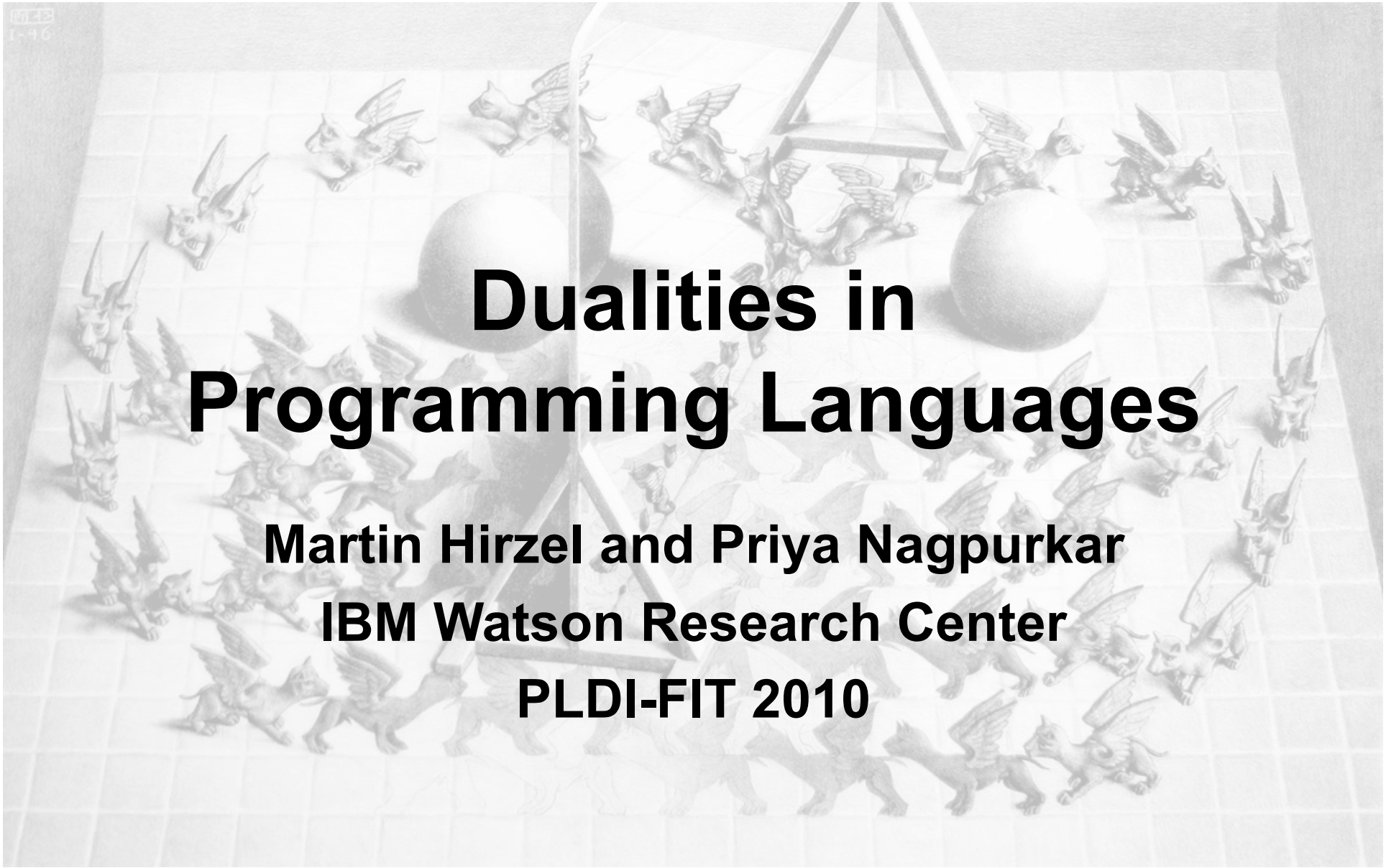
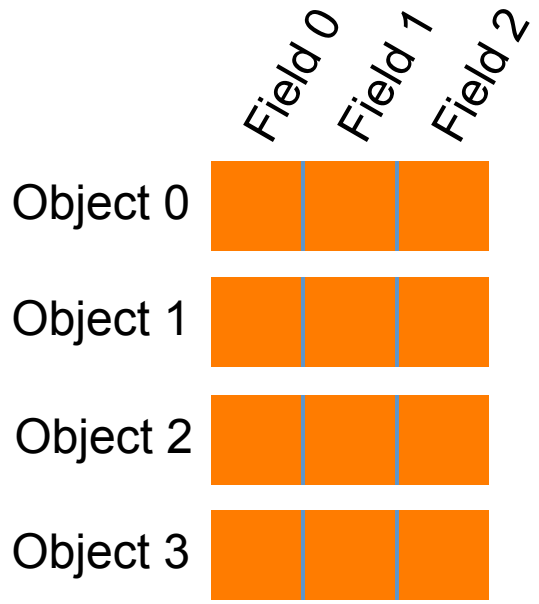


Dualities in Programming Languages

Martin Hirzel and Priya Nagpurkar
IBM Watson Research Center
PLDI-FIT 2010

Row-Based Layout

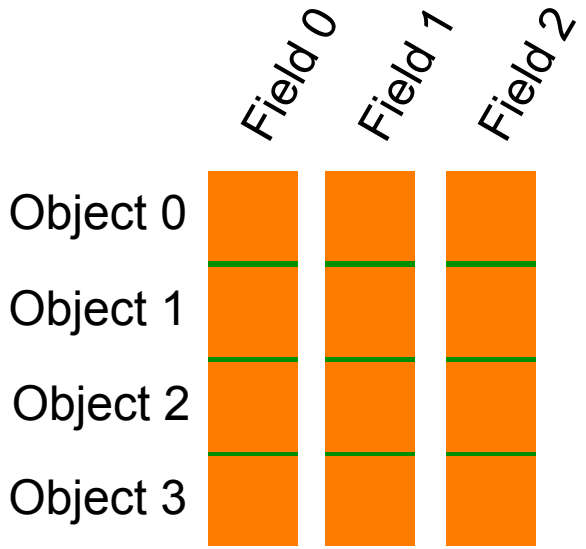


Values of different fields but same object contiguous.

Good locality if few hot objects.

Field access dereferences objectAddress + fieldOffset.

Column-Based Layout

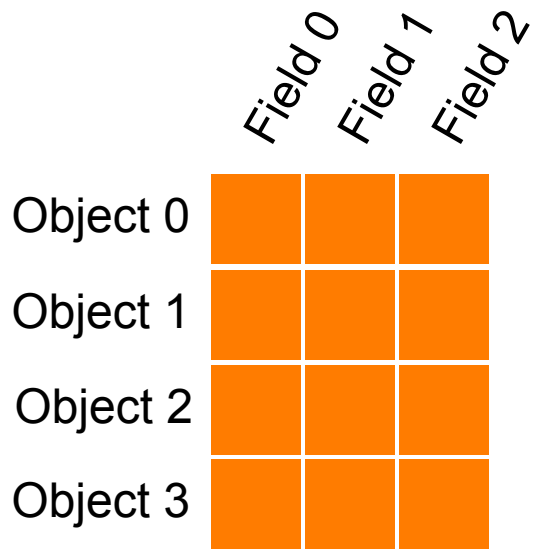


Values of different objects but same field contiguous.

Good locality if few hot fields.

Field access dereferences fieldAddress + objectOffset.

Row-Based ≅ Column-Based Layout



Values of different fields / objects but same object / field contiguous.

Good locality if few hot objects. / fields.

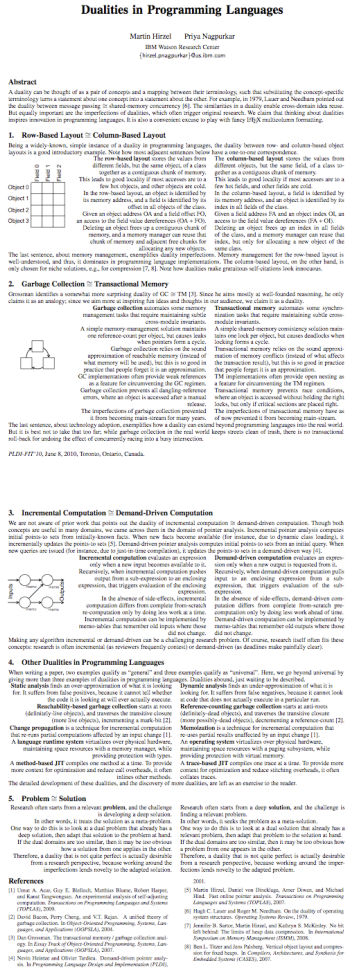
Field access dereferences objectAddress / fieldAddress + fieldOffset. / objectOffset.

Observations

- Duality =
  pair of concepts
  + terminology substitution.
- Not always perfect
  (e.g., memory management
  easier for row-based layout.)

Observations

- Duality =
  pair of concepts
  + terminology substitution.
- Not always perfect
  (e.g., memory management
  easier for row-based layout.)
- Good excuse for fancy formatting.



Dualities in Programming Languages

Martin Hirzel Priya Nagpurkar
IBM Watson Research Center
{hirzel,pnagpurkar}@us.ibm.com

Abstract

A duality can be thought of as a pair of concepts and a mapping between their terminology, such that substituting the concepts-specific terminology turns a statement about one concepts into a statement about the other. For example, in 1979, Lazar and Needham pointed out the duality between message passing & shared-memory concurrency [8]. The similarities in a duality enable cross-domain idea reuse. But equally important are the imperfections of dualities, which often trigger original research. We claim that thinking about dualities inspires innovation in programming languages. It is also a convenient excuse to play with fancy LaTeX multicolumn formatting.

1. Row-Based Layout ☆ Column-Based Layout

Being a widely-known, simple instance of a duality in programming languages, the duality between row- and column-based object layouts is a good introductory example. Next how most adjacent sentences below have a one-to-one correspondence.

The **row-based layout** stores the values from different fields, but the same object, of a class together as a contiguous chunk of memory.

This leads to good locality if most accesses are to a few hot objects, and other objects are cold.

In the row-based layout, an object is identified by its memory address, and a field is identified by its offset in all objects of the class.

Given an object address OA and a field offset FO, an access to the field value dereferences (OA + FO).

Deleting an object frees up a contiguous chunk of memory, and a memory manager can reuse that chunk of memory only for allocating a new object of the same class.

The last sentence, about memory management, exemplifies duality imperfections. Memory management for the row-based layout is well-understood, and thus, it dominates in programming language implementations. The column-based layout, on the other hand, is only chosen for niche solutions, e.g., for compression [7, 8]. Note how dualities make gratuitous self-citations look innocuous.

The **column-based layout** stores the values from different objects, but the same field, of a class together as a contiguous chunk of memory.

This leads to good locality if most accesses are to a few hot fields, and other fields are cold.

In the column-based layout, a field is identified by its memory address, and an object is identified by its index in all fields of the class.

Given a field address FA and an object index OI, an access to the field value dereferences (FA + OI).

Deleting an object frees up an index in all fields of the class, and a memory manager can reuse that index only for allocating a new object of the same class.

2. Garbage Collection ☆ Transactional Memory

Grossman identifies a somewhat more surprising duality of GC ☆ TM [3]. Since he aims mostly at well-founded reasoning, he only claims it as an analogy, since we aim more at inspiring fun ideas and thoughts in our audience, we claim it as a duality.

Garbage collection automates some memory management tasks that require maintaining subtle cross-module invariants.

A simple memory-management solution maintains one reference-count per object, but causes leaks when pointers form a cycle.

Garbage collection relies on the sound approximation of reachable memory (instead of what memory will be used), but this is so good in practice that people forget it is an approximation.

GC implementations often provide weak references as a feature for circumventing the GC regimen.

Garbage collection prevents all dangling-reference errors, where an object is accessed after a manual release.

The imperfections of garbage collection prevented it from becoming main-stream for many years.

Transactional memory automates some synchronization tasks that require maintaining subtle cross-module invariants.

A simple shared-memory consistency solution maintains one lock per object, but causes deadlocks when locking forms a cycle.

Transactional memory relies on the sound approximation of memory conflicts (instead of what affects the transaction result), but this is so good in practice that people forget it is an approximation.

TM implementations often provide open nesting as a feature for circumventing the TM regimen.

Transactional memory prevents race conditions, where an object is accessed without holding the right locks, but only if critical sections are placed right.

The imperfections of transactional memory have as of now prevented it from becoming main-stream.

The last sentence, about technology adoption, exemplifies how a duality can extend beyond programming languages into the real world. But it is best not to take that too far; while garbage collection in the real world keeps sirens clear of trash, there is no transactional roll-back for undoing the effect of concurrently racing into a bury intersection.

PLDI-FIT'10, June 8, 2010, Toronto, Ontario, Canada.

3. Incremental Computation ☆ Demand-Driven Computation

We are not aware of prior work that points out the duality of incremental computation & demand-driven computation. Though both concepts are useful in many domains, we came across them in the domain of pointer analysis. Incremental pointer analysis computes initial points-to sets from initially-known facts. When new facts become available (for instance, due to dynamic class loading), it incrementally updates the points-to sets [5]. Demand-driven pointer analysis computes initial points-to sets from an initial query. When new queries are issued (for instance, due to just-in-time compilation), it updates the points-to sets in a demand-driven way [6].

Incremental computation evaluates an expression only when a new input becomes available to it.

Recursively, when incremental computation pushes outputs from a sub-expression to an enclosing expression, that triggers evaluation of the enclosing expression.

In the absence of side-effects, incremental computation differs from complete from-scratch re-computation only by doing less work at a time.

Incremental computation can be implemented by memo-tables that remember old inputs where those did not change.

Demand-driven computation evaluates an expression only when a new output is requested from it.

Recursively, when demand-driven computation pulls input to an enclosing expression from a sub-expression, that triggers evaluation of the sub-expression.

In the absence of side-effects, demand-driven computation differs from complete from-scratch pre-computation only by doing less work ahead of time.

Demand-driven computation can be implemented by memo-tables that remember old outputs where those did not change.

Making any algorithm incremental or demand-driven can be a challenging research problem. Of course, research itself often fits these concepts: research is often incremental (as reviewers frequently contest) or demand-driven (as deadlines make painfully clear).

4. Other Dualities in Programming Languages

When writing a paper, two examples qualify as "general" and three examples qualify as "universal". Here, we go beyond universal by giving more than three examples of dualities in programming languages. Dualities abound, just waiting to be described.

Static analysis finds an over-approximation of what it is looking for. It suffers from false positives, because it cannot tell whether the code it is looking at will ever actually execute.

Reachability-based garbage collection starts at roots (definitely-live objects), and traverses the transitive closure (more live objects), incrementing a mark-bit [2].

Change propagation is a technique for incremental computation that re-runs partial computations affected by an input change [1].

A **language runtime system** virtualizes over physical hardware, maintaining space resources with a memory manager, while providing protection with types.

A **method-based JIT** compiles one method at a time. To provide more context for optimization and reduce call overheads, it often inlines other methods.

Dynamic analysis finds an under-approximation of what it is looking for. It suffers from false negatives, because it cannot look at code that does not actually execute in a particular run.

Reference-counting garbage collection starts at anti-roots (definitely-dead objects), and traverses the transitive closure (more possibly-dead objects), decrementing a reference-count [2].

Memoization is a technique for incremental computation that re-uses partial results unaffected by an input change [1].

An **operating system** virtualizes over physical hardware, maintaining space resources with a paging subsystem, while providing protection with virtual memory.

A **trace-based JIT** compiles one trace at a time. To provide more context for optimization and reduce stitching overheads, it often collates traces.

The detailed development of these dualities, and the discovery of more dualities, are left as an exercise to the reader.

5. Problem ☆ Solution

Research often starts from a relevant **problem**, and the challenge is developing a deep solution.

In other words, it treats the solution as a meta-problem.

One way to do this is to look at a dual problem that already has a deep solution, then adapt that solution to the problem at hand.

If the dual domains are too similar, then it may be too obvious how a solution from one applies in the other.

Therefore, a duality that is not quite perfect is actually desirable from a research perspective, because working around the imperfections lends novelty to the adapted solution.

Research often starts from a deep **solution**, and the challenge is finding a relevant problem.

In other words, it seeks the problem as a meta-solution.

One way to do this is to look at a dual solution that already has a relevant problem, then adapt that problem to the solution at hand.

If the dual domains are too similar, then it may be too obvious how a problem from one appears in the other.

Therefore, a duality that is not quite perfect is actually desirable from a research perspective, because working around the imperfections lends novelty to the adapted problem.

References

[1] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 2009.

[2] David Bacon, Perry Cheng, and V.T. Rajan. A unified theory of garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[3] Dan Grossman. The transactional memory / garbage collection analogy. In *Essay Track of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[4] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Programming Language Design and Implementation (PLDI)*, 2001.

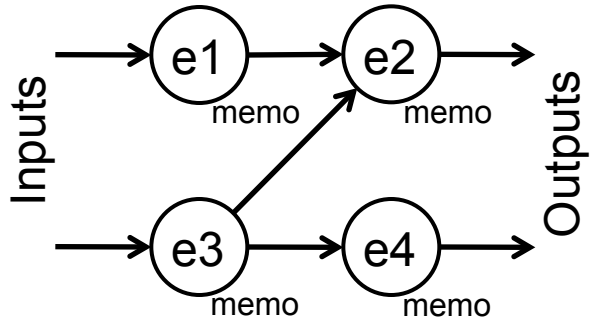
[5] Martin Hirzel, Daniel von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *Transactions on Programming Languages and Systems (TOPLAS)*, 2007.

[6] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *Operating Systems Review*, 1979.

[7] Jennifer B. Sartor, Martin Hirzel, and Kathryn S. McKinley. No bit left behind: The limits of heap data compression. In *International Symposium on Memory Management (ISMM)*, 2008.

[8] Ben L. Titzer and Jens Palsberg. Vertical object layout and compression for fixed heaps. In *Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, 2007.

Incremental ≅ Demand-driven Computation



Evaluate expression only when input becomes available. / output is requested.

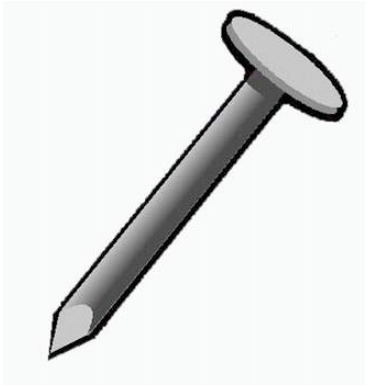
Trigger next / previous expression when pushing output / pulling input from previous / next expression.

Do less work per input / per output than from-scratch computation.

Other Dualities in Programming Languages

Garbage collection (GC)	≅	Transactional memory (TM)
Static analysis	≅	Dynamic analysis
Reachability-based GC	≅	Reference-counting GC
Change propagation	≅	Memoization
Language runtime system	≅	Operating system
Method-based JIT	≅	Trace-based JIT
Subject/Observer	≅	Iterator

And finally, the most practical duality,
which you can use right away …



Problem ≅ Solution

Research often starts from problem / solution and seeks a solution. / problem.

In other words, the solution / problem becomes a meta- problem. / solution.

Can do this by looking at dual problem / solution and adopting its solution. / problem.

Happy dualities hunting!