Cost Semantics for Space Usage in a Parallel Language

Daniel Spoonhower

Carnegie Mellon University

(Joint work with Guy Blelloch & Robert Harper)

DAMP – 16 Jan 2007
Interested in **intensional** behavior of programs

- more than just final result
- *e.g.* time & space required
Interested in **intensional** behavior of programs

- more than just final result
- e.g. time & space required

State-of-the-art = compile, run, & profile
Interested in intensional behavior of programs
  ▶ more than just final result
  ▶ e.g. time & space required

State-of-the-art = compile, run, & profile
  ✗ architecture specific (e.g. # cores)
  ✗ dependent on configuration (e.g. scheduler)
  ✗ compilers for functional languages are complex (e.g. closure, CPS conversion)
Motivating Example: Quicksort

Assume fine-grained parallelism

- pairs \(< e_1 \mid\mid e_2 >\) may evaluate in parallel
- schedule determined by compiler & run-time

```haskell
fun qsort xs =
    case xs of nil => nil
           | x::xs =>
              append <qsort (filter (le x) xs) ||
                        x::(qsort (filter (gt x) xs))>
```

16 Jan 2007

DAMP '07 Cost Semantics for Space
Quicksort: High-Water Mark for Heap

![Graph showing space high-water mark over list size for configurations A, B, and C.](graph)
Approach

Cost Semantics
- define execution costs for high-level language
- account for parallelism & space

Provable Implementation
- make parallelism explicit
- translate to lower-level language
- prove costs are preserved at each step
- consider scheduler, GC implementation
Cost Semantics

- define execution costs for high-level language
- account for parallelism & space

Provable Implementation

- make parallelism explicit
- translate to lower-level language
- prove costs are preserved at each step
- consider scheduler, GC implementation
A *cost semantics* is a *dynamic* semantics

- *i.e.* execution model for high-level language

Yields a cost metric, some abstract measure of cost

- *e.g.* steps of evaluation, upper bound on space
A cost semantics is a dynamic semantics
▶ i.e. execution model for high-level language

Yields a cost metric, some abstract measure of cost
▶ e.g. steps of evaluation, upper bound on space

We will consider a cost model that accounts for parallelism and space.
Consider a pure, functional language.

- includes functions, pairs, and booleans

Pair components evaluated in parallel.

- denoted $< e_1 \parallel e_2 >$

Values are disjoint from source language.

- values are labeled to make sharing explicit
  - e.g. $(v_1, v_2)^\ell$
Cost semantics is a big-step (evaluation) semantics
- yields two graphs: computation and heap
- sequential, unique result per program

\[ e \Downarrow v; g; h \]

Expression \( e \) evaluates to value \( v \) with computation graph \( g \) and heap graph \( h \).
Computation Graphs

Track control dependencies using a DAG with distinguished start and end nodes.

\[ g = (n_{\text{start}}, n_{\text{end}}, E) \]
Computation Graphs

Track control dependencies using a DAG with distinguished start and end nodes.

\[ g = (n_{start}, n_{end}, E) \]

\[ 1 \quad [n] \quad g_1 \oplus g_2 \quad g_1 \otimes g_2 \]

\[ \bullet \quad \bullet n \]

\[ \text{DAMP '07} \quad \text{Cost Semantics for Space} \]
Heap Graphs

Track heap dependencies using a directed graph

\[ h = E \]

- nodes shared with corresponding \( g \)
- edges run in \textit{opposite} direction
Heap Graphs

Track heap dependencies using a directed graph

\[ h = E \]

- nodes shared with corresponding \( g \)
- edges run in opposite direction
Cost graphs are tools for programmers.

- relate execution costs to source code
- later: simulate runtime behavior

Many concrete metrics possible

- considered maximum heap size in example
- impact of GC: measure overhead, latency
Using Cost Graphs

Cost graphs are tools for programmers.
▶ relate execution costs to source code
▶ later: simulate runtime behavior

Many concrete metrics possible
▶ considered maximum heap size in example
▶ impact of GC: measure overhead, latency

However, this reasoning is only valid if the implementation respects these costs.
Provable Implementation

Guaranteed to faithfully mirror high-level costs
▶ “implementation” = lower-level semantics

Costs ⇒ contract for lower-level implementations
▶ e.g. environment trimming, tail calls
▶ can guide concrete implementation on hardware
Provable Implementation

Guaranteed to faithfully mirror high-level costs

- “implementation” = lower-level semantics

Costs ⇒ contract for lower-level implementations

- e.g. environment trimming, tail calls
- can guide concrete implementation on hardware

This work: transition semantics defines parallelism

- several (non-)deterministic versions
- can incorporate specific scheduling algorithms
Transition Semantics

Non-deterministic, parallel, small step semantics

- parallel construct for in-progress computations

(expressions) \( e ::= \ldots \mid \text{let par} \ d \ \text{in} \ e \)

(declarations) \( d ::= x = e \mid d_1 \ \text{and} \ d_2 \)
Transition Semantics

Non-deterministic, parallel, small step semantics

- parallel construct for in-progress computations

(expressions) \[ e ::= \ldots \mid \text{let par } d\text{ in } e \]

(declarations) \[ d ::= x = e \mid d_1 \text{ and } d_2 \]

- declarations simulate a call “stack”

- allows unbounded parallelism, e.g.

\[
\begin{align*}
  d_1 \rightarrow d'_1 & \quad d_2 \rightarrow d'_2 \\
  (d_1 \text{ and } d_2) \rightarrow (d'_1 \text{ and } d'_2)
\end{align*}
\]
Define a schedule of $g$ as any covering traversal from $n_{start}$ to $n_{end}$.

- ordering must respect control dependencies
Schedules

Define a schedule of $g$ as any covering traversal from $n_{\text{start}}$ to $n_{\text{end}}$.

- ordering must respect control dependencies

**Definition (Schedule)**

A *schedule* of a graph $g = (n_{\text{start}}, n_{\text{end}}, E)$ is a sequence of sets of nodes $N_0, \ldots, N_k$ such that $n_{\text{start}} \not\in N_0$, $n_{\text{end}} \in N_k$, and for all $i \in [0, k)$,

- $N_i \subseteq N_{i+1}$, and
- for all $n \in N_{i+1}$, $\text{pred}(n) \subseteq N_i$. 
Every schedule corresponds to a sequence of derivations in the transition semantics.

\[
\begin{align*}
\text{If } e \Downarrow v; g; h \text{ then,} \\
\quad N_0, \ldots, N_k \text{ is a schedule of } g \\
\quad \iff \\
\quad \text{there exists a sequence of } k \text{ transitions} \\
\quad e \rightarrow \ldots \rightarrow v \text{ such that } i \in [0, k], \\
\quad \text{roots}(N_i; h) = \text{labels}(e_i).
\end{align*}
\]
Measuring Space Usage

GC roots determined by heap graph $h$ and schedule

- roots = edges that cross schedule frontier

Reachable values determined by reachability in $h$. 
Measuring Space Usage (con’t)

Note that edges in $h$ correspond to direct dependencies as well as “possible last uses.”

\[
e_1 \downarrow \text{false}^1; g_1; h_1 \quad e_3 \downarrow v_3; g_3; h_3 \quad (n \text{ fresh})
\]

if $e_1$ then $e_2$ else $e_3 \downarrow v_3; 1 \oplus g_1 \oplus [n] \oplus 1 \oplus g_3$

$h_1 \cup h_3 \cup \{(n, \ell_1)\} \cup \{(n, \ell)\} \mid \ell \in \text{labels}(e_2)$
Measuring Space Usage (con’t)

Note that edges in $h$ correspond to direct dependencies as well as “possible last uses.”

\[
\begin{align*}
e_1 &\downarrow \text{false}^{\ell_1}; g_1; h_1 & e_3 &\downarrow \nu_3; g_3; h_3 & (n \text{ fresh}) \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 &\downarrow \nu_3; 1 \oplus g_1 \oplus [n] \oplus 1 \oplus g_3 \\
h_1 \cup h_3 \cup \{(n, \ell_1)\} \cup \{(n, \ell)\} &\quad \ell \in \text{labels}(e_2)
\end{align*}
\]
Measuring Space Usage (con’t)

Note that edges in $h$ correspond to direct dependencies as well as “possible last uses.”

\[
\begin{align*}
e_1 & \downarrow \text{false}^{\ell_1} ; g_1 ; h_1 & e_3 & \downarrow \nu_3 ; g_3 ; h_3 & (n \text{ fresh}) \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \downarrow \nu_3 ; 1 \oplus g_1 \oplus [n] \oplus 1 \oplus g_3 \\
h_1 & \cup h_3 \cup \{(n, \ell_1)\} \cup \{(n, \ell)\}_{\ell \in \text{labels}(e_2)}
\end{align*}
\]
Measuring Space Usage (con’t)

Note that edges in $h$ correspond to direct dependencies as well as “possible last uses.”

\[
e_1 \Downarrow \text{false}^{\ell_1}; \ g_1; \ h_1 \quad e_3 \Downarrow \nu_3; \ g_3; \ h_3 \ (n \ \text{fresh})
\]

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow \nu_3; \ 1 \oplus g_1 \oplus [n] \oplus 1 \oplus g_3
\]

\[
h_1 \cup h_3 \cup \{(n, \ell_1)\} \cup \{(n, \ell)\}_{\ell \in \text{labels}(e_2)}
\]
Measuring Space Usage (con’t)

Note that edges in $h$ correspond to direct dependencies as well as “possible last uses.”

\[
\begin{align*}
   e_1 & \Downarrow \text{false}_{\ell_1}; g_1; h_1 \\
   e_3 & \Downarrow v_3; g_3; h_3 \quad (n \text{ fresh})
\end{align*}
\]

\[
\begin{align*}
   \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \Downarrow v_3; 1 \oplus g_1 \oplus [n] \oplus 1 \oplus g_3 \\
   h_1 \cup h_3 \cup \{(n, \ell_1)\} & \cup \{(n, \ell)\}_{\ell \in \text{labels}(e_2)}
\end{align*}
\]
Measuring Space Usage (con’t)

Note that edges in $h$ correspond to direct dependencies as well as “possible last uses.”

$$e_1 \downarrow \text{false}^{\ell_1}; g_1; h_1 \quad e_3 \downarrow v_3; g_3; h_3 \quad (n \text{ fresh})$$

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \downarrow v_3; 1 \oplus g_1 \oplus [n] \oplus 1 \oplus g_3$$

$$h_1 \cup h_3 \cup \{(n, \ell_1)\} \cup \{(n, \ell)\}_{\ell \in \text{labels}(e_2)}$$

Heap graphs have a “static” character

► necessary to simulate GC decisions
Transition semantics (above) allowed *all* possible parallel executions.

Given finite processors, which sub-expressions should be evaluated?
Scheduling Algorithms

Transition semantics (above) allowed all possible parallel executions.

Given finite processors, which sub-expressions should be evaluated?

*E.g.* depth- and breadth-first & work stealing
  - DF and BF traversals of cost graph $g$

Formalized as *deterministic* transition semantics
  - abstract presentation: no queues, &c.
Quicksort: Revisited

append <qsort (filter (le x) xs) || x::(qsort (filter (gt x) xs))>
Quicksort: Revisited

append <qsort (filter (le x) xs) || x::(qsort (filter (gt x) xs))>

16 Jan 2007
let val (ls, gs) = <filter (le x) xs ||
    filter (gt x) xs>

in
   append <qsort ls || x::(qsort gs)> 
end
let val (ls, gs) = <filter (le x) xs ||
filter (gt x) xs>
in
append <qsort ls || x::(qsort gs)> end
let val (ls, gs) = <filter (le x) xs ||
filter (gt x) xs>
in
 append <qsort ls || x::(qsort gs)> end

⇝ (via inlining)

append <qsort (filter (le x) xs) ||
x::(qsort (filter (gt x) xs))>
Greiner & Blelloch measure time and space together [ICFP ’96, TOPLAS ’99]
  ▶ upper bounds based on size and depth of DAG
Minamide shows CPS conversion preserves space usage [HOOTS ’99]
  ▶ constant overhead independent of program
Gustavsson & Sands give laws for reasoning about program transformations in Haskell [HOOTS ’99]
  ▶ formalize “safe for space” as cost semantics
Future Work

Empirical evaluation

- full-scale implementation, predict & measure performance (different GCs, schedulers)
- killer app?

Language extensions

- static discipline to help control (or at least make explicit) performance costs
- e.g. distinguish implementations of quicksort
Functional programming:
  ▶ traditionally, easy to reason about result
  ▶ ... but hard to reason about performance

In this work, we have
  ▶ related parallelism & space usage to source
  ▶ proved costs preserved by implementation
  ▶ considered effects of scheduler, collector

Ongoing: reason about performance in parallel ML