Two Notions of Beauty in Programming

Robert Harper
(With Guy E. Blelloch)

Computer Science Department
Carnegie Mellon University

In Honor of Dana Scott’s 80th Birthday
Computation is Beautiful

The signature feature of Dana’s work is its beauty.

- Mathematically, e.g. the topological interpretation.
- Expressively, e.g. computable functionals.
- Practically, e.g. denotational semantics for compilation.
The signature feature of Dana’s work is its beauty.

- Mathematically, e.g. the topological interpretation.
- Expressively, e.g. computable functionals.
- Practically, e.g. denotational semantics for compilation.

Dana showed us that Church’s $\lambda$-calculus was the key to a practical and elegant theory of computation.
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 key is to follow Dana’s advice and use $\lambda$-calculus!
Reconciling the Two Theories

The problems are not (solely) social, but technical:

- Machine-based models do not support composition.
- Cost measures for $\lambda$-based models are lacking.
Reconciling the Two Theories

The problems are not (solely) social, but technical:

- Machine-based models do not support composition.
- Cost measures for $\lambda$-based models are lacking.

Consequently,

- Algorithms are analyzed in isolation.
- Higher-order methods are disregarded.
- Verification rarely considers complexity.
Traditionally, the **cost** of a computation is measured in two ways:

- **Time**: number of instructions in a RAM.
- **Space**: number of words of storage required.
Traditionally, the **cost** of a computation is measured in two ways:

- **Time**: number of instructions in a RAM.
- **Space**: number of words of storage required.

Algorithms are defined in *pseudo-code* and analyzed in terms of a presumed *compilation* to a machine.
Traditionally, the cost of a computation is measured in two ways:
  - **Time**: number of instructions in a RAM.
  - **Space**: number of words of storage required.

Algorithms are defined in pseudocode and analyzed in terms of a presumed compilation to a machine.

The practice leads to an artificial distinction between an algorithm and a program.
Cost Semantics for Real Code

Goal: work in a realistic language based on the \( \lambda \)-calculus.

- No pseudo-code, only real code!
- Higher-order data structures and algorithms.
Goal: work in a realistic language based on the $\lambda$-calculus.

- No pseudo-code, only real code!
- Higher-order data structures and algorithms.

But how do we analyze their cost?

- **Cost semantics** defines an abstract measure of complexity (time, space, I/O).
- **Provable implementation** transfers abstract cost to concrete cost on a machine model.
Parallelism [B & Greiner 96]

Associate a dynamic dependency graph to an evaluation derivation.

- Records true, not approximate, data dependencies.
- Exposes inherent parallelism and sequentiality.
Parallelism [B & Greiner 96]

Associate a *dynamic dependency graph* to an evaluation derivation.

- Records *true*, not approximate, data dependencies.
- Exposes inherent parallelism and sequentiality.

Two *measures* of a cost graph $g$:

- **Work**, or sequential complexity: size of $g$.
- **Span**, or parallel complexity: diameter of $g$. 
Example: function application.

\[ \begin{align*} 
  e_1 \downarrow & \quad \lambda x. e \\
  e_2 \downarrow & \quad v_2 \\
  [v_2/x]e \downarrow & \quad v \\
  e_1(e_2) \downarrow & \quad v 
\end{align*} \]
Example: function application.

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

\[
e_1(e_2) \Downarrow v
\]
Parallelism

Example: function application.

\[
\begin{align*}
  e_1 & \Downarrow^{g_1} \lambda x. e \\
  e_2 & \Downarrow^{g_2} v_2 \\
  [v_2/x]e & \Downarrow^{g} v \\
  e_1(e_2) & \Downarrow^{(g_1 \otimes g_2) \oplus 1 \oplus g} v
\end{align*}
\]
Cost Graphs

\[ \text{Work} = w_1 + w_2 + w + 1, \quad \text{Span} = \max(s_1, s_2) + 1 + s. \]
Brent’s Theorem: A computation with work $w$ and span $s$ can be implemented on a $p$-processor PRAM in time $O(w/p + s)$.

- Work in chunks of $p$ as much as possible.
- Proof is constructive: it exhibits a scheduler.
Brent’s Theorem: A computation with work $w$ and span $s$ can be implemented on a $p$-processor PRAM in time $O(w/p + s)$.

- Work in chunks of $p$ as much as possible.
- Proof is constructive: it exhibits a scheduler.

A schedule is a strategy in the $p$-pebble game for the dependency graph.

- Given at most $p$ pebbles, move a pebble from source to sink.
- If all input nodes are pebbled, remove them, and put a pebble on the output.
A cost semantics is crucial for teaching parallelism!

- Clear depiction of dependencies.
- Schedules are easily envisioned as strategies.
- Run-time implements a $p$-pebbling.
Teaching Parallelism

A cost semantics is crucial for teaching parallelism!

- Clear depiction of dependencies.
- Schedules are easily envisioned as strategies.
- Run-time implements a $p$-pebbling.

Useful for semantic profiling of parallel code:

- Eliminates confounding OS effects.
- Provides predictions to assess measurements.
Red edges show live data at high-water mark.
IO Model [Aggarwal & Vitter 88]

RAM-based IO model:

- Unbounded secondary memory, bounded primary memory.
- Cost = blocked transfers between primary and secondary.
IO Model [Aggarwal & Vitter 88]

RAM-based IO model:

- Unbounded secondary memory, bounded primary memory.
- Cost = blocked transfers between primary and secondary.

Example results:

- Matrix multiply without blocking: $O(n^3 / B)$.
- ...with blocking: $O(n^3 / (B \sqrt{M}))$.
- 2-way merge sort: $O((n/B) \log_2(n/B))$.
- ...$M/B$-way: $O((n/B) \log_{(M/B)}(n/B))$.
RAM-based IO model:

- Unbounded secondary memory, bounded primary memory.
- Cost = blocked transfers between primary and secondary.

Example results:

- Matrix multiply without blocking: $O(n^3/B)$.
- ...with blocking: $O(n^3/(B \sqrt{M})$.
- 2-way merge sort: $O((n/B) \log_2(n/B))$.
- ...$M/B$-way: $O((n/B) \log_{(M/B)}(n/B))$

Memory management done by hand!
Replicate A&V results in a purely functional language model.
- Abstract costs reflect memory traffic.

Restriction to machine models is not essential!
Replicate A&V results in a purely functional language model.

- Abstract costs reflect memory traffic.
- Provably efficient implementation on A&V model.

Restriction to machine models is not essential!
Replicate A&V results in a purely functional language model.
  • Abstract costs reflect memory traffic.
  • Provably efficient implementation on A& V model.
  • Purely functional code, not pseudo-code.

Restriction to machine models is not essential!
Simplified Cost Semantics for IO

Evaluation: \( \sigma @ e \Downarrow^n \sigma' @ l \).

- All values are allocated at a location in storage.
Simplified Cost Semantics for IO

Evaluation: \( \sigma \circ e \downarrow^n \sigma' \circ l \).

- All values are allocated at a location in storage.
- Cost \( n \) measures memory traffic.
Simplified Cost Semantics for IO

Evaluation: $\sigma \circ e \downarrow^n \sigma' \circ l$.

- All values are allocated at a location in storage.
- Cost $n$ measures memory traffic.
Simplified Cost Semantics for IO

Evaluation: $\sigma @ e \downarrow^n \sigma' @ l$.

- All values are allocated at a location in storage.
- Cost $n$ measures memory traffic.

Storage model: $\sigma = (\mu, \rho, \nu)$ [Morrisett, Felleisen, & H. 95]

- $\mu$: unbounded secondary memory with blocks of size $B$. 
Simplified Cost Semantics for IO

Evaluation: $\sigma@e \Downarrow^n \sigma'@l$.
- All values are allocated at a location in storage.
- Cost $n$ measures memory traffic.

Storage model: $\sigma = (\mu, \rho, \nu)$ [Morrisett, Felleisen, & H. 95]
- $\mu$: unbounded secondary memory with blocks of size $B$.
- $\rho$: bounded primary memory of size $M = k \times B$. 
Simplified Cost Semantics for IO

**Evaluation:** \( \sigma @ e \Downarrow^n \sigma' @ l \).

- All values are allocated at a location in storage.
- Cost \( n \) measures memory traffic.

**Storage model:** \( \sigma = (\mu, \rho, \nu) \) [Morrisett, Felleisen, & H. 95]
  - \( \mu \): unbounded secondary memory with blocks of size \( B \).
  - \( \rho \): bounded primary memory of size \( M = k \times B \).
  - \( \nu \): nursery of size \( M \) with a linear ordering on its domain.
Simplified Cost Semantics for IO

Evaluation: $\sigma \Downarrow^n e \Downarrow^n \sigma' \Downarrow l$.

- All values are allocated at a location in storage.
- Cost $n$ measures memory traffic.

Storage model: $\sigma = (\mu, \rho, \nu)$ [Morrisett, Felleisen, & H. 95]

- $\mu$: unbounded secondary memory with blocks of size $B$.
- $\rho$: bounded primary memory of size $M = k \times B$.
- $\nu$: nursery of size $M$ with a linear ordering on its domain.
Simplified Cost Semantics for IO

Evaluation: $\sigma @ e \Downarrow^n \sigma' @ l$.
- All values are allocated at a location in storage.
- Cost $n$ measures memory traffic.

Storage model: $\sigma = (\mu, \rho, \nu)$ [Morrisett, Felleisen, & H. 95]
- $\mu$: unbounded secondary memory with blocks of size $B$.
- $\rho$: bounded primary memory of size $M = k \times B$.
- $\nu$: nursery of size $M$ with a linear ordering on its domain.

Key invariant: temporal locality implies spatial locality.
Simplified Cost Semantics

Read: \( \sigma @ l \downarrow^n \sigma' @ v \).

- Read location \( l \) from store \( \sigma \) to obtain value \( v \).
- Cost accounts for loads to and evictions from primary.
- Ideal Cache Model: always evict latest required block.
Simplified Cost Semantics

Read: \( \sigma \circ l \downarrow^n \sigma' \circ v \).
- Read location \( l \) from store \( \sigma \) to obtain value \( v \).
- Cost accounts for loads to and evictions from primary.
- **Ideal Cache Model**: always evict latest required block.

Allocate: \( \sigma \circ v \uparrow^n \sigma' \circ l \).
- Allocate value \( v \) in \( \sigma \) obtaining \( \sigma' \) and \( l \).
- Cost \( n \) accounts for migration to secondary.
- Live objects are **blocked** on migration to secondary.
Simplified Cost Semantics

Functions are allocated in memory:

\[
\begin{align*}
\sigma \circ \lambda x. e \uparrow^n & \quad \sigma' \circ l \\
\sigma \circ \lambda x. e \downarrow^n & \quad \sigma' \circ l
\end{align*}
\]
Simplified Cost Semantics

Functions are allocated in memory:

\[ \sigma @ \lambda x. e \uparrow^n \sigma' @ l \]
\[ \sigma @ \lambda x. e \downarrow^n \sigma' @ l \]

Application follows pointers:

\[ \sigma @ \text{app}(e_1; e_2) \downarrow^{n'_1 + n''_1 + n'_2 + n''_2} \sigma' @ l'' \]
Simplified Cost Semantics

Functions are \textbf{allocated} in memory:

\[
\begin{align*}
\sigma \circ \lambda x. e & \uparrow^n \sigma' \circ \ell \\
\sigma \circ \lambda x. e & \downarrow^n \sigma' \circ \ell
\end{align*}
\]

Application \textbf{follows} pointers:

\[
\begin{align*}
\{ \\
s_1' \circ \ell_1' & \downarrow^{n_1''} s_2'' \circ \lambda x. e \\
\} & \\
\sigma \circ \text{app}(e_1; e_2) & \downarrow^{n_1' + n_1'' + n_2' + n_2'} \sigma' \circ \ell''
\end{align*}
\]
Simplified Cost Semantics

Functions are allocated in memory:

\[ \begin{align*}
\sigma \mapsto \lambda x.e & \uparrow^n \sigma' \mapsto l \\
\sigma \mapsto \lambda x.e & \downarrow^n \sigma' \mapsto l
\end{align*} \]

Application follows pointers:

\[ \begin{align*}
\left\{ \begin{align*}
\sigma' \mapsto l' & \downarrow_{n'} \sigma'' \mapsto \lambda x.e \\
\sigma'' \mapsto e_2 & \downarrow{n_2} \sigma' \mapsto l'' \\
\sigma \mapsto \text{app}(e_1; e_2) & \downarrow n_1 + n'' + n_2 + n' \sigma' \mapsto l''
\end{align*} \right. \]
Simplified Cost Semantics

Functions are **allocated** in memory:

\[
\sigma @ \lambda x. e \uparrow^n \sigma' @ I \\
\sigma @ \lambda x. e \downarrow^n \sigma' @ I
\]

Application **follows** pointers:

\[
\begin{align*}
\sigma_1 @ \text{app}(e_1; e_2) \downarrow^{n_1' + n_1'' + n_2 + n_2'} & \quad \sigma' @ I' \\
\sigma_1 @ e_1 \downarrow^{n_1'} & \quad \sigma' @ I' \\
\sigma_1 @ l_1' \downarrow^{n_1''} & \quad \sigma_1 @ \lambda x. e \\
\sigma_2 @ e_2 \downarrow^{n_2} & \quad \sigma_2 @ l_2' \\
\sigma_2 @ [l_2'/x] e \downarrow^{n_2'} & \quad \sigma' @ I' \\
\end{align*}
\]
Example: Map

Mapping over a list:

```haskell
fun map f nil = nil
  | map f (h::t) = (f t) :: map f t
```
Mapping over a list:

```haskell
fun map f nil = nil
| map f (h::t) = (f t) :: map f t
```

**Definition** A list is **compact** if it can be traversed in time $O(n/B)$.

- Intuitively, it is not scattered through memory.
- Robust with respect to forward or backward traversal.
Example: Map

Mapping over a list:

```latex
fun map f nil = nil
    | map f (h::t) = (f t) :: map f t
```

**Definition** A list is **compact** if it can be traversed in time $O(n/B)$.

- Intuitively, it is not scattered through memory.
- Robust with respect to forward or backward traversal.

**Theorem** If $l$ is compact and $f$ is simple, then $\text{map } f \ l$ is compact and has IO cost $O(n/B)$. 
Nearly standard implementation:

```
fun merge nil ys = ys
  | merge xs nil = xs
  | merge (xs as x::xs') (ys as y::ys') =
    case compare x y of
    LESS ⇒ id x::merge xs' ys
    | GTEQ ⇒ id y::merge xs ys'
```
Example: Merge

Nearly standard implementation:

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

The applications `id x` and `id y` are “hard-way” identities that reconstruct their arguments to preserve compactness.
Example: Merge Sort

**Theorem** For compact inputs of size $n$ and simple comparison, `merge xs ys` has cost $O(n/B)$. 
**Example: Merge Sort**

**Theorem** For compact inputs of size $n$ and simple comparison, `merge xs ys` has cost $O(n/B)$.

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

**Theorem** For compact inputs of size $n$ and simple comparison, 
merge $\text{merge } \text{xs } \text{ys}$ has cost $O(n/B)$.

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

**Theorem** For compact input of size $n$, sort $\text{sort } \text{xs}$ has cost $O((n/B) \log_{(M/B)}(n/B))$. 
Example: Merge Sort

**Theorem** For compact inputs of size \( n \) and simple comparison, \( \text{merge } \text{xs } \text{ys} \) has cost \( O(n/B) \).

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

**Theorem** For compact input of size \( n \), \( \text{sort } \text{xs} \) has cost \( O((n/B) \log_{(M/B)}(n/B)) \).

(Matches A&V bound in IO model.)
“Theorem” If $\sigma \circ e \downarrow^n \sigma' \circ l$, then $e$ may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$. However, the "theorem" is not correct as stated...
“Theorem” If $\sigma \circ e \downarrow^n \sigma' \circ l$, then $e$ may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$. 
“Theorem” If $\sigma @ e \Downarrow^n \sigma' @ l$, then $e$ may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$.
- LRU is 2-competitive with ICM [Sleator & Tarjan 85]: $2 \times M$. 
“Theorem” If $\sigma \circ e \downarrow^n \sigma' \circ I$, then $e$ may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$.
- LRU is 2-competitive with ICM [Sleator & Tarjan 85]: $2 \times M$. 
“Theorem” If $\sigma \circ e \Downarrow^n \sigma' \circ I$, then $e$ may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$.
- LRU is 2-competitive with ICM [Sleator & Tarjan 85]: $2 \times M$.

However, the “theorem” is not correct as stated ....
Simplified semantics does not account for control stack.

- For programs such as `map`, control stack space may be amortized against allocation of result.
- But this is not always possible!
Simplified semantics does not account for control stack.

- For programs such as `map`, control stack space may be amortized against allocation of 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

Simplified semantics does not account for control stack.

- For programs such as `map`, control stack space may be amortized against allocation of result.
- But this is not always possible!

Consider non-tail recursive factorial:

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

Over-simplified semantics predicts $O(1)$ cost, but the true cost is $O(n/B)$ due to the control stack.
Theorem If $\sigma \circ e \downarrow^\mathcal{R}_R n \sigma' \circ l$, then $e$ can be executed in the IO model in time $k \times n$ using a primary cache of size $4 \times M + B$. 
Theorem If $\sigma @ e \downarrow^R_n \sigma' @ l$, then $e$ can be executed in the IO model in time $k \times n$ using a primary cache of size $4 \times M + B$.

Require major steps:

- Enhance cost semantics to allocate frames.
- Implement cost semantics on a stack machine.
- Implement stack machine on A&V IO model.
The cost semantics must be enhanced to allocate frames:
Cost Semantics for IO

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

\[
\begin{align*}
\sigma \circ \text{app}(-; e_2) \uparrow^{n_1}_{R \cup \text{locs}(e_1)} & \sigma_1 \circ k_1 \\
\sigma \circ \text{app}(e_1; e_2) \downarrow^{n_1 + n'_1 + n''_1 + n'''_1 + n_2 + n'_2}_R & \sigma' \circ \lambda x. e \downarrow^{n_1 + n'_1 + n''_1 + n'''_1 + n_2 + n'_2} \sigma' \circ l''
\end{align*}
\]

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:

\[
\begin{cases}
\sigma \circ \text{app}(-; e_2) \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \circ k_1 \\
\sigma_1 \circ e_1 \downarrow_{R \cup \{k_1\}}^{n_1'} \sigma_1' \circ l_1'
\end{cases}
\]

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

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:

\[
\begin{align*}
\sigma \circ \text{app}(\_, e_2) &\uparrow^{n_1}_{R \cup \text{locs}(e_1)} \sigma_1 \circ k_1 \\
\sigma_1 &\circ e_1 \downarrow^{n'_1}_{R \cup \{k_1\}} \sigma'_1 \circ l'_1 \\
\sigma'_1 &\circ l'_1 \downarrow^{n''_1} \sigma''_1 \circ \lambda x. e \\
\sigma \circ \text{app}(e_1; e_2) &\downarrow^{n_1+n'_1+n''_1+n'''_1+n_2+n'_2}_{R} \sigma' \circ l''
\end{align*}
\]

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:

$$\begin{align*}
\sigma @ \text{app}(\_; e_2) & \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 @ k_1 \\
\sigma_1 @ e_1 & \downarrow_{R \cup \{k_1\}}^{n'_1} \sigma_1' @ l_1' \\
\sigma_1' @ l_1' & \downarrow_{R}^{n''_1} \sigma_1' @ \lambda x. e \\
\sigma_1'' @ \text{app}(l_1'; \_) & \uparrow_{R}^{n'''_1} \sigma_2 @ k_2 \\
\sigma @ \text{app}(e_1; e_2) & \downarrow_{R}^{n_1+n_1'+n''_1+n'''_1+n_2+n_2'} \sigma' @ l''
\end{align*}$$

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:

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

The cost semantics must be enhanced to allocate frames:

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

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:

\[
\begin{align*}
\sigma @ \text{app}(-; e_2) &\gtrdot^{n_1}_R \text{loc}_1(e_1) \quad \sigma_1 @ k_1 &\quad \sigma_1 @ e_1 &\gtrdot^{n'_1}_R \text{loc}_1\{k_1\} \quad \sigma'_1 @ l'_1 \\
\sigma'_1 @ l'_1 &\gtrdot^{n''_1}_l \lambda x. e &\quad \sigma''_1 @ \text{app}(l'_1; -) &\gtrdot^{n''''_1}_R \quad \sigma_2 @ k_2 \\
\sigma_2 @ e_2 &\gtrdot^{n_2}_R \text{loc}_2\{k_2\} \quad \sigma'_2 @ l'_2 &\quad \sigma'_2 @ [l'_2/x]e &\gtrdot^{n'_2}_R \quad \sigma' @ l' \\
\sigma @ \text{app}(e_1; e_2) &\gtrdot^{n_1+n'_1+n''_1+n''''_1+n_2+n'_2}_R \quad \sigma' @ l'
\end{align*}
\]
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

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

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

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

Typical Stack

Deep Recursion
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".

[B & Greiner 96]

[Spoonhower, B, Gibbons, & H 09]

[B & H 13]
Cost semantics supports analysis of complexity of high-level code.

- No need for “pseudo-code”.
- Avoid reasoning about compilation.
Cost semantics supports analysis of complexity of high-level code.

- No need for “pseudo-code”.
- Avoid reasoning about compilation.
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.
Summary

$\lambda$-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

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

- Inherently compositional.
- Mathematically elegant.

Cost semantics integrates the combinatorial aspects:

- Enrich the tools available to algorithms designers.
\(\lambda\)-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.
Thanks, Dana!

Dana stressed the $\lambda$-caclulus as the *primus inter pares* among models of computation.
Thanks, Dana!

Dana stressed the *λ*-calculus as the *primus inter pares* among models of computation.

May his influence continue to guide us in the future!
Thanks, Dana!

Dana stressed the $\lambda$-calculus as the *primus inter pares* among models of computation.

May his influence continue to guide us in the future!

To be eighty years young is more cheerful and hopeful than forty years old.