Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Justin Raymond

Professor Norman Danner

April 26, 2016

Abstract

- Complexity analysis aims to predict the resources, most often time and space, which a program requires
- Previous work by Danner et al. [2013] and Danner et al. [2015] formalizes the analysis of higher-order function programs
- We use the method of Danner et al. [2015] to analyze higher order functional programs
- We extend the method to parallel cost semantics
- We prove an interesting fact about the recurrences for the cost of programs

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Abstract

Abstract

- Complexity analysis aims to predict the resources, most often time and space, which a program requires
- Previous work by Danner et al. [2013] and Danner et al. [2015] formalizes the analysis of higher-order function programs
- We use the method of Danner et al. [2015] to analyze higher order functional programs
- We extend the method to parallel cost semantics
- We prove an interesting fact about the recurrences for the cost of programs

Complexity analysis aims to predict the resources, most often time and space, which a program requires. Traditiional complexity analysis, you look at the program, you write down a recurrence which describes the cost, you solve the recurrence, you drop everything but the highest order term. There are drawbacks to this approach. The foremost is that there is no formal relationship between the source language program and the recurrence. This makes it easier to make mistakes. The second drawback is the traditional approach is not compositional. To analyze the composition of two functions $g \circ f$ we need to know something about the size of the result of f. This gets more complicated with higher order functions such as `fold`.

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

└Abstract

Abstract

- Complexity analysis aims to predict the resources, most often time and space, which a program requires
- Previous work by Danner et al. [2013] and Danner et al. [2015] formalizes the analysis of higher-order function programs
- We use the method of Danner et al. [2015] to analyze higher order functional programs
- We extend the method to parallel cost semantics
- We prove an interesting fact about the recurrences for the cost of programs

My thesis comes in three parts. In the first part I analyze the cost of higher-order functional programs. I go through the analysis in complete detail. In the second part I demonstrate the flexibility of the formalism by changing the cost model to parallel cost semantics. In the third part I prove an interesting theorem about the recurrences extracted from source language programs.

Introduction

- 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

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

- Programmer defined datatypes
  - `datatype list = Nil | Cons int×list`
- Structural Recursion
  - OCaml:

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

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

Sequential Cost Semantics

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

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

$$\frac{}{\texttt{delay}(e) \downarrow^{0} \texttt{delay}(e)}$$

$$\frac{e \downarrow^{n_0} \texttt{delay}(e_0) \qquad e_0 \downarrow^{n_1} v}{\texttt{force}(e) \downarrow^{n_0+n_1} v}$$

Complexity Language

- Language for recurrences
- Source language without syntactic constructs for controlling costs
- No `let`, `delay`, `split`

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
- Some cases of the translation function $\| \cdot \|$:
  - $\|x\| = \langle 0, x \rangle$
  - $\|\langle e_0, e_1 \rangle\| = \langle \|e_0\|_c + \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p \rangle \rangle$
  - $\|\lambda x.e\| = \langle 0, \lambda x.\|e\| \rangle$
  - $\|e_0\ e_1\| = (1 + \|e_0\|_c + \|e_1\|_c) +_c \|e_0\|_p \|e_1\|_p$

Fast Reverse - Specification and Implementation

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

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

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

Fast Reverse - Specification and Implementation

- $\texttt{rev (Cons}\langle\texttt{0,Cons}\langle\texttt{1, Nil}\rangle\rangle\texttt{)}$
- $\rightarrow_\beta \texttt{rec(Cons}\langle\texttt{0,Cons}\langle\texttt{1,Nil}\rangle\rangle\texttt{, Nil} \mapsto \lambda\texttt{a.a}$
  $\texttt{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\lambda\texttt{a.force(r) Cons}\langle\texttt{x,a}\rangle\texttt{) Nil}$
- $\rightarrow_\beta^* (\lambda\texttt{a0.(}\lambda\texttt{a1.(}\lambda\texttt{a2.a2) Cons}\langle\texttt{1,a1}\rangle\texttt{) Cons}\langle\texttt{0,a0}\rangle\texttt{) Nil}$
- $\rightarrow_\beta (\lambda\text{a1.(}\lambda\text{a2.a2) Cons}\langle\text{1,a1}\rangle\text{) Cons}\langle\text{0,Nil}\rangle$
- $\rightarrow_\beta (\lambda\text{a2.a2) Cons}\langle\text{1,Cons}\langle\text{0,Nil}\rangle\rangle$
- $\rightarrow_\beta \text{Cons}\langle\text{1,Cons}\langle\text{0,Nil}\rangle\rangle$

Fast Reverse - Translation

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

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

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

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

Fast Reverse - Interpretation

We need to provide an interpretation for programmer-defined datatypes

$$\begin{aligned} [\![\texttt{list}]\!] &= \mathbb{N} \\ D^{list} &= \{*\} + \{1\} \times \mathbb{N} \\ size_{list}(*) &= 1 \\ size_{list}(1, n) &= 1 + n \end{aligned}$$

The interpretation of the recursor is

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

where

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

Fast Reverse - Interpretation

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

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

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

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

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

Parametric Insertion Sort - Source Language Insert

$$\texttt{data list} = \texttt{Nil of unit} \mid \texttt{Cons of int} \times \texttt{list}$$

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

```
insert f y xs =
  match xs with
    [] -> [y]
    x::xs' | f y x -> y::xs
    x::xs' -> x::insert f y xs'
```

Parametric Insertion Sort - Source Language Sort

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

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

Parametric Insertion Sort - Complexity Language

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

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

Parametric Insertion Sort - Insert Interpretation

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

where

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

$$f_{Cons}(j,m) = (4 + (f\ \ i)_c) +_c ((1,(max(x,j),2+m)) \vee (1+g_c(j,m),(max(j,\pi_0 r_p),1+\pi_1 g_p(j,m))))$$

- 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

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

where

$$f_{Nil} = (1, (-\infty, 0))$$

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

- 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

- Cost graphs

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

Evaluation Semantics

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

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

Work and Span

- **Work** total steps to run program

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

- **Span** critical path of program

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

Parallel Complexity Translation

$$
\begin{aligned}
\|\langle e_0, e_1\rangle\| &= \langle \|e_0\|_c \otimes \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p\rangle\rangle \\
\|\lambda x.e\| &= \langle 0, \lambda x.\|e\|\rangle \\
\|e_0\ e_1\| &= 1 \oplus (\|e_0\|_c \otimes \|e_1\|_c) \oplus_c \|e_0\|_p\ \|e_1\|_p \\
\|\mathit{delay}(e)\| &= \langle 0, \|e\|\rangle \\
\|\mathit{force}(e)\| &= \|e\|_c \oplus_c \|e\|_p
\end{aligned}
$$

Bounding relation: if $\gamma \vdash e : \tau$, then $e \sqsubseteq_\tau \|e\|$. Proof by logical relations.

Mutual Recurrences

Pure Potential Translation

$$
\begin{aligned}
|\langle e_0, e_1 \rangle| &= \langle |e_0|, |e_1| \rangle \\
|\lambda x.e| &= \lambda x.|e| \\
|e_0\ e_1| &= |e_0|\ |e_1| \\
|\mathit{delay}(e)| &= |e| \\
|\mathit{force}(e)| &= |e|
\end{aligned}
$$

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`.