Two Notions of Beauty in Programming

Robert Harper

Computer Science Department
Carnegie Mellon University

Programming Languages Mentoring Workshop 2016

Thanks

Thanks to the PLMW organizers for the invitation!

Ongoing work with Guy E. Blelloch at Carnegie Mellon.

Two Sources of Beauty in Programs

For me beauty in a program arises from two sources:

- Structure: code as an expression of an idea.
- Efficiency: code as instructions for a computer.

Two Sources of Beauty in Programs

For me beauty in a program arises from two sources:

- Structure: code as an expression of an idea.
- Efficiency: code as instructions for a computer.

This has given rise to two theories of computation.

- Logical: compositionality (human effort).
- Combinatorial: efficiency (machine effort).

Two Sources of Beauty in Programs

For me beauty in a program arises from two sources:

- Structure: code as an expression of an idea.
- Efficiency: code as instructions for a computer.

This has given rise to two theories of computation.

- Logical: compositionality (human effort).
- Combinatorial: efficiency (machine effort).

Oddly, these are largely disparate communities!

Reconciling the Two Theories

Historically,

- The logical side neglects efficiency in favor of structure.
- The combinatorial side neglects structure in favor of efficiency.

Reconciling the Two Theories

Historically,

- The logical side neglects efficiency in favor of structure.
- The combinatorial side neglects structure in favor of efficiency.

Prospectively,

- The logical side should pay more attention to efficiency.
- The combinatorial side should pay more attention to structure.

Reconciling the Two Theories

Historically,

- The logical side neglects efficiency in favor of structure.
- The combinatorial side neglects structure in favor of efficiency.

Prospectively,

- The logical side should pay more attention to efficiency.
- The combinatorial side should pay more attention to structure.

The λ-calculus is the key!

The Great Rift

"On the fact that the Atlantic Ocean has two sides." [EWD]

- American theory $\approx$ combinatorial theory.
- Euro-theory $\approx$ semantics and logic.

The Great Rift

"On the fact that the Atlantic Ocean has two sides." [EWD]

- American theory $\approx$ combinatorial theory.
- Euro-theory $\approx$ semantics and logic.

Both have had a big influence on practice:

- Efficient algorithms for a broad range of problems.
- Language design and verification tools.

The Great Rift

"On the fact that the Atlantic Ocean has two sides." [EWD]

- American theory $\approx$ combinatorial theory.
- Euro-theory $\approx$ semantics and logic.

Both have had a big influence on practice:

- Efficient algorithms for a broad range of problems.
- Language design and verification tools.

Yet these two "theories" operate largely in isolation!

American Theory

Algorithm analysis is based on machine models:

- Turing machine (TM) or Random Access Machine (RAM).
- Low-level: no abstraction, no composition.
- Allegedly, close to the hardware.

Machine models provide natural complexity measures:

- Time = number of instructions.
- Space = tape or memory usage.

Asymptotics smoothes over differences among models.

American Theory

In practice algorithms are described using C-like notation.

- Clearer than TM or RAM code.
- Analyze compiled code, rather than source code.

An improvement, but still very limited:

- ephemeral data structures.
- manual memory management.
- poor composability.
- no abstraction.

Euro Theory

Euro theory is based on language models:

- Church's (typed and untyped) λ-calculus.
- High-level: abstraction, composition are fundamental.
- Platform-independent.

Language models support composition via variables:

- If ϕ true $\vdash \psi$ true, then if ϕ true, then ψ true.
- If $x : \sigma \vdash N : \tau$, then if $M : \sigma$, then $[M/x]N : \tau$.

The λ-calculus is an elegant theory of composition.

Euro Theory

Languages based on λ-calculus stress

- persistent data structures.
- automatic memory management.
- strong composability.
- abstract types.

But there is relatively little emphasis on efficiency.

- No clear complexity measures.
- Few analytic results (but see Okasaki's CMU Ph.D.).

Thesis

Traditional imperative methods of programming are obsolete.

- Tedious to program, a nightmare to maintain.
- Largely incompatible with parallelism.

Functional methods are destined to dominate.

- Support verification and composition.
- Naturally accommodate parallelism.

The way forward is to synthesize Euro- and American theory.

Cost Semantics

To elevate the level of discourse we require a cost semantics.

- Define the abstract cost of execution of a language.
- Defines the parallel and sequential complexity.

Algorithm analysis is conducted at the level of the code we write.

- Cost semantics assigns a measure to each execution.
- Analyze asymptotic complexity in terms of this measure.

Cost Semantics

The abstract cost is validated by a bounded implementation.

- Transform abstract cost into concrete cost on a machine.
- Account for platform characteristics such as number of processors, cache hierarchy, and interconnect.

An end-to-end asymptotics with a clear separation of concerns.

- High-level, composable development and reasoning.
- Low-level implementation on hardware platforms.

Cost Semantics

The abstract cost is validated by a bounded implementation.

- Transform abstract cost into concrete cost on a machine.
- Account for platform characteristics such as number of processors, cache hierarchy, and interconnect.

An end-to-end asymptotics with a clear separation of concerns.

- High-level, composable development and reasoning.
- Low-level implementation on hardware platforms.

So simple we teach it to first-year undergraduates!

Cost Semantics for Time

Associate a cost graph to the evaluation of a program.

- Dynamic, fully accurate record of data dependencies.
- Not a static analysis or an approximation!

Example: function application.

$$\frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1(e_2) \Downarrow v}$$

Cost Semantics for Time

Associate a cost graph to the evaluation of a program.

- Dynamic, fully accurate record of data dependencies.
- Not a static analysis or an approximation!

Example: function application.

$$\frac{e_1 \Downarrow^{g_1} \lambda x.e \quad e_2 \Downarrow^{g_2} v_2 \quad [v_2/x]e \Downarrow^{g} v}{e_1(e_2) \Downarrow \qquad\qquad v}$$

Cost Semantics for Time

Associate a cost graph to the evaluation of a program.

- Dynamic, fully accurate record of data dependencies.
- Not a static analysis or an approximation!

Example: function application.

$$\frac{e_1 \Downarrow^{g_1} \lambda x.e \quad e_2 \Downarrow^{g_2} v_2 \quad [v_2/x]e \Downarrow^{g} v}{e_1(e_2) \Downarrow^{(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g} v}$$

Cost Graphs

Series-parallel cost graphs:

- $\mathbf{1}$: one unit of computation.

Application cost $(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g$ specifies that

Cost Graphs

Series-parallel cost graphs:

- $\mathbf{1}$: one unit of computation.
- $g_1 \oplus g_2$: g_2 depends on result of g_1.

Application cost $(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g$ specifies that

Cost Graphs

Series-parallel cost graphs:

- $\mathbf{1}$: one unit of computation.
- $g_1 \oplus g_2$: g_2 depends on result of g_1.
- $g_1 \otimes g_2$: g_1 and g_2 are independent.

Application cost $(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g$ specifies that

Cost Graphs

Series-parallel cost graphs:

- $\mathbf{1}$: one unit of computation.
- $g_1 \oplus g_2$: g_2 depends on result of g_1.
- $g_1 \otimes g_2$: g_1 and g_2 are independent.

Application cost $(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g$ specifies that

- Function and argument are evaluated in parallel.

Cost Graphs

Series-parallel cost graphs:

- $\mathbf{1}$: one unit of computation.
- $g_1 \oplus g_2$: g_2 depends on result of g_1.
- $g_1 \otimes g_2$: g_1 and g_2 are independent.

Application cost $(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g$ specifies that

- Function and argument are evaluated in parallel.
- Function call costs one unit.

Cost Graphs

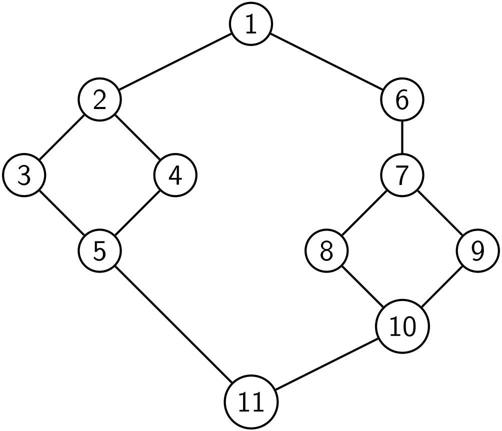
Series-parallel cost graphs:

- $\mathbf{1}$: one unit of computation.
- $g_1 \oplus g_2$: g_2 depends on result of g_1.
- $g_1 \otimes g_2$: g_1 and g_2 are independent.

Application cost $(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g$ specifies that

- Function and argument are evaluated in parallel.
- Function call costs one unit.
- Function execution depends on the function and argument.

Cost Graphs



Work and Span

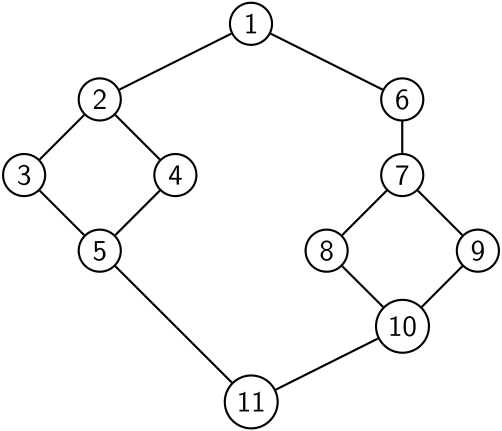
The work $w(g)$ of a cost graph g is the size of g.

- $w(\mathbf{1}) = 1$, $w(g_1 \otimes g_2) = w(g_1 \oplus g_2) = w(g_1) + w(g_2)$.
- Measures the sequential time complexity.

The span $d(g)$ of a cost graph g is the critical path length of g.

- $d(\mathbf{1}) = 1$, $d(g_1 \otimes g_2) = \max(d(g_1), d(g_2))$, $d(g_1 \oplus g_2) = d(g_1) + d(g_2)$.
- Measures the parallel time complexity.

Cost Graphs



Work = 11, Span = 6

Mergesort

```
fun merge xs ys =
  case (xs, ys) of
    ([], ys) ⇒ ys
  | (xs,[]) ⇒ xs
  | (x::xs', y::ys') ⇒
    case x<y of
      true ⇒ x :: merge xs' ys
    | false ⇒ y :: merge xs ys'

fun sort [] = []
  | sort [x] = [x]
  | sort xs =
    let val (ys, zs) = split xs
    in  merge (sort ys, sort zs)  end
```

Mergesort

The work (sequential time) is optimal, $O(n \log n)$ for n items.

The span (parallel time) is sensitive to the data structure:

- For lists, $O(n)$, because splitting is slow.
- For trees, $O(\log^3 n)$, using rebalancing.

The parallelizability ratio, w/d, is $O(n/\log^2 n)$ for trees.

The correctness of the parallel implementation is never in question!

Bounded Implementation for Time

Brent's Principle: A computation with work w and span d can be implemented on a p-processor PRAM in time $O(\max(w/p, d))$.

- Work in chunks of p as much as possible.
- Number of processors is chosen at run-time.
- Proof is constructive: exhibits a scheduler.

Relates abstract cost to its concrete realization. No need for pseudo-code!

IO Efficiency

Aggarwal and Vitter introduced the IO Model:

- Distinguish primary from secondary memory.
- Cache size $M = k \times B$ words.
- Evaluate algorithm efficiency in terms of M and B.

Main result: k-way merge sort is optimal for the IO model:

$$O(n/B \log_{M/B}(n/B))$$

IO Efficiency

A&V's results can be matched in a purely functional model.

- No manual memory management.
- Natural functional programming.

Key idea: temporal locality implies spatial locality.

- Allocation order determines proximity.
- Reloading of migrated objects preserves proximity.
- Control stack specially managed to avoid cache contention.

Cost Semantics for IO

Cost semantics makes storage explicit:

$$\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} v$$

Store σ has three components:

Cost Semantics for IO

Cost semantics makes storage explicit:

$$\sigma \,@\, e \Downarrow^{n} \sigma' \,@\, v$$

Store σ has three components:

- Unbounded main memory with blocks of size B.

Cost Semantics for IO

Cost semantics makes storage explicit:

$$\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} v$$

Store σ has three components:

- Unbounded main memory with blocks of size B.
- Read cache of size $M = k \times B$.

Cost Semantics for IO

Cost semantics makes storage explicit:

$$\sigma \; @ \; e \Downarrow^{n} \sigma' \; @ \; v$$

Store σ has three components:

- Unbounded main memory with blocks of size B.
- Read cache of size $M = k \times B$.
- Linearly ordered allocation cache of size M.

Cost Semantics for IO

Cost semantics makes storage explicit:

$$\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} v$$

Store σ has three components:

- Unbounded main memory with blocks of size B.
- Read cache of size $M = k \times B$.
- Linearly ordered allocation cache of size M.

Figure of merit: traffic between main memory and cache expressed in terms of M and B.

(Simplified) Cost Semantics

$$\frac{\left\{ \quad \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'} \quad \sigma_1' \mathbin{@} l_1' \quad \right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1' + n_1'' + \quad n_2 + n_2'} \sigma' \mathbin{@} l'}$$

(Simplified) Cost Semantics

$$\frac{\left\{\begin{array}{ll} & \sigma_1 \,@\, e_1 \Downarrow^{n_1'} \qquad \sigma_1' \,@\, l_1' \\ \sigma_1' \,@\, l_1' \downarrow^{n_1''} \sigma_1'' \,@\, \lambda x.e & \\ & \end{array}\right\}}{\sigma \,@\, \mathtt{app}(e_1; e_2) \Downarrow^{n_1'+n_1''+ \qquad n_2+n_2'} \sigma' \,@\, l'}$$

(Simplified) Cost Semantics

$$\frac{\left\{\begin{array}{l} \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e \\ \sigma_1'' \mathbin{@} e_2 \Downarrow^{n_2} \sigma_2' \mathbin{@} l_2' \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1' + n_1'' + n_2 + n_2'} \sigma' \mathbin{@} l'}$$

(Simplified) Cost Semantics

$$\frac{\left\{\begin{array}{ll} & \sigma_1 \mathbin{@} e_1 \Downarrow^{n'_1} \quad \sigma'_1 \mathbin{@} l'_1 \\ \sigma'_1 \mathbin{@} l'_1 \downarrow^{n''_1} \sigma''_1 \mathbin{@} \lambda x.e & \\ \sigma''_1 \mathbin{@} e_2 \Downarrow^{n_2} \quad \sigma'_2 \mathbin{@} l'_2 & \sigma'_2 \mathbin{@} [l'_2/x]e \Downarrow^{n'_2} \sigma' \mathbin{@} l' \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n'_1 + n''_1 + n_2 + n'_2} \sigma' \mathbin{@} l'}$$

Bounded Implementation for IO

Thm (Blelloch & H) An evaluation of cost n may be implemented on a stack machine with cache of size $4 \times M + B$ with cache complexity $k \times n$ for some small constant k.

Bounded Implementation for IO

Thm (Blelloch & H) An evaluation of cost n may be implemented on a stack machine with cache of size $4 \times M + B$ with cache complexity $k \times n$ for some small constant k.

- Sleator, et al.: LRU eviction policy is 2-competitive with ICM.

Bounded Implementation for IO

Thm (Blelloch & H) An evaluation of cost n may be implemented on a stack machine with cache of size $4 \times M + B$ with cache complexity $k \times n$ for some small constant k.

- Sleator, et al.: LRU eviction policy is 2-competitive with ICM.
- Appel: cost of copying GC is asymptotically free.

Bounded Implementation for IO

Thm (Blelloch & H) An evaluation of cost n may be implemented on a stack machine with cache of size $4 \times M + B$ with cache complexity $k \times n$ for some small constant k.

- Sleator, et al.: LRU eviction policy is 2-competitive with ICM.
- Appel: cost of copying GC is asymptotically free.
- B&H: Stack management induces small constant overhead.

Bounded Implementation for IO

Thm (Blelloch & H) An evaluation of cost n may be implemented on a stack machine with cache of size $4 \times M + B$ with cache complexity $k \times n$ for some small constant k.

- Sleator, et al.: LRU eviction policy is 2-competitive with ICM.
- Appel: cost of copying GC is asymptotically free.
- B&H: Stack management induces small constant overhead.

Thus, the cost semantics is a valid basis for IO analysis.

Merge, Revisited

```
fun merge nil ys = ys
  | merge xs nil = xs
  | merge (xs as x::xs’) (ys as y::ys’) =
    case compare x y of
      LESS ⇒ !a::merge xs’ ys
    | GTEQ ⇒ !b::merge xs ys’
```

Merge, Revisited

```
fun merge nil ys = ys
  | merge xs nil = xs
  | merge (xs as x::xs') (ys as y::ys') =
    case compare x y of
      LESS ⇒ !a::merge xs' ys
    | GTEQ ⇒ !b::merge xs ys'
```

Merge, Revisited

A data structure is compact iff it may be traversed in time $O(n/B)$.

Thm: For compact inputs `xs` and `ys` the call `merge xs ys` has cache complexity $O(n/B)$.

- Recurs down lists allocating only stack n frames: $O(n/B)$.
- Returns allocating n list cells: $O(n/B)$.

Copying operations `!a` and `!b` ensure compactness (locality).

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for “pseudo-code”.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about compilation.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about compilation.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about compilation.

Costs can be chosen to reflect different notions of complexity:

- Sequential and parallel time [B & Greiner 96].

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about compilation.

Costs can be chosen to reflect different notions of complexity:

- Sequential and parallel time [B & Greiner 96].
- Space usage of scheduling [Spoonhower, B, Gibbons, & H 09].

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about compilation.

Costs can be chosen to reflect different notions of complexity:

- Sequential and parallel time [B & Greiner 96].
- Space usage of scheduling [Spoonhower, B, Gibbons, & H 09].
- Memory hierarchy effects [B& H 13, 15].

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically sensible.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically sensible.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically sensible.

Cost semantics integrates the combinatorial aspects:

- Enrich the tools available to algorithms designers.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically sensible.

Cost semantics integrates the combinatorial aspects:

- Enrich the tools available to algorithms designers.
- Extend complexity analysis to mathematically elegant languages.

Where From Here?

Develop new (abstract and concrete) cost measures.

- Acar, B, & Muller: Memory traffic between parallel processors.

Where From Here?

Develop new (abstract and concrete) cost measures.

- Acar, B, & Muller: Memory traffic between parallel processors.
- Communication costs.

Where From Here?

Develop new (abstract and concrete) cost measures.

- Acar, B, & Muller: Memory traffic between parallel processors.
- Communication costs.

Where From Here?

Develop new (abstract and concrete) cost measures.

- Acar, B, & Muller: Memory traffic between parallel processors.
- Communication costs.

Develop methods for analysis and verification of resource usage.

- Hofmann, Hoffmann, Shao: type-based methods

Where From Here?

Develop new (abstract and concrete) cost measures.

- Acar, B, & Muller: Memory traffic between parallel processors.
- Communication costs.

Develop methods for analysis and verification of resource usage.

- Hofmann, Hoffmann, Shao: type-based methods
- Appel, others: general theorem proving methods.

Where From Here?

Develop new (abstract and concrete) cost measures.

- Acar, B, & Muller: Memory traffic between parallel processors.
- Communication costs.

Develop methods for analysis and verification of resource usage.

- Hofmann, Hoffmann, Shao: type-based methods
- Appel, others: general theorem proving methods.

Where From Here?

Develop new (abstract and concrete) cost measures.

- Acar, B, & Muller: Memory traffic between parallel processors.
- Communication costs.

Develop methods for analysis and verification of resource usage.

- Hofmann, Hoffmann, Shao: type-based methods
- Appel, others: general theorem proving methods.

Improve both the structure and efficiency of programs!