Two Notions of Beauty in Programming

Robert Harper

Computer Science Department
Carnegie Mellon University

IU CSD Distinguished Lecture Series
November 2013

Thanks

Thanks to IU CSD for the invitation!

This talk represents work with Guy E. Blelloch at Carnegie Mellon.

And with Ph.D. students John Greiner and Daniel Spoonhower.

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 ≈ combinatorial theory.
- Euro-theory ≈ 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 ≈ combinatorial theory.
- Euro-theory ≈ 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.).

A (Tendentious) 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.

An Iatrogenic Disorder

Consider the AHU Quicksort Algorithm:

- Naturally parallel: recursive calls are independent.
- Elegantly high-level: uses only a sequence abstraction.

An imperative reformulation on a PRAM mutilates the algorithm:

- Manual storage allocation and mutation.
- Manual processor allocation for scheduling.
- Concurrency control for mutation.

What should be a matter of efficiency becomes a matter of correctness!

AHU Quicksort


```
   procedure QUICKSORT(S):
1.    if S contains at most one element then return S
      else
         begin
2.          choose an element a randomly from S;
3.          let S1, S2, and S3 be the sequences of elements in S less
              than, equal to, and greater than a, respectively;
4.          return (QUICKSORT(S1) followed by S2 followed by
              QUICKSORT(S3))
         end
```

Fig. 3.7. Quicksort program.

constructed at line 3, and therefore maximize the average time spent in the recursive calls at line 4. Let $T(n)$ be the expected time required by QUICKSORT to sort a sequence of n elements. Clearly, $T(0) = T(1) = b$ for some constant b.

Suppose that element a chosen at line 2 is the ith smallest element of the n elements in sequence S. Then the two recursive calls of QUICKSORT at line 4 have an expected time of $T(i-1)$ and $T(n-i)$, respectively. Since i is equally likely to take on any value between 1 and n, and the balance of QUICKSORT(S) clearly requires time cn for some constant c, we have the relationship:

$$T(n) \le cn + \frac{1}{n}\sum_{i=1}^{n} [T(i-1) + T(n-i)], \qquad \text{for} \quad n \ge 2. \tag{3.3}$$

Algebraic manipulation of (3.3) yields

$$T(n) \le cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i). \tag{3.4}$$

We shall show that for $n \ge 2$, $T(n) \le kn \log_e n$, where $k = 2c + 2b$ and $b = T(0) = T(1)$. For the basis $n = 2$, $T(2) \le 2c + 2b$ follows immediately from (3.4). For the induction step, write (3.4) as

$$T(n) \le cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} ki \log_e i. \tag{3.5}$$

Since $i \log_e i$ is concave upwards, it is easy to show that

$$\sum_{i=2}^{n-1} i \log_e i \le \int_2^n x \log_e x \, dx \le \frac{n^2 \log_e n}{2} - \frac{n^2}{4}. \tag{3.6}$$

Substituting (3.6) in (3.5) yields

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 provable 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 provable 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 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 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 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

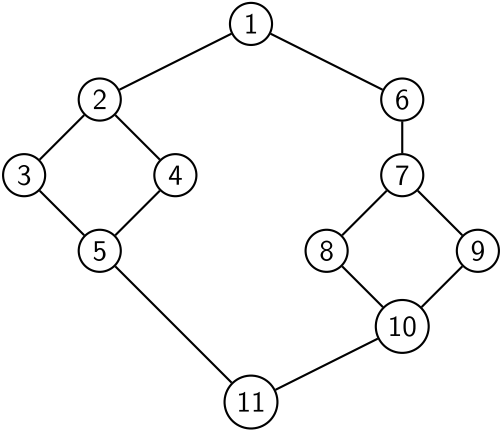
Series-parallel cost graphs:

- $\mathbf{1}$: 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



Cost Semantics

Operations on sequences have similar cost semantics:

$$\frac{e \Downarrow \lambda x.e \qquad e' \Downarrow [v_1, \ldots, v_n] \qquad [v_1/x]e \Downarrow v_1' \quad \ldots \quad [v_n/x]e \Downarrow v_n'}{\mathsf{map}(e; e') \Downarrow [v_1', \ldots, v_n']}$$

To map a function over a sequence,

Cost Semantics

Operations on sequences have similar cost semantics:

$$\frac{e \Downarrow^{g} \lambda x.e \qquad e' \Downarrow^{g'} [v_1, \ldots, v_n] \qquad [v_1/x]e \Downarrow \ v_1' \quad \ldots \quad [v_n/x]e \Downarrow \ v_n'}{\mathsf{map}(e; e') \Downarrow \qquad [v_1', \ldots, v_n']}$$

To map a function over a sequence,

- Evaluate the function and the sequence in parallel, and then

Cost Semantics

Operations on sequences have similar cost semantics:

$$\frac{e \Downarrow^{g} \lambda x.e \qquad e' \Downarrow^{g'} [v_1, \ldots, v_n] \qquad [v_1/x]e \Downarrow^{g_1} v_1' \quad \ldots \quad [v_n/x]e \Downarrow^{g_n} v_n'}{\mathsf{map}(e; e') \Downarrow \qquad [v_1', \ldots, v_n']}$$

To map a function over a sequence,

- Evaluate the function and the sequence in parallel, and then
- Apply the function to each element in parallel.

Cost Semantics

Operations on sequences have similar cost semantics:

$$\frac{e \Downarrow^{g} \lambda x.e \qquad e' \Downarrow^{g'} [v_1, \ldots, v_n] \qquad [v_1/x]e \Downarrow^{g_1} v_1' \qquad \ldots \qquad [v_n/x]e \Downarrow^{g_n} v_n'}{\mathsf{map}(e; e') \Downarrow^{(g \otimes g') \oplus (\bigotimes_i g_i) \oplus \mathbf{1}} [v_1', \ldots, v_n']}$$

To map a function over a sequence,

- Evaluate the function and the sequence in parallel, and then
- Apply the function to each element in parallel.
- Create a new sequence of results.

Work and Span

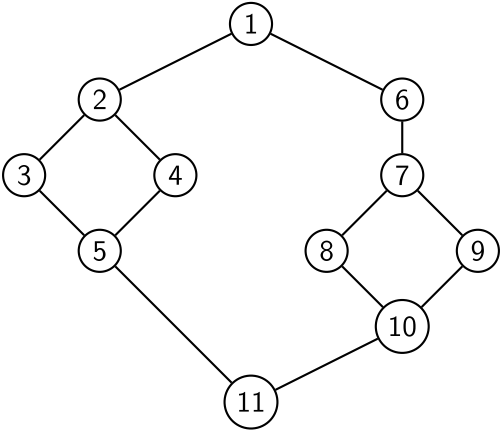
The work $w(g)$ of a cost graph g is the size of g.

- $w(\mathbf{1}) = 1$, $w(g_1 \otimes g_2) = w(g_1 \oplus g_2) = w(g_1) + w(g_2)$.
- 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!

Provable Implementation

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.

Parallelizability ratio determines which factor dominates.

2-DFS Schedule

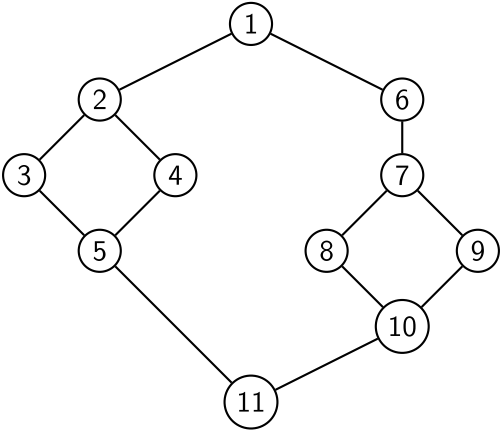
A schedule is a pebbling of the cost graph.

- Given $p > 0$ pebbles.
- Goal: move a pebble from the start to the end node.
- Move: when all predecessors are pebbled, then pick them up and pebble the successor.

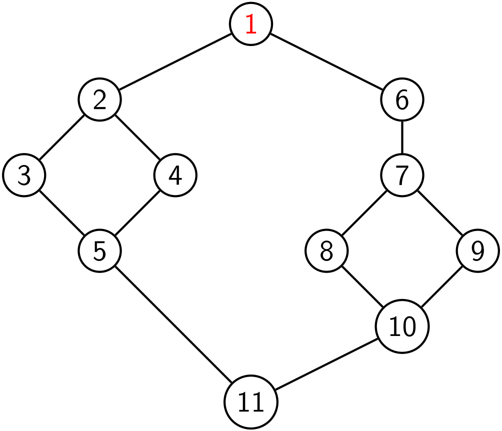
A pebbling strategy is an algorithm for pebbling a cost graph.

- p-DFS: depth-first search, p visits at a time.
- p-BFS: breadth-first search, p visits at a time.
- p-WS: work-stealing schedule.

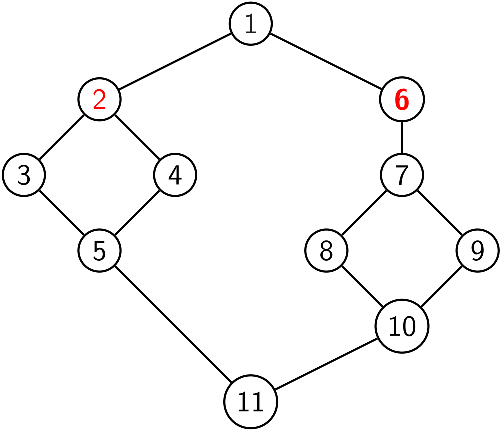
2-DFS Schedule



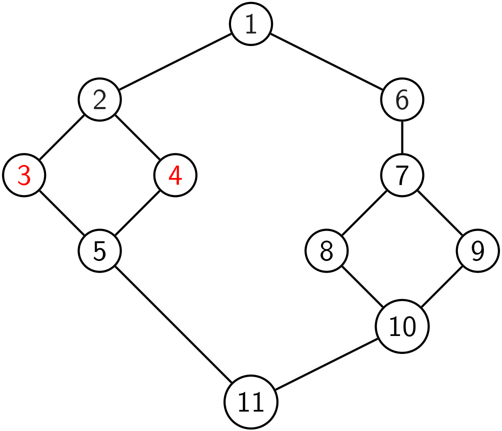
2-DFS Schedule



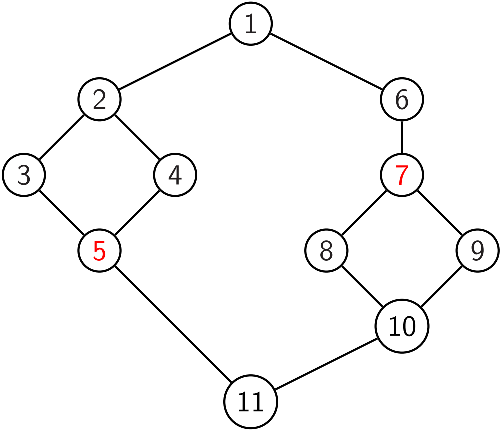
2-DFS Schedule



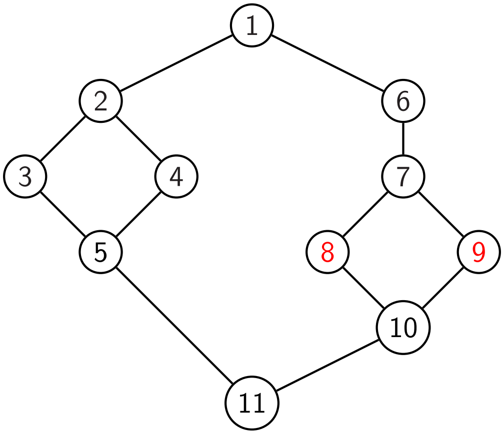
2-DFS Schedule



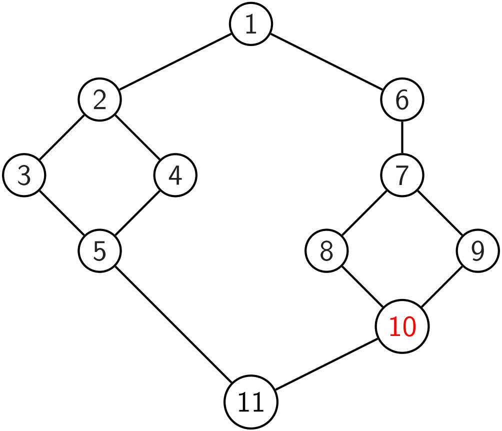
2-DFS Schedule



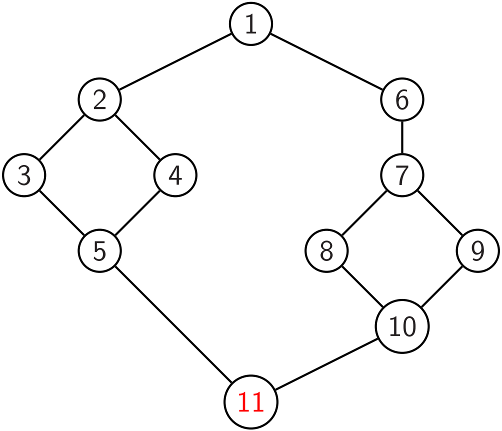
2-DFS Schedule



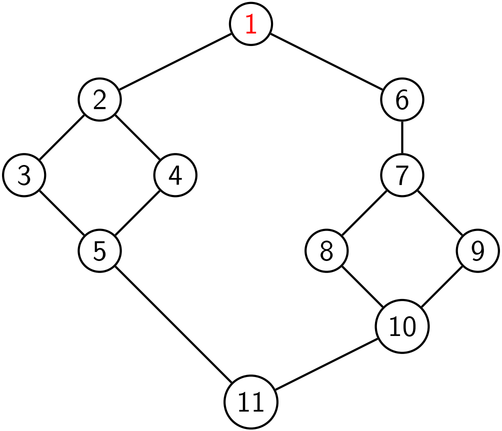
2-DFS Schedule



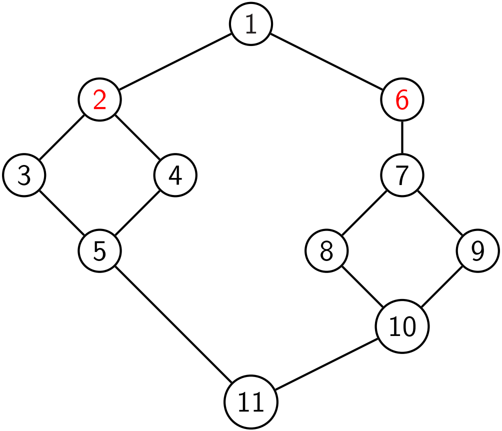
2-DFS Schedule



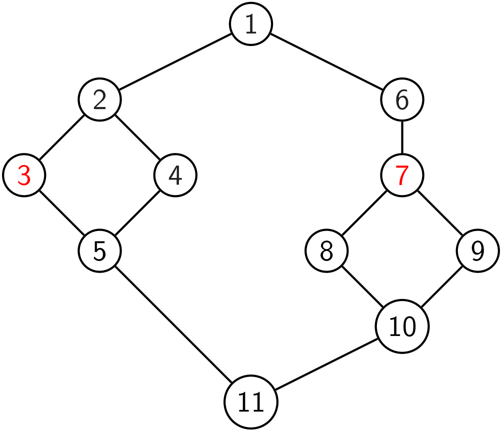
2-WS Schedule



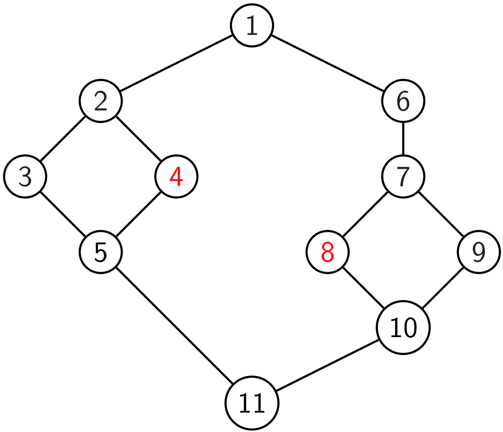
2-WS Schedule



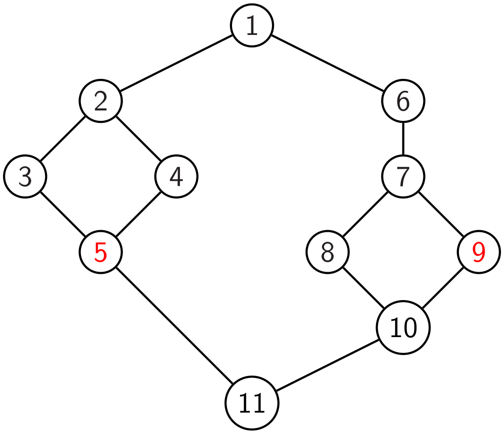
2-WS Schedule



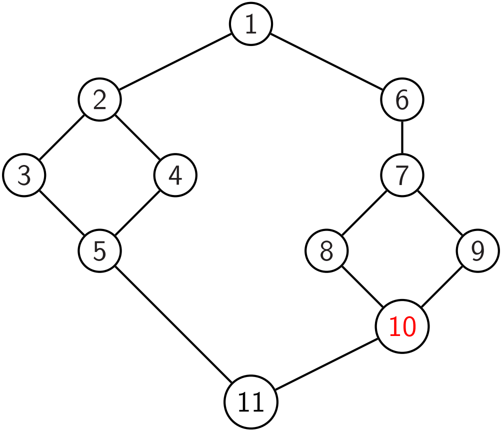
2-WS Schedule



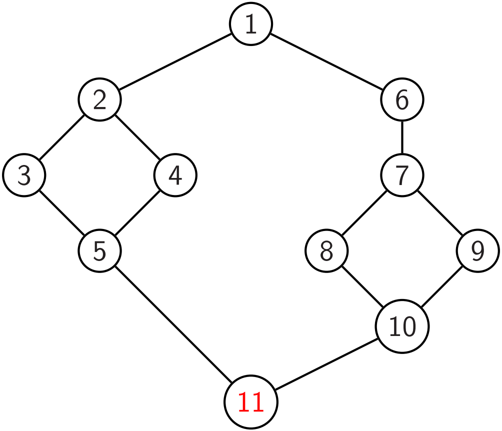
2-WS Schedule



2-WS Schedule



2-WS Schedule



Scheduling and Space

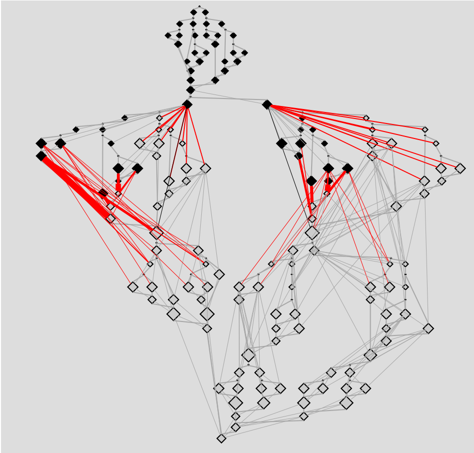
Key idea: measure the number of deviations from sequential order.

- Each deviation implies an interaction with the scheduler.
- Deviations incur cache misses.

Thm (Spoonhower): Space for scheduling is proportional to number of deviations.

Thm (Spoonhower, et al.): For parallel futures a work-stealing scheduler incurs expected $O(p\,d + t\,d)$ deviations on p processors with t touches.

Visualization of Cost Graphs



Red edges mark live roots at high-water mark.

Introductory CS at CMU

Introductory curriculum emphasizes:

- Parallelism as the general case, sequential being degenerate.
- Verification by rigorous proof.

The best way to achieve this is functional programming.

- 2nd semester: parallel FP, abstraction, verification.
- 3rd semester: parallel data structures and algorithms using FP.

See
`www.cs.cmu.edu/~15150/previous-semesters/2012-spring`
and
`www.cs.cmu.edu/afs/cs/academic/class/15210-s12/www/`.

Fallacies Refuted

It is often alleged that machine models are "realistic".

- Manual storage allocation.
- Manual scheduling.
- Primary and secondary storage effects.

But research developments have shown

- Automatic storage management is faster and more robust.
- Automatic scheduling is practical and efficient.

Even memory hierarchy effects can be accounted for cleanly and elegantly using cost semantics.

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:

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

(Not cache-oblivious: k is proportional to M/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:

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

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:

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

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.

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.

In-cache operations are zero cost; reads and evictions are unit cost.

(Simplified) Cost Semantics

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

(Simplified) Cost Semantics

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

(Simplified) Cost Semantics

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

(Simplified) Cost Semantics

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

Provable 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.

Provable 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.

Provable 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.

Provable 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.

Provable 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` are needed to ensure compactness (locality).

Stack Management

The main complication is accounting for the control stack.

- For `map` stack space may be amortized against allocation of the result.
- But this is not always possible!

Stack Management

The main complication is accounting for the control stack.

- For `map` stack space may be amortized against allocation of the result.
- But this is not always possible!

Consider non-tail recursive factorial:

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

Stack Management

The main complication is accounting for the control stack.

- For `map` stack space may be amortized against allocation of the result.
- But this is not always possible!

Consider non-tail recursive factorial:

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

Without accounting for stack, we would predict $O(1)$ cost, but the true cost is $O(n/B)$.

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{ \begin{array}{l} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 \\ \\ \\ \end{array} \right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'}_{R} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{ \begin{array}{ll} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{@} l_1' \\ & \\ & \end{array} \right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow_R^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{ll} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \quad \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \\ & \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow_R^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \sigma_1'' \mathbin{@} \mathtt{app}(l_1'; -) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} k_2 \\ & \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'}_{R} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{ll} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \sigma_1'' \mathbin{@} \mathtt{app}(l_1'; -) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} k_2 \\ \sigma_2 \mathbin{@} e_2 \Downarrow^{n_2}_{R \cup \{k_2\}} \sigma_2' \mathbin{@} l_2' & \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow_R^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \,@\, \mathtt{app}(-;e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \,@\, k_1 & \sigma_1 \,@\, e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \,@\, l_1' \\ \sigma_1' \,@\, l_1' \downarrow^{n_1''} \sigma_1'' \,@\, \lambda x.e & \sigma_1'' \,@\, \mathtt{app}(l_1';-) \uparrow^{n_1'''}_{R} \sigma_2 \,@\, k_2 \\ \sigma_2 \,@\, e_2 \Downarrow^{n_2}_{R \cup \{k_2\}} \sigma_2' \,@\, l_2' & \sigma_2' \,@\, [l_2'/x]e \Downarrow^{n_2'}_{R} \sigma' \,@\, l' \end{array}\right\}}{\sigma \,@\, \mathtt{app}(e_1;e_2) \Downarrow^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'}_{R} \sigma' \,@\, l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \sigma_1'' \mathbin{@} \mathtt{app}(l_1'; -) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} k_2 \\ \sigma_2 \mathbin{@} e_2 \Downarrow^{n_2}_{R \cup \{k_2\}} \sigma_2' \mathbin{@} l_2' & \sigma_2' \mathbin{@} [l_2'/x]e \Downarrow^{n_2'}_{R} \sigma' \mathbin{@} l' \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow_R^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \,@\, \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \,@\, k_1 & \sigma_1 \,@\, e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \,@\, l_1' \\ \sigma_1' \,@\, l_1' \downarrow^{n_1''} \sigma_1'' \,@\, \lambda x.e & \sigma_1'' \,@\, \mathtt{app}(l_1'; -) \uparrow^{n_1'''}_{R} \sigma_2 \,@\, k_2 \\ \sigma_2 \,@\, e_2 \Downarrow^{n_2}_{R \cup \{k_2\}} \sigma_2' \,@\, l_2' & \sigma_2' \,@\, [l_2'/x]e \Downarrow^{n_2'}_{R} \sigma' \,@\, l' \end{array}\right\}}{\sigma \,@\, \mathtt{app}(e_1; e_2) \Downarrow^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'}_{R} \sigma' \,@\, l'}$$

Modifications:

- Frames are never read, but just allocated for their effect.
- Root set R records live data in the control stack.

Stack Management

Stack frames are allocated in the nursery.

- May exist solely within nursery.
- May migrate to secondary memory.

Stack Management

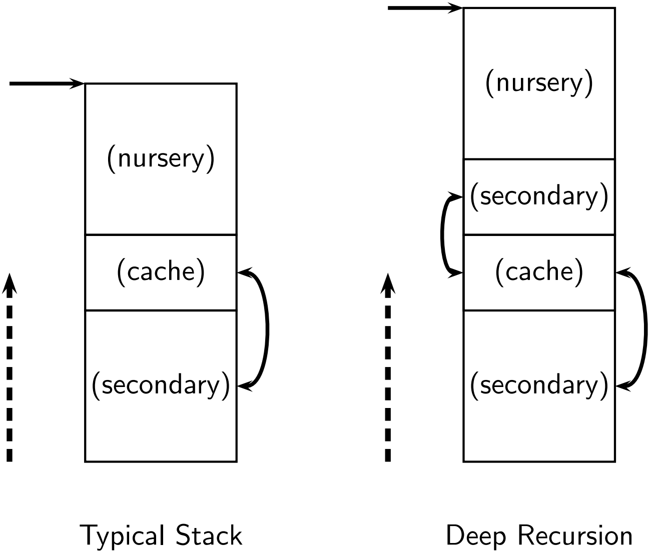
Stack frames are allocated in the nursery.

- May exist solely within nursery.
- May migrate to secondary memory.

Dedicate a cache block of B frames in primary memory.

- Not influenced by frames in nursery.
- Specially managed read cache for stack frames.

Stack Management



Stack Management

Stack cache block may be evicted up to B times.

- Newer frames may overflow nursery.
- Reading evicted frames replaces stack cache.

Stack Management

Stack cache block may be evicted up to B times.

- Newer frames may overflow nursery.
- Reading evicted frames replaces stack cache.

Amortize cost of eviction over allocation of newer frames.

- Put \$3 on each frame block as it is migrated to secondary.
- Use \$1 for migration.
- Use \$1 for initial load.
- Use \$1 for reload of evicted block.

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 effects 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 effects of scheduling [Spoonhower, B, Gibbons, & H 09].
- Memory hierarchy effects [B& H 13].

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically elegant.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically elegant.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically elegant.

Cost semantics integrates the combnatorial aspects:

- Enrich the tools available to algorithms designers.

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically elegant.

Cost semantics integrates the combnatorial aspects:

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

Summary

λ-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically elegant.

Cost semantics integrates the combnatorial aspects:

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

What's not to like?