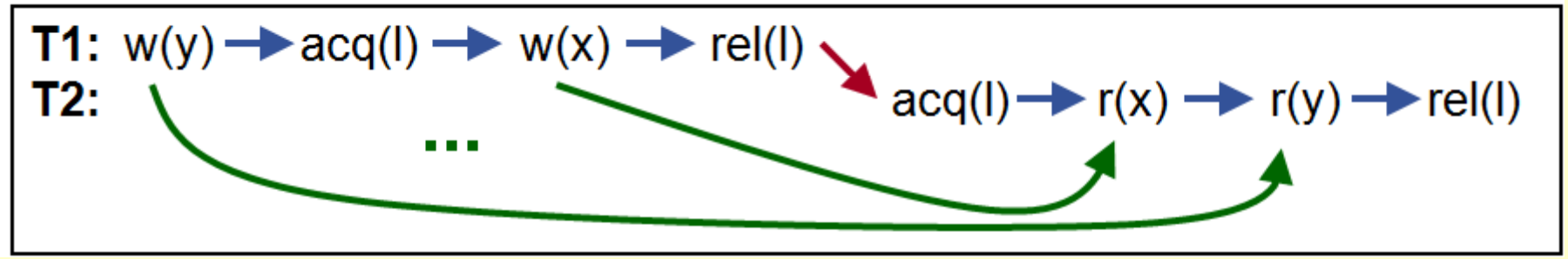
Anecdote I

Based on
"Sound Predictive Race Detection in Polynomial Time"
[POPL'12]

Happens-Before (HB)

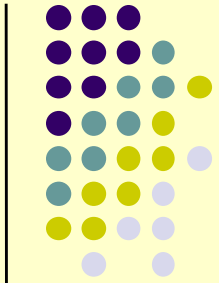


- Lamport's happens-before relation: a very common concept in distributed systems
- HB: 3-part relation (partial order):



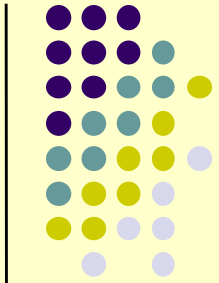
  - all events by the same thread are HB-ordered (in the order observed)
  - a release of a lock happens-before subsequent acquisitions of the same lock
  - HB is transitively closed

Happens-Before Implementation



- Imagine you have the primitive blocks:
  - tables with ordered events, maps from event to its thread, to its type, to the lock it accesses, etc.
- How long would it take you to implement HB?
  - all events by the same thread are HB-ordered (in the order observed)
  - a release of a lock happens-before subsequent acquisitions of the same lock
  - HB is transitively closed

Happens-Before Datalog Implementation

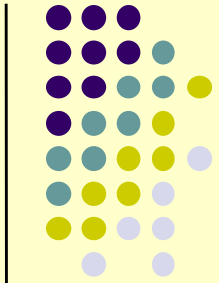


```
HB(e1,e2) <-
  e1<e2, Thread[e1] = Thread[e2].
HB(e1,e2) <-
  UnlockEvent(e1,l), LockEvent(e2,l), e1<e2.
HB(e1,e2) <- HB(e1,e3), HB(e3,e2).
```

- all events by the same thread are HB-ordered (in the order observed)
- a release of a lock happens-before subsequent acquisitions of the same lock
- HB is transitively closed

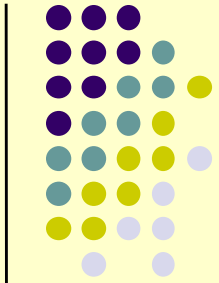
- How long would it take you to write *that* after a year of Datalog practice?

I know, I know…



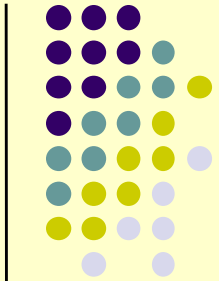
- Lots of possible objections:
  - Amdahl's law / diminishing returns: what's the different when you add in all the scaffolding?
  - This is not yet a well-performing Datalog implementation
  - The problem happens to be a particularly good fit
  - etc.
- But still…

Causally-Precedes (CP)

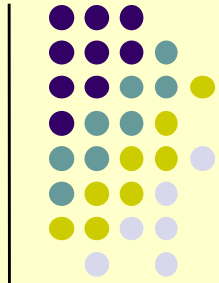


- CP: another 3-part relation
  - a release of a lock causally-precedes a subsequent acquisition *(of same lock)* if the two critical sections contain conflicting events
  - a release of a lock causally-precedes a subsequent acquisition if the two critical sections contain CP-ordered events
    - (necessary for soundness in >2 threads)
  - CP is closed under left- and right-composition with HB ($CP \circ HB = HB \circ CP = CP$)

Experience With CP



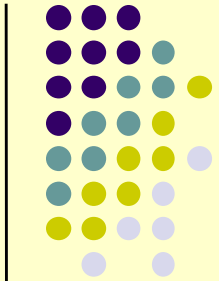
- For 2-3 weeks before the POPL'12 deadline, 3 Ph.D. students were trying to implement CP efficiently (i.e., well-enough to get numbers)
  - since 2012, two of them highly successful at Google
- They failed
- I did the CP implementation we used for the paper's benchmarks in 1 day in Datalog
  - efficient enough and good enough for our purposes by design: "Sound Predictive Race Detection ***in Polynomial Time***"



Anecdote II

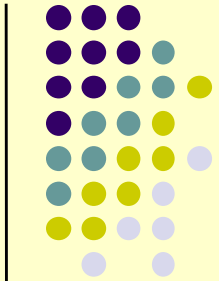
Experience with Doop

Doop, 7 years ago



- Martin Bravenboer created the first version of the framework in under 12 months (Feb-Dec'08)
- About ¼ its current size (~6KLoC), single dev
  - until the summer of 2009, I (or anyone other than Martin) had not written a single line of Doop
- Already comparable to existing points-to analysis frameworks for Java bytecode
  - e.g., IBM Wala, by a team of several (back then)

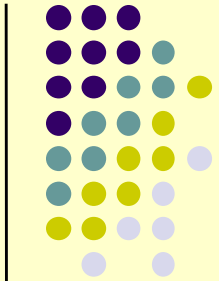
Doop, 7 years ago



- Doop could (at birth) fully replicate the results of the Paddle framework
  - Ph.D. work of a very competent programmer (Ondrej Lhotak)
- Yet was ~10x faster!
- How much of a productivity boost is that?
- What were the drivers? We'll get back to that

Revisiting the 3 Questions

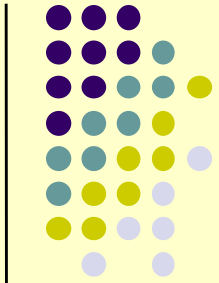
The Three Questions



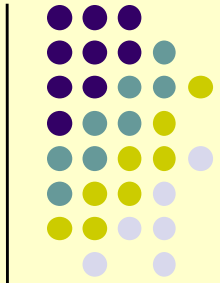
- **Question 1**: can we get large productivity increases?
- **Question 2**: are there designs that offer large productivity gains **now**?
- **Question 3**: are there useful languages that have no loop constructs, no arrays, and no functions?

- I think you know my answers

What Can We Learn From This?

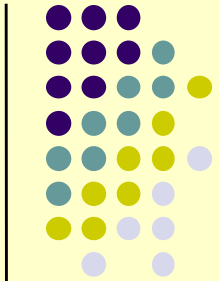


- Declarative languages are probably just one part of the productivity answer
- Can we take a step back?
- Speculative, subjective “lessons”



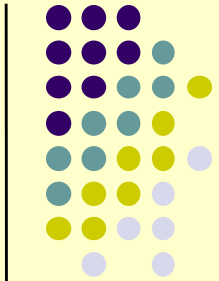
Lesson: Productivity and Performance Tied Together

Lesson: Productivity and Performance Tied Together



- *If a language can give orders-of-magnitude improvements in productivity*
  THEN
  its implementation has the potential for orders-of-magnitude improvements in performance
  - both are aspects of being abstract
  - how is it possible to get productivity improvements if one needs to specify data and algorithms concretely, with “loops and arrays”?

Lesson: Productivity and Performance Tied Together

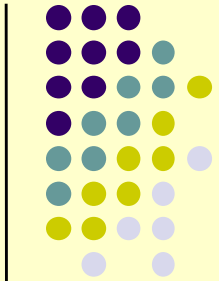


- Abstract languages can change the *asymptotic complexity* of a program
- E.g., in Datalog:

```
A(x,y) <- A(y,z), B(z,x,w), C(w,y).
C(x,y) <- A(y,w), D(w,x).
```

  - order of joins
  - indexing
  - incrementalization

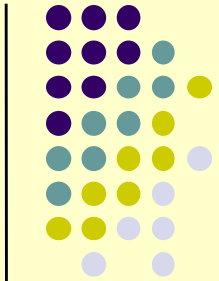
Lesson: Productivity and Performance Tied Together



- Order of joins: `A<-A,B,C` possibly catastrophic
- Is `A<-A,C,B` better? Probably
- What if no `C` index on `y`?

```
A(x,y) <- A(y,z), B(z,x,w), C(w,y).
C(x,y) <- A(y,w), D(w,x).
```

Lesson: Productivity and Performance Tied Together



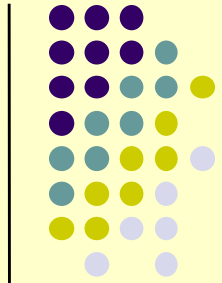
- Joining tables is one kind of looping, recursion is the other

```
A(x,y) <- A(y,z), B(z,x,w), C(w,y).
C(x,y) <- A(y,w), D(w,x).
```

  - implemented as:

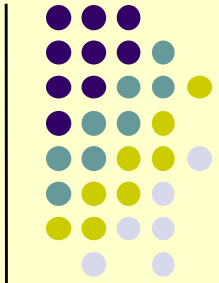
```
ΔA(x,y) <- ΔA(y,z), B(z,x,w), C(w,y).
ΔA(x,y) <- A(y,z), B(z,x,w), ΔC(w,y).
ΔC(x,y) <- ΔA(y,w), D(w,x).
```

- Would you do this by hand? Main source of inefficiencies in Paddle, handwritten analysis



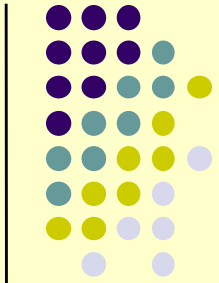
Lesson: Need For Firm Mental Ground

Lesson: Need For Firm Mental Ground

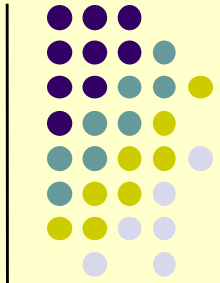


- *If a language can give orders-of-magnitude improvements in productivity*
  THEN
  it will make it too easy to break things. The language design should naturally keep sanity

Lesson: Need for Firm Mental Ground

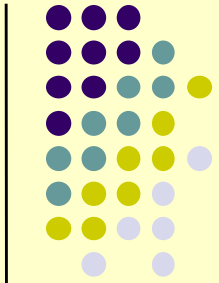


- In Datalog development, the #1 sanity-keeping feature is *monotonicity*
- Extra rules can only produce *more* results
- Everything that used to hold, still does
  - though not entirely true, close enough
- Also, *termination*: programs will converge
  - though not entirely true, close enough



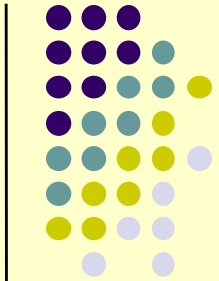
Lesson: Development Patterns Change

Lesson: Development Patterns Change

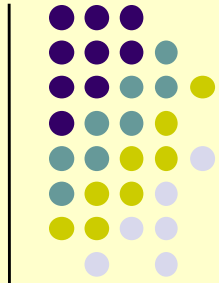


- *If a language can give orders-of-magnitude improvements in productivity*
  THEN
  a programmer's workflow will change fairly radically

Lesson: Development Patterns Change

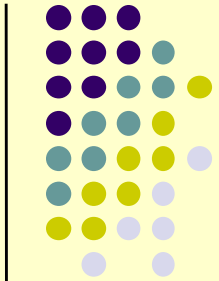


- My Datalog experience
  - much easier to pick up code after a while
  - much easier to develop incrementally
  - debugging not trivial
    - goes with performance improvement: lots of intermediate results missed
  - more time running than writing code



Conclusions: Revisiting the 3 Questions

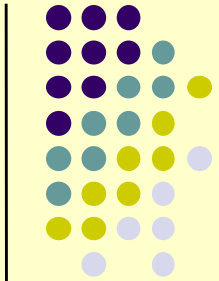
The Three Questions



- **Question 1**: can we get large productivity increases?
- **Question 2**: are there designs that offer large productivity gains **now**?
- **Question 3**: are there useful languages that have no loop constructs, no arrays, and no functions?

- I will claim “yes” on all three

More Importantly



- We expect this story (productivity, different design) from domain-specific languages
- What's the common domain of
  - race detection
  - points-to analysis
  - retail prediction applications?