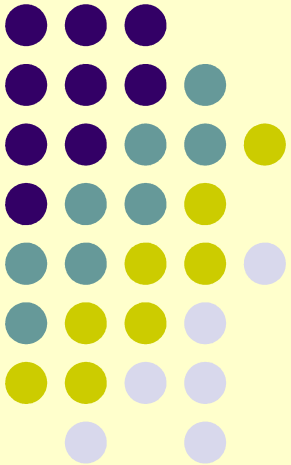
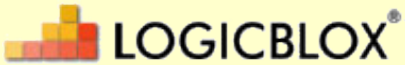
# Languages for Large Productivity Gains: What Will they Look Like?

## (Was: My Foray Into Declarative Languages)

Yannis Smaragdakis
**University of Athens**

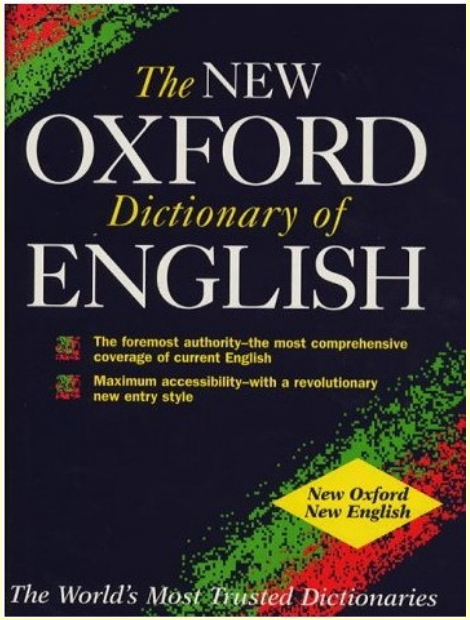



European Research Council
Established by the European Commission

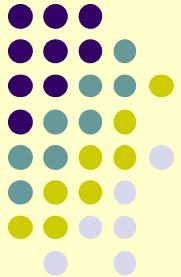
# “Declarative”?

*“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

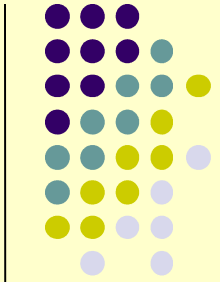
# What Am I Doing in This Space?



- 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.

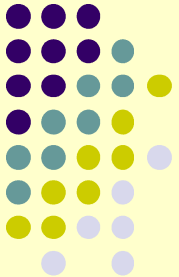
# Since Then…

- 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

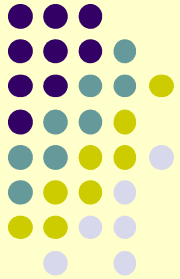
# LogicBlox



- 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, …]**

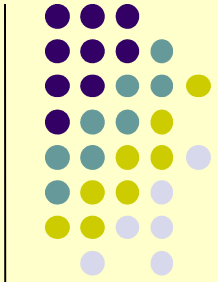


- Datalog-based analysis frameworks for Java, C, C++

DOOP

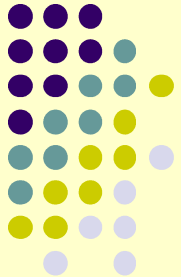
- 2-3K logical rules (20-25KLoC)
- Very high performance (often 10x over prior work)
- Sophisticated, very rich set of analyses
  - subset-based analysis, fully on-the-fly call graph discovery, field-sensitivity, context-sensitivity, call-site sensitive, object sensitive, thread sensitive, context-sensitive 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**



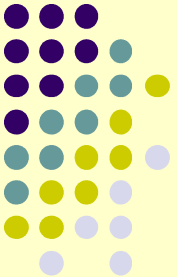
# Back To Our Group (Language Design) …

# Quotes From "Blue. No! Yellow!"



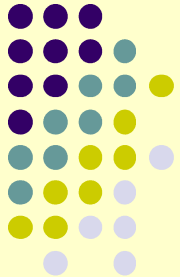
- *"[W]e've passed the point of diminishing returns. No future language will give us the factor of 10 advantage that assembler gave us over binary. No future language will give us 50%, or 20%, or even 10% reduction in workload"*
  - **Question 1**: can we get large productivity increases?
  - Also "assembler 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?

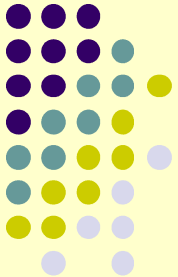
# My Anecdotes



- Anecdote 1: developed implementation of CP relation (for POPL'12 paper) in 1 day, vs. 2-3 weeks of failed attempts in Java
- Anecdote 2: Doop captured a very rich set of pointer analysis algorithms with ~12 months of development effort
  - and 10x performance improvement!

# Revisiting the 3 Questions

# The Three Questions



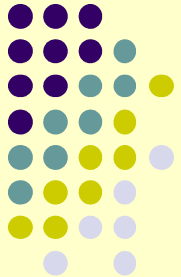
- **Question 1**: can we get large productivity increases?
- **Question 2**: are there designs that offer large productivity gains **now**?
- **Question 3**: are there useful languages that have no loop constructs, no arrays, and no functions?

- I think you know my answers

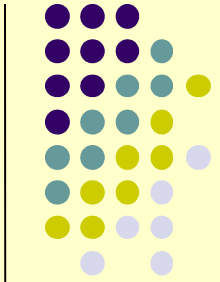
# More Importantly

- 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?

# What Can We Learn From This?

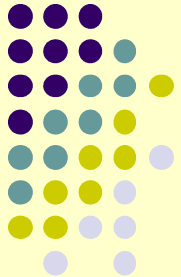


- Declarative languages are probably just one part of the productivity answer
- Can we take a step back?
- Speculative, subjective “lessons” for high-productivity languages of the future



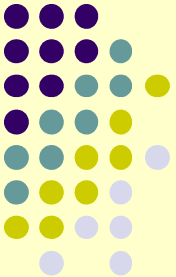
# Lesson: Productivity and Performance Tied Together

# 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

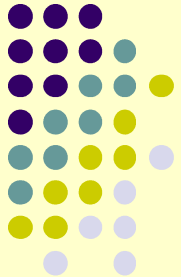


- Abstract languages can change the *asymptotic complexity* of a program
- E.g., in Datalog:

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

  - order of joins
  - indexing
  - incrementalization

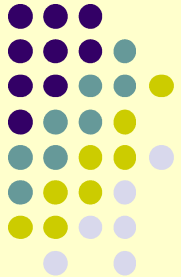
# Lesson: Productivity and Performance Tied Together



- Order of joins: `A<-A,B,C` possibly catastrophic
- `A<-A,C,B` better? `A<-C,B,A` even more
- What if no `C` index on `z`?

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

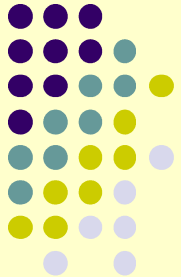
  - implemented as:

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

- Would you do this by hand? Main source of inefficiencies in past analyses

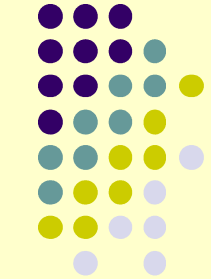
# Lesson: Need For Firm Mental Ground

# Lesson: Need For Firm Mental Ground

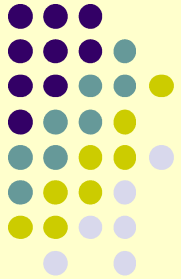


- *If a language can give orders-of-magnitude improvements in productivity*
  *THEN*
  *it will make it too easy to break things. The language design should naturally keep sanity*

# Lesson: Need for Firm Mental Ground

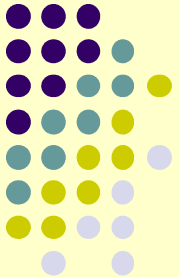


- In Datalog development, the #1 sanity-keeping feature is *monotonicity*
- Extra rules can only produce *more* results
- Everything that used to hold, still does
  - though not entirely true, close enough
- Also, *termination*: programs will converge
  - though not entirely true, close enough



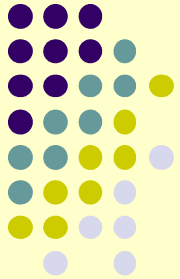
# Lesson: Development Patterns Change

# Lesson: Development Patterns Change

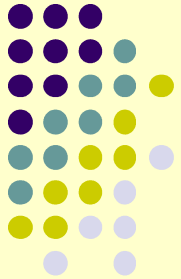


- *If a language can give orders-of-magnitude improvements in productivity*
  *THEN*
  *a programmer's workflow will change fairly radically*

# Lesson: Development Patterns Change

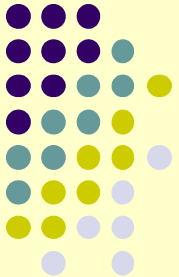


- My Datalog experience
  - much easier to pick up code after a while
  - much easier to develop incrementally
  - debugging not trivial
    - goes with performance improvement: lots of intermediate results missed
  - more time running than writing code



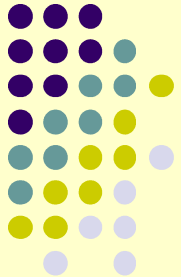
# Lesson: Need for Formal Proof

# Lesson: Different Balance of Formal Reasoning and Coding



- I speculate that with high-productivity languages:
  - formal proofs will be easier
  - formal proofs will be *less* necessary!
- Both are an outcome of “code”

# Conclusion: Starting From 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
- Positive instances give us glimpses of future high-productivity languages
    - let’s try to generalize!