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

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
```

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

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

- $\texttt{datatype list} = \texttt{Nil of unit} \mid \texttt{Cons of int} \times \texttt{list}$
- Specification: $\texttt{rev } [x_0, \ldots, x_{n-1}] = [x_{n-1}, \ldots, x_0]$
- Implementation:
  $\texttt{rev xs} = \lambda\texttt{xs.rec(xs,}$
  $\texttt{Nil} \mapsto \lambda\texttt{a.a,}$
  $\texttt{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\lambda\texttt{a.force(r) Cons}\langle\texttt{x,a}\rangle))) \texttt{ Nil}$
- Specification of auxilary function:
  $\texttt{rec}([x_0, \ldots, x_{n-1}], \ldots) \ [y_0, \ldots, y_{m-1}] =$
  $[x_{n-1}, \ldots, x_0, y_0, \ldots, y_{m-1}]$

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

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

Sequential Cost Semantics

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

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

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

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

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

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

Pure Potential Translation

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`.