Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Justin Raymond

Professor Norman Danner

April 28, 2016

Overview

- Previous work by Danner et al. [2013] and Danner et al. [2015] formalizes the extraction of recurrences for the cost of higher-order functional programs

- Contribution:
  - We use the method of Danner et al. [2015] to analyze the complexity of higher-order functional programs

  - We extend the method to parallel cost semantics to help answer the question: how long will this program take to run on an arbitrary number of processors?

  - We prove an interesting fact about the recurrences for the cost of programs

Traditional Complexity Analysis

Source program:

```
fold f z xs =
  match xs with
    [] -> z
  | x::xs' -> f x (fold f z xs')
```

Write down recurrence for cost:

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ c_1 + T(n-1) & \text{otherwise} \end{cases}$$

Obtain a closed form solution:

$$T(n) = c_1 n + c_1 = \mathcal{O}(n)$$

Method of Danner et al. [2015]

- ▶ Write programs in a "source language"

- ▶ Translate the programs to a "complexity language"

- ▶ The translated programs are recurrences for the complexity of the source language program

- ▶ **complexity** = cost $\times$ potential

- ▶ **cost**: steps to run a program

- ▶ **potential**: size of the result of evaluating program

Source Language

- Variant of System-T
  Types:

  $$\tau ::= \text{unit} \mid \tau \times \tau \mid \tau \to \tau \mid \text{susp } \tau \mid \delta$$

  Expressions:

  $$
  \begin{aligned}
  e ::=\ & x \mid \langle \rangle \mid \lambda x.e \mid e\ e \mid \langle e, e \rangle \mid \text{split}(e, x.x.e) \\
  & \mid \text{delay}(e) \mid \text{force}(e) \mid C^\delta\ e \mid \text{rec}^\delta(e, \overline{C \mapsto x.e_C}) \\
  & \mid \text{map}^\phi(x.v, v) \mid \text{let}(e, x.e)
  \end{aligned}
  $$

- Programmer defined datatypes:

  ```
  datatype list = Nil | Cons int×list
  ```

Source Language - Cost Semantics

- Tuples:

$$\frac{e_0 \downarrow^{n_0} v_0 \qquad e_1 \downarrow^{n_1} v_1}{\langle e_0, e_1 \rangle \downarrow^{n_0+n_1} \langle v_0, v_1 \rangle}$$

- Application:

$$\frac{e_0 \downarrow^{n_0} \lambda x.e_0' \qquad e_1 \downarrow^{n_1} v_1 \qquad e_0'[v_1/x] \downarrow^{n} v}{e_0 \ e_1 \downarrow^{1+n_0+n_1+n} v}$$

- Structural Recursion:

$$\frac{e \downarrow^{n_0} C v_0 \qquad \mathtt{map}^{\phi} C \, (y.\langle y, \mathtt{delay}(rec(y, \overline{C \mapsto x.e_C}))\rangle, v_0) \downarrow^{n_1} v_1 \qquad e_C[v_1/x] \downarrow^{n_2} v}{rec(e, \overline{C \mapsto x.e_C}) \downarrow^{1+n_0+n_1+n_2} v}$$

Source Language

- OCaml:

```
length xs =
  match xs with
    [] -> 0
  | (x::xs) -> 1 + length xs
```

- Source language:

$$\lambda xs.\text{rec}(xs, \text{Nil} \mapsto 0, \text{Cons} \mapsto \langle x, \langle xs, r \rangle \rangle.1 + \text{force}(r))$$

Complexity Language

- Types

$$T ::= \mathbf{C} \mid \mathtt{unit} \mid \Delta \mid T \times T \mid T \to T$$
$$\mathbf{C} ::= 0 \mid 1 \mid 2 \mid ...$$
$$\mathtt{datatype}\Delta = C_0^\Delta \mathtt{of}\Phi_{C_0}[\Delta] \mid ... \mid C_{n-1}^\Delta \mathtt{of}\Phi_{C_{n-1}}[\Delta]$$

- Expressions

$$E ::= x \mid 0 \mid 1 \mid E + E \mid \langle\rangle \mid \langle E, E\rangle \mid$$
$$\pi_0 E \mid \pi_1 E \mid \lambda x.E \mid E\ E \mid C^\delta\ E \mid \mathrm{rec}^\Delta(E, \overline{C \mapsto x.E_C})$$

- No longer need mechanisms for controlling cost

Translation

- Translate source language programs of type τ to complexity language programs of type $\mathbf{C} \times \langle\!\langle \tau \rangle\!\rangle$

- $\mathbf{C}$ bound on the steps to evaluate the program

- $\langle\!\langle \tau \rangle\!\rangle$ expression for the size of the value

- Types of the translation function $\|\cdot\|$:

$$\|\tau\| = \mathbf{C} \times \langle\!\langle \tau \rangle\!\rangle$$
$$\langle\!\langle \mathtt{unit} \rangle\!\rangle = \mathtt{unit}$$
$$\langle\!\langle \sigma \times \tau \rangle\!\rangle = \langle\!\langle \sigma \rangle\!\rangle \times \langle\!\langle \tau \rangle\!\rangle$$
$$\langle\!\langle \sigma \to \tau \rangle\!\rangle = \langle\!\langle \sigma \rangle\!\rangle \to \|\tau\|$$
$$\langle\!\langle \mathtt{susp}\ \tau \rangle\!\rangle = \|\tau\|$$
$$\langle\!\langle \delta \rangle\!\rangle = \delta$$

Translation

- Some cases of the translation function:

$$\|x\| = \langle 0, x \rangle$$
$$\|\langle e_0, e_1 \rangle\| = \langle \|e_0\|_c + \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p \rangle \rangle$$
$$\|\lambda x.e\| = \langle 0, \lambda x.\|e\| \rangle$$
$$\|e_0 \ e_1\| = (1 + \|e_0\|_c + \|e_1\|_c) +_c \|e_0\|_p \|e_1\|_p$$

Size-Based Denotational Semantics

- Complexity translation does not lose any information
- Abstract values to their sizes using a denotational semantics
- **denotational semantics** - assign meanings to programs by interpreting types as sets and terms as elements of sets
- We use a standard denotational semantics
- Must define the following for programmer defined datatypes:
    - $\llbracket \delta \rrbracket$ - the set in which to interpret the type
    - D^{δ} - sum type representing the unfolding of the datatype
    - $size : D^{\delta} \to \llbracket \delta \rrbracket$ - function from sum type to interpretation
- The interpretation of the recursor is non-standard

$$\llbracket rec(E, \overline{C \mapsto x.E_C}) \rrbracket \xi = \bigvee_{size(z) \leq \llbracket E \rrbracket \xi} case(z, \overline{f_C})$$

Fast Reverse - Specification and Implementation

- datatype list = Nil of unit | Cons of int × list
- Specification: rev $[x_0, \ldots, x_{n-1}]$ = $[x_{n-1}, \ldots, x_0]$

  Implementation:

  ```
  rev ys =
    let go xs =
      match xs with
        [] -> fun ys -> ys
      | (x::xs') -> fun a -> (go xs') (x::a)
    in go ys []
  ```

  ```
  rev xs = foldl (fun xs x -> x::xs) [] xs
  ```

  ```
  rev = λxs.rec(xs,
    Nil ↦ λa.a,
    Cons↦ ⟨x,⟨xs,r⟩⟩.λa.force(r) Cons⟨x,a⟩))) Nil
  ```

Fast Reverse - Implementation

- `rev (Cons⟨0,Cons⟨1, Nil⟩⟩)`

- $\to_\beta$ `rec(Cons⟨0,Cons⟨1,Nil⟩⟩, Nil` $\mapsto \lambda$`a.a`
  `Cons`$\mapsto \langle x, \langle xs, r \rangle \rangle.\lambda$`a.force(r) Cons⟨x,a⟩) Nil`

- $\to_\beta^*$ $(\lambda$a0.$(\lambda$a1.$(\lambda$a2.a2$)$ Cons⟨1,a1⟩$)$ Cons⟨0,a0⟩$)$ Nil

- $\to_\beta$ $(\lambda$a1.$(\lambda$a2.a2$)$ Cons⟨1,a1⟩$)$ Cons⟨0,Nil⟩

- $\to_\beta$ $(\lambda$a2.a2$)$ Cons⟨1,Cons⟨0,Nil⟩⟩

- $\to_\beta$ Cons⟨1,Cons⟨0,Nil⟩⟩

Fast Reverse - Translation

- $\|\texttt{rev}\|$

    $\langle 0, \lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a \rangle\rangle$
    $\quad \texttt{Cons} \mapsto \langle x, \langle xs', r \rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p \texttt{Cons}\langle \pi_1 x, a \rangle\rangle) \texttt{Nil}\rangle$

Fast Reverse - Interpretation

We need to provide an interpretation for programmer-defined datatypes

$$\llbracket \texttt{list} \rrbracket = \mathbb{N}$$
$$D^{list} = \{*\} + \{1\} \times \mathbb{N}$$
$$size_{list}(*) = 1$$
$$size_{list}(1, n) = 1 + n$$

Fast Reverse - Interpretation

Interpretation of the helper function

$$g(n) = \bigvee_{size \ z \leq n} case(z, f_C, f_N)$$

where
$$f_{Nil}(x) = (1, \lambda a.(0, a))$$
$$f_{Cons}(b) = (1, \lambda a.(1 + g_c(\pi_1 b)) +_c g_p(\pi_1 b) \ (a + 1))$$

Fast Reverse - Interpretation

- Let $h(n, a) = g_p(n) \, a$.
- The recurrence for the cost:

$$h_c(n, a) = \begin{cases} 0 & n = 0 \\ 2 + h_c(n - 1, a + 1) & n > 0 \end{cases} \quad (1)$$

- $h_c(n, a) = 2n$
- The recurrence for the potential:

$$h_p(n, a) = \begin{cases} a & n = 0 \\ h_p(n - 1, a + 1) & n > 0 \end{cases} \quad (2)$$

- $h_p(n, a) = n + a$

Parametric Insertion Sort - Source Language Insert

- list datatype:
  ```
  data list = Nil of unit | Cons of int × list
  ```
- Specification:
  insert f x $[x_0, ..., x_{n-1}] = [x_0, ..., x_i, x, x_{i+1}, ..., x_{n-1}]$ where f x
  x_{i+1} = True.
- OCaml:
  ```
  insert f y xs =
    match xs with
      [] -> [y]
    | x::xs' | f y x -> y::xs
    | x::xs' -> x::insert f y xs'
  ```

- Source language:

$$\text{insert} = \lambda f.\lambda x.\lambda xs.\text{rec}(xs, \text{Nil} \mapsto \text{Cons}\langle x, \text{Nil}\rangle,$$
$$\text{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\text{rec}(f \ x \ y, \text{True} \mapsto \text{Cons}\langle x, \text{Cons}\langle y, ys\rangle\rangle,$$
$$\text{False} \mapsto \text{Cons}\langle y, \text{force}(r)\rangle))$$

Parametric Insertion Sort - Source Language Sort

- Specification:
  sort f $[x_0, ..., x_{n-1}] = [y_0, ..., y_{n-1}]$ such that
  $\forall y_i, x_{i+1}.$f y_i y_{i+1} = True and $[y_0, ..., y_{n-1}]$ is a sorted permutation
  of $[x_0, ..., x_{n-1}]$

- OCaml:
  ```
  sort f xs =
    match xs with
      [] -> []
    | x::xs' -> insert f x (sort f xs')
  ```

- Source language:

  $$\text{sort} = \lambda f.\lambda xs.\text{rec}(xs, \text{Nil} \mapsto \text{Nil},$$
  $$\text{Cons} \mapsto \langle y, \langle ys, r \rangle \rangle.\text{insert } f \ y \ \text{force}(r))$$

Parametric Insertion Sort - Complexity Language

$$\|\texttt{insert}\| = \langle 0, \lambda f.\langle 0, \lambda x.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle,$$
$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(3 + (f\ x)_c) +_c \texttt{rec}(((f\ x)_p\ y)_p,$$
$$\texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle,$$
$$\texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle))))$$

$$\|\texttt{sort}\| = \langle 0, \lambda f.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \langle 0, \texttt{Nil}\rangle,$$
$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(4 + r_c) +_c ((\|\texttt{insert}\|_p f)_p y)_p r_p)\rangle\rangle$$

Parametric Insertion Sort - Denotational Semantics

We interpret lists as a pair of their largest element and their length.

$$\llbracket \texttt{list} \rrbracket = \mathbb{Z} \times \mathbb{N}$$
$$D^{list} = \{*\} + \mathbb{Z} \times \mathbb{N}$$
$$size_{list}(*) = (0, 0)$$
$$size_{list}((i, (j, n))) = (max\{i, j\}, 1 + n)$$

Parametric Insertion Sort - Insert Interpretation

$$g(i, n) = \bigvee_{size(z) \leq (i,n)} case(z, f_{Nil}, f_{Cons})$$

where

$$f_{Nil}(*) = (1, (i, 1))$$

$$f_{Cons}(j, m) = (3 + ((f \ x)_p \ i)_c) +_c ((1, (max(x, j), 2 + m))$$
$$\vee (1 + g_c(j, m), (max(j, \pi_0 r_p), 1 + \pi_1 g_p(j, m))))$$

▶ Closed form solution for the cost:

$$g_c(i, n) \leq (4 + ((f \ x)_p \ i)_c n + 1$$

▶ Closed form solution for the potential:

$$g_p(i, n) \leq (max\{x, i\}, n + 1)$$

Parametric Insertion Sort - Sort Interpretation

$$g(i, n) = \bigvee_{size(z) \leq n} case(z, f_{Nil}, f_{Cons})$$

$$\text{where}$$

$$f_{Nil} = (1, (0, 0))$$

$$f_{Cons} = (5 + g_c(j, m) + (f\ i)_p\ j)_c g_p(j, m), (max\{i, j\}, g_p(j, m) + 1))$$

▶ Closed form solution for potential:

$$g_p(i, n) \leq (i, n)$$

▶ Closed form solution for the cost:

$$g_c(i, n) \leq (3 + ((f\ i)_p\ i)_c n^2 + 5n + 1$$

Parallel Cost Semantics

- We can reduce the running time of a program by allowing independent computations to be executed simultaneously

- Extent to which parallelism can be exploited is limited by the dependencies between subcomputations

- We add *implicit binary fork-join* parallelism to the source language

- Analysis should not be tied to a number of processors

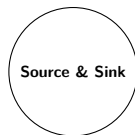- Need a new measure of cost which reflects potential parallelism opportunities

Parallel Cost Semantics

- **Cost graphs**

$$\mathcal{C} ::= 0 \mid 1 \mid \mathcal{C} \oplus \mathcal{C} \mid \mathcal{C} \otimes \mathcal{C}$$
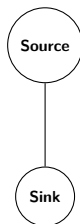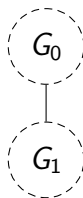
Parallel Cost Semantics

- **0** graph

Source & Sink

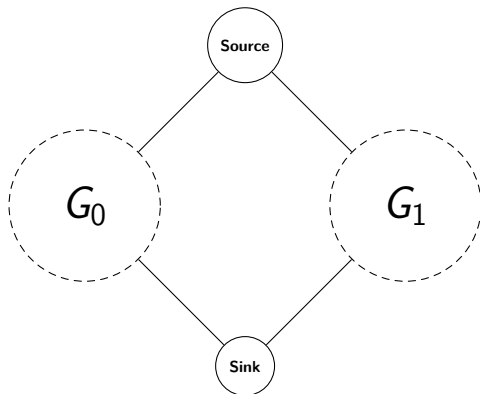# Parallel Cost Semantics

- **1** graph

Parallel Cost Semantics

- $G_0 \oplus G_1$:

Parallel Cost Semantics

- $G_0 \otimes G_1$:

Parallel Cost Semantics

- Assign cost graphs to the evaluation of expressions

- Tuples:

$$\frac{e_0 \downarrow^{n_0} v_0 \qquad e_1 \downarrow^{n_1} v_1}{\langle e_0, e_1 \rangle \downarrow^{n_0 \otimes n_1} \langle v_0, v_1 \rangle}$$

- Application:

$$\frac{e_0 \downarrow^{n_0} \lambda x.e_0' \qquad e_1 \downarrow^{n_1} v_1 \qquad e_0'[v_1/x] \downarrow^{n} v}{e_0 \ e_1 \downarrow^{(n_0 \otimes n_1) \oplus n \oplus 1} v}$$

Parallel Complexity Translation

$$\|\langle e_0, e_1 \rangle\| = \langle \|e_0\|_c \otimes \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p \rangle \rangle$$
$$\|\lambda x.e\| = \langle 0, \lambda x.\|e\| \rangle$$
$$\|e_0 \ e_1\| = 1 \oplus (\|e_0\|_c \otimes \|e_1\|_c) \oplus_c \|e_0\|_p \ \|e_1\|_p$$
$$\|delay(e)\| = \langle 0, \|e\| \rangle$$
$$\|force(e)\| = \|e\|_c \oplus_c \|e\|_p$$

Bounding Theorem: If $\gamma \vdash e : \tau$, then $e \sqsubseteq_\tau \|e\|$
proof: by logical relations.

Work and Span

- **Work** total steps to run program

$$work(c) = \begin{cases} 0 & \text{if } c = 0 \\ 1 & \text{if } c = 1 \\ work(c_0) + work(c_1) & \text{if } c = c_0 \oplus c_1 \\ work(c_0) + work(c_1) & \text{if } c = c_0 \otimes c_1 \end{cases}$$

- **Span** critical path of program

$$span(c) = \begin{cases} 0 & \text{if } c = 0 \\ 1 & \text{if } c = 1 \\ span(c_0) + span(c_1) & \text{if } c = c_0 \oplus c_1 \\ max(span(c_0), span(c_1)) & \text{if } c = c_0 \otimes c_1 \end{cases}$$

Brent's Theorem

- A program with work w and span s may be evaluated on p processors in $O(max(w/p, s))$ steps.

Parallel Tree Map

- Tree definition:

  ```
  datatype tree = E of Unit | N of int×tree×tree
  ```

- OCaml:
  ```
  map f t =
    match t with
      E -> E
    | N(x,l,r) -> N(f x, map f l, map f r)
  ```

- Source Language:

$$\texttt{map} = \lambda f.\lambda t.\texttt{rec}(t, \texttt{E} \mapsto \texttt{E},$$
$$\texttt{N} \mapsto \langle x, \langle t_0, r_0 \rangle, \langle t_1, r_1 \rangle \rangle.\texttt{N}\langle f\ x, \texttt{force}(r_0), \texttt{force}(r_1) \rangle)$$

Parallel Tree Map

- Translation:

$$\|\mathtt{map}\| = \langle 0.\lambda f.\langle 0, \lambda t.\mathtt{rec}(t_p, \mathtt{E} \mapsto \langle 1, \mathtt{E} \rangle,$$
$$\mathtt{N} \mapsto \langle y, \langle t_0, r_0 \rangle \langle t_1, r_1 \rangle \rangle.$$
$$\langle 2 \oplus (f\ x)_c \otimes r_{0c} \otimes r_{1c}, \mathtt{N} \langle (f\ x)_p, r_{0p}, r_{1p} \rangle \rangle$$

Parallel Tree Map

- Interpret trees as largest label and number of nodes:

$$\llbracket tree \rrbracket = \mathbb{Z} \times \mathbb{Z}$$
$$D_{\text{tree}} = \{*\} + \mathbb{Z} \times \llbracket \text{tree} \rrbracket \times \llbracket \text{tree} \rrbracket$$
$$size_{\text{tree}}(*) = (0, 0)$$
$$size_{\text{tree}}(x, (m_0, n_0), (m_1, n_1)) = (max(x, m_0, m_1), 1 + n_0 + n_1)$$

- Interpret cost graphs as work and span:

$$\llbracket 0 \rrbracket \xi = (0, 0)$$
$$\llbracket 1 \rrbracket \xi = (1, 1)$$
$$\llbracket c_0 \oplus c_1 \rrbracket \xi = (\llbracket c_0 \rrbracket \xi + \llbracket c_1 \rrbracket \xi, \llbracket c_0 \rrbracket \xi + \llbracket c_0 \rrbracket \xi)$$
$$\llbracket c_0 \otimes c_1 \rrbracket \xi = (\llbracket c_0 \rrbracket \xi + \llbracket c_1 \rrbracket \xi, max(\llbracket c_0 \rrbracket \xi, \llbracket c_1 \rrbracket \xi))$$

Parallel Tree Map

-
$$g(i, n) = \bigvee_{size(z) \leq (i,n)} case(z, f_E, f_N)$$

where

$$f_E(*) = [\![\langle 1, \text{E} \rangle]\!] \xi = ((1, 1), (0, 0))$$

$$f_N(j, (j_0, n_0), (j_1, n_1)) = [\![\langle 2 \oplus (f_p \ x)_c \otimes r_{0c} \otimes r_{1c}, \text{N}\langle (f_p \ x)_p, r_{0p}, r_{1p} \rangle \rangle]\!]$$

$$\{t \mapsto (i, n), f \mapsto f, x \mapsto j, r_0 \mapsto g(j_0, n_0), r_1 \mapsto g(j_1, n_1)\}$$

$$= ((2 + (f_p \ j)_c + \pi_0 g_c(j_0, n_0) + \pi_0 g_c(j_1, n_1),$$

$$2 + max((f_p \ j)_c, \pi_1 g_c(j_0, n_0), \pi_1 g_c(j_1, n_1))),$$

$$(max((f_p \ i)_p, \pi_0 g_p(j_0, n_0), \pi_1 g_p(j_1, n_1)),$$

$$1 + \pi_1 g_p(i_0, n_0) + \pi_1 g_p(i_1, n_1)))$$

Parallel Tree Map

- Work:

$$\pi_0 g_c(i, n) = (3 + (f\ i)_c)n + 1$$

- Span:

$$\pi_1 g_c(i, n) \leq 2 + (f\ i)_c + n$$

- Brent's Theorem:

$$\mathcal{O}(max\left(\frac{(f\ i)_c n}{p}, (f\ i)_c + n\right))$$

Mutual Recurrences

Pure Potential Translation

$$|\langle e_0, e_1 \rangle| = \langle |e_0|, |e_1| \rangle$$
$$|\lambda x.e| = \lambda x.|e|$$
$$|e_0 \ e_1| = |e_0| \ |e_1|$$
$$|delay(e)| = |e|$$
$$|force(e)| = |e|$$

Theorem: For all $\gamma \vdash e : \tau$, $|e| : \langle\langle \tau \rangle\rangle \sim_\tau \|e\| : \|\tau\|$
proof: by logical relations

Bibliography

Norman Danner, Jennifer Paykin, and James S. Royer. A static cost analysis for a higher-order language. In *Proceedings of the 7th workshop on Programming languages meets program verification*, pages 25–34. ACM Press, 2013. doi: 10.1145/2428116.2428123. URL `http://arxiv.org/abs/1206.3523`.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *In Proceedings of the International Conference on Functional Programming*, volume abs/1506.01949, 2015. URL `http://arxiv.org/abs/1506.01949`.