Two Notions of Beauty in Programming

Robert Harper

Computer Science Department
Carnegie Mellon University

IU CSD Distinguished Lecture Series
November 2013
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).
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 \( \lambda \)-calculus is the key!
“On the fact that the Atlantic Ocean has two sides.” [EWD]

- American theory ≈ combinatorial theory.
- Euro-theory ≈ semantics and logic.
"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!
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.
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) $\lambda$-calculus.
- High-level: abstraction, composition are fundamental.
- Platform-independent.

Language models support composition via variables:

- If $\phi \text{true} \vdash \psi \text{true}$, then if $\phi \text{true}$, then $\psi \text{true}$.
- If $x : \sigma \vdash N : \tau$, then if $M : \sigma$, then $[M/x]N : \tau$.

The $\lambda$-calculus is an elegant theory of composition.
Languages based on $\lambda$-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.
- Concurrency control for mutation.

What should be a matter of **efficiency** becomes a matter of **correctness**!
procedure QUICKSORT(S):
1. if S contains at most one element then return S
else
2. begin
3. choose an element a randomly from S;
4. let S₁, S₂, and S₃ be the sequences of elements in S less than, equal to, and greater than a, respectively;
5. 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 i-th 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:

\[ T(n) \leq cn + \frac{1}{n} \sum_{i=1}^{n} [T(i−1) + T(n−i)], \quad \text{for} \quad n \geq 2. \quad (3.3) \]

Algebraic manipulation of (3.3) yields

\[ T(n) \leq cn + \frac{2}{n} \sum_{i=1}^{n} T(i). \quad (3.4) \]

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

\[ T(n) \leq cn + \frac{4b}{n} + \frac{2}{n} \sum_{i=1}^{n−1} ki \log_{n} i. \quad (3.5) \]

Since \( i \log_{n} i \) is concave upwards, it is easy to show that

\[ \sum_{i=2}^{n−1} i \log_{n} i \leq \int_{2}^{n} x \log_{n} x \, dx = \frac{n^2 \log_{n} n - n^2}{4}. \quad (3.6) \]

Substituting (3.6) in (3.5) yields
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.
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!
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.

\[
\begin{array}{ccccccc}
\text{\(e_1 \downarrow \lambda x.e\)} & \text{\(e_2 \downarrow \nu_2\)} & \text{\([\nu_2/x]e \downarrow \nu\)} & \\
\text{\(e_1(e_2) \downarrow \nu\)} & \\
\end{array}
\]
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.

\[
\begin{align*}
& e_1 \Downarrow^{g_1} \lambda x. e \\
& e_2 \Downarrow^{g_2} v_2 \\
& [v_2/x]e \Downarrow^{g} v \\
\hline
& e_1(e_2) \Downarrow v
\end{align*}
\]
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.

\[
\begin{align*}
\text{e}_1 \Downarrow^{g_1} \lambda x. e & \quad \text{e}_2 \Downarrow^{g_2} v_2 & \quad [v_2/x]e \Downarrow^{g} v \\
\text{e}_1(\text{e}_2) \Downarrow^{(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g} v
\end{align*}
\]
Series-parallel cost graphs:

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

Application cost \((g_1 \otimes g_2) \oplus \mathbf{1} \oplus g\) specifies that
Series-parallel cost graphs:

- \( 1 \): one unit of computation.
- \( g_1 \oplus g_2 \): \( g_2 \) depends on result of \( g_1 \).

Application cost \((g_1 \otimes g_2) \oplus 1 \oplus g\) specifies that
Series-parallel cost graphs:

- **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 \((g_1 \otimes g_2) \oplus 1 \oplus g\) specifies that
Series-parallel cost graphs:

- **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 ($g_1 \otimes g_2 \oplus 1 \oplus g$) specifies that
- Function and argument are evaluated in parallel.
Series-parallel cost graphs:

- \(1\): 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 \((g_1 \otimes g_2) \oplus 1 \oplus g\) 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
Operations on sequences have similar cost semantics:

\[
\begin{align*}
\text{map}(e; e') & \downarrow \\
\lambda x. e & \downarrow \quad e' & \downarrow \quad [v_1, \ldots, v_n] \\
[v_1/x]e & \downarrow \quad v'_1 & \ldots & [v_n/x]e & \downarrow \quad v'_n \\
\end{align*}
\]

To map a function over a sequence,
Cost Semantics

Operations on sequences have similar cost semantics:

\[ e \Downarrow^g \lambda x. e \quad e' \Downarrow^{g'} [v_1, \ldots, v_n] \]

\[ [v_1/x]e \Downarrow v'_1 \quad \ldots \quad [v_n/x]e \Downarrow v'_n \]

\[ \text{map}(e; e') \Downarrow [v'_1, \ldots, v'_n] \]

To map a function over a sequence,

- Evaluate the function and the sequence in parallel, and then
Operations on sequences have similar cost semantics:

\[
\begin{align*}
\text{map}(e; e') & \Downarrow \quad [v_1', \ldots, v_n'] \\
\end{align*}
\]

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:

\[ e \Downarrow^g \lambda x. e \quad e' \Downarrow^{g'} [v_1, \ldots, v_n] \]

\[ [v_1/x]e \Downarrow^{g_1} v'_1 \quad \ldots \quad [v_n/x]e \Downarrow^{g_n} v'_n \]

\[ \text{map}(e; e') \Downarrow (g \otimes g') \oplus (\bigotimes_i g_i) \oplus 1 \quad [v'_1, \ldots, v'_n] \]

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

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

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

- $d(1) = 1$, $d(g_1 \otimes g_2) = \max(d(g_1), d(g_2))$,
  $d(g_1 \oplus g_2) = d(g_1) + d(g_2)$.
- Measures the parallel time complexity.
fun merge xs ys =  
case (xs, ys) of  
  ([], ys) \Rightarrow ys 
| (xs,[]) \Rightarrow xs  
| (x::xs', y::ys') \Rightarrow  
    case x<y of  
      true \Rightarrow x :: merge xs' ys 
    | false \Rightarrow y :: merge xs ys' 

fun sort [] = []  
| sort [x] = [x]  
| sort xs =  
  let val (ys, zs) = split xs  
  in  merge (sort ys, sort zs)  end
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!
Brent’s Principle: A computation with work $w$ and span $d$ can be implemented on a $p$-processor PRAM in time $O\left(\max\left(\frac{w}{p}, d\right)\right)$.

- 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-WS Schedule

Graph representation of the schedule.
2-WS Schedule
2-WS Schedule
2-WS Schedule

Graph representation of a 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.
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/.
It is often alleged that machine models are “realistic”.

- Manual storage allocation.
- 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.
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:

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

(Not cache-oblivious: $k$ is proportional to $M/B$.)
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:

\[ \sigma \circ e \Downarrow^n \sigma' \circ v \]

Store \( \sigma \) has three components:
Cost Semantics for IO

Cost semantics makes storage explicit:

\[ \sigma \circ e \downarrow^n \sigma' \circ v \]

Store \( \sigma \) has three components:

- Unbounded main memory with blocks of size \( B \).
Cost Semantics for IO

Cost semantics makes storage explicit:

\[ \sigma \odot e \Downarrow^n \sigma' \odot v \]

Store \( \sigma \) 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:

\[ \sigma \oplus e \downarrow^n \sigma' \oplus v \]

Store \( \sigma \) 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:

\[ \sigma @ e \Downarrow^n \sigma' @ v \]

Store \( \sigma \) 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

\[
\sigma \odot \text{app}(e_1; e_2) \Downarrow n_1' + n_1'' + n_2' + n_2'' \sigma' \odot l'
\]

\[
\begin{align*}
\sigma_1 & \odot e_1 \Downarrow n_1' & \sigma_1' & \odot l_1'
\end{align*}
\]
(Simplified) Cost Semantics

\[
\begin{align*}
\sigma_1 @ e_1 \downarrow^{n'_1} & \quad \sigma'_1 @ l'_1 \\
\left\{\sigma'_1 @ l'_1 \downarrow^{n''_1} \sigma''_1 @ \lambda x. e\right\} & \\
\sigma @ \text{app}(e_1; e_2) \downarrow & \quad n'_1 + n''_1 + n'_2 + n''_2 \quad \sigma' @ l'
\end{align*}
\]
(Simplified) Cost Semantics

\[
\begin{align*}
\sigma @ \text{app}(e_1; e_2) \downarrow & \quad n_1' + n_1'' + n_2 + n_2' \quad \sigma' @ l'' \\
\sigma_1 @ e_1 \downarrow & \quad n_1' \quad \sigma_1 @ l'_1 \\
\sigma_1' @ l'_1 \downarrow & \quad n_1'' \quad \sigma_1'' @ \lambda x. e \\
\sigma_1'' @ e_2 \downarrow & \quad n_2 \quad \sigma_2' @ l'_2
\end{align*}
\]
(Simplified) Cost Semantics

\[
\begin{align*}
\{ &\sigma_1 \odot e_1 \downarrow^{n_1'} \quad \sigma_1' \odot l_1' \\
\sigma_1' \odot l_1' \downarrow^{n_1''} \quad \sigma_1'' \odot \lambda x. e \\
\sigma_1'' \odot e_2 \downarrow^{n_2} \quad \sigma_2' \odot l_2' \\
\sigma_2' \odot [l_2'/x]e \downarrow^{n_2'} \quad \sigma' \odot l' \\
\sigma \odot \text{app}(e_1; e_2) \downarrow^{n_1' + n_1'' + n_2 + n_2'} \quad \sigma' \odot l' \}
\end{align*}
\]
Thm (Bleloch & 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 $k$. 
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.
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.
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.
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’
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'
A data structure is compact iff it may be traversed in time $O(n/B)$.

Thm: For compact inputs $xs$ and $ys$ the call $\text{merge } xs \; ys$ 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).
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!
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:

```haskell
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:

```haskell
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:
The cost semantics must be enhanced to allocate frames:

\[
\begin{array}{l}
\sigma \circ \text{app}(-; e_2) \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \circ k_1 \\
\sigma \circ \text{app}(e_1; e_2) \downarrow_{R}^{n_1+n_1'+n_1''+n_2+n_2'} \sigma' \circ l'
\end{array}
\]

Modifications:
• Frames are never read, but just allocated for their effect.
• Root set $R$ records live data in the control stack.
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\sigma \odot \text{app}(\_; e_2) & \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \odot k_1 & \sigma_1 \odot e_1 & \downarrow_{R \cup \{k_1\}}^{n_1'} \sigma_1' \odot l_1' \\
\sigma \odot \text{app}(e_1; e_2) & \downarrow_{R}^{n_1+n_1'+n_1''+n_2+n_2'} \sigma' \odot l'
\end{align*}
\]
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\sigma @ \text{app}(\sigma_2; e_2) & \uparrow_{R \cup \text{loc}(e_1)}^{n_1} \sigma_1 @ k_1 & \sigma_1 @ e_1 & \downarrow_{R \cup \{k_1\}}^{n_1'} \sigma_1' @ l_1' \\
\sigma_1' @ l_1' & \downarrow_{R}^{n_1''} \sigma_1'' @ \lambda x. e & \\
\sigma @ \text{app}(e_1; e_2) & \downarrow_{R}^{n_1 + n_1' + n_1'' + n_2 + n_2'} \sigma' @ l' 
\end{align*}
\]

Modifications:
- Frames are never read, but just allocated for their effect.
- Root set \( R \) records live data in the control stack.
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\sigma \odot \text{app}(\_, e_2) & \uparrow^{n_1}_R \cup \text{locs}(e_1) \quad \sigma_1 \odot k_1 \\
\sigma_1 \odot e_1 & \downarrow^{n'_1}_R \cup \{k_1\} \quad \sigma'_1 \odot l'_1 \\
\sigma'_1 \odot l'_1 \downarrow^{n''_1}_l \sigma''_1 \odot \lambda x. e \\
\sigma''_1 \odot \text{app}(l'_1; \_) & \uparrow^{n'''_1}_R \quad \sigma_2 \odot k_2
\end{align*}
\]

\[
\sigma \odot \text{app}(e_1; e_2) \downarrow^{n_1+n'_1+n''_1+n'''_1+n_2+n'_2}_R \quad \sigma' \odot l''
\]
The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
&\sigma \circ \text{app}(\mathord{\_}; e_2) \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \circ k_1 \\
&\sigma_1 \circ e_1 \downarrow_{R \cup \{k_1\}}^{n_1'} \sigma_1' \circ l_1' \\
&\sigma_1' \circ l_1' \downarrow_{l_1'}^{n_1''} \sigma_1'' \circ \lambda x. e \\
&\sigma_1'' \circ \text{app}(l_1'; \mathord{\_}) \uparrow_{R}^{n_1'''} \sigma_2 \circ k_2 \\
&\sigma_2 \circ e_2 \downarrow_{R \cup \{k_2\}}^{n_2} \sigma_2' \circ l_2' \\
&\sigma \circ \text{app}(e_1; e_2) \downarrow_{R}^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'} \sigma' \circ l'
\end{align*}
\]
Cost Semantics for IO

The cost semantics must be enhanced to **allocate** frames:

\[
\begin{aligned}
\sigma @ \text{app}(\varepsilon_1; \varepsilon_2) & \uparrow_{R \cup \text{locs}(\varepsilon_1)}^{n_1} \sigma_1 @ k_1 & \sigma_1 @ \varepsilon_1 & \downarrow_{R \cup \{k_1\}}^{n_1'} \sigma_1' @ l_1' \\
\sigma_1' @ l_1' & \downarrow_{R \cup \{k_2\}}^{n_1''} \sigma_1'' @ \lambda x. \varepsilon & \sigma_1'' @ \text{app}(l_1'; \varepsilon_2) & \uparrow_{R}^{n_1'''} \sigma_2 @ k_2 \\
\sigma_2 @ \varepsilon_2 & \downarrow_{R \cup \{k_2\}}^{n_2} \sigma_2' @ l_2' & \sigma_2' @ [l_2'/x] \varepsilon & \downarrow_{R}^{n_2'} \sigma @ l' \\
\sigma @ \text{app}(\varepsilon_1; \varepsilon_2) & \downarrow_{R}^{n_1+n_1'+n_1'''+n_2+n_2'} \sigma @ l'
\end{aligned}
\]

**Modifications:**
- Frames are never read, but just allocated for their effect.
- Root set \( R \) records live data in the control stack.
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\sigma \circ \text{app}(\_; e_2) & \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \circ k_1 \quad \sigma_1 \circ e_1 \downarrow_{R \cup \{k_1\}}^{n_1'} \sigma_1' \circ l_1' \\
\sigma_1' \circ l_1' & \downarrow_{R \cup \{k_2\}}^{n_1'} \sigma_1'' \circ \lambda x. e \quad \sigma_1'' \circ \text{app}(l_1'; \_) \uparrow_{R}^{n_1'''} \sigma_2 \circ k_2 \\
\sigma_2 \circ e_2 & \downarrow_{R \cup \{k_2\}}^{n_2} \sigma_2' \circ l_2' \quad \sigma_2' \circ \left[l_2'/x\right] e \downarrow_{R}^{n_2'} \sigma' \circ l'
\end{align*}
\]

\[
\sigma \circ \text{app}(e_1; e_2) \downarrow_{R}^{n_1+n_1'+n_1''+n_2+n_2'} \sigma' \circ l'
\]
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\sigma \odot \text{app}(-; e_2) \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \odot k_1 & \quad \sigma_1 \odot e_1 \downarrow_{R \cup \{k_1\}}^{n'_1} \sigma'_1 \odot l'_1 \\
\sigma'_1 \odot l'_1 \downarrow_{R \cup \{k_2\}}^{n''_1} \sigma''_1 \odot \lambda x. e & \quad \sigma''_1 \odot \text{app}(l'_1; -) \uparrow_{R}^{n'''_1} \sigma_2 \odot k_2 \\
\sigma_2 \odot e_2 \downarrow_{R \cup \{k_2\}}^{n_2} \sigma'_2 \odot l'_2 & \quad \sigma'_2 \odot [l'/x] e \downarrow_{R}^{n_2} \sigma' \odot l' \\
\sigma \odot \text{app}(e_1; e_2) \downarrow_{R}^{n_1 + n'_1 + n''_1 + n'''_1 + n_2 + n_2'} \sigma' \odot l'
\end{align*}
\]

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 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

Typical Stack

- (nursery)
- (cache)
- (secondary)

Deep Recursion

- (nursery)
- (secondary)
- (cache)
- (secondary)
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.
Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".

Summary
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].
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].
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].
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].
\(\lambda\)-calculus provides a logical model of computation.

- Inherently compositional.

What's not to like?
Summary

\(\lambda\)-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically elegant.
Summary

$\lambda$-calculus provides a logical model of computation.

- Inherently compositional.
- Mathematically elegant.
Summary

$\lambda$-calculus provides a **logical** model of computation.
- Inherently compositional.
- Mathematically elegant.

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 elegant.

Cost semantics integrates the combinatorial aspects:

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

\(\lambda\)-calculus provides a *logical* model of computation.

- Inherently compositional.
- Mathematically elegant.

Cost semantics integrates the *combinatorial* aspects:

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

What’s not to like?