Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Justin Raymond

Professor Norman Danner

April 27, 2016

Overview

- 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

Overview

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`.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.

Complexity Analysis

Source program

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

Recurrence for cost

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

Closed form solution

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Complexity Analysis

Complexity Analysis

Source program

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

Recurrence for cost

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

Closed form solution

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

Traditional complexity analysis of a recursive program, we write down a recurrence for the cost of the program. Then we solve the recurrence and drop the constant factors. There is no formal relation between the program and the recurrence. Can we make this analysis higher-order? How do we take into account the asymptotic complexity of f?
We could try to say that f is $\mathcal{O}(x)$ where x is the size of the sum of its arguments. But how big are the sum of the arguments to f?
The size of the inputs to f depend on the size of each element in the list and the size of recursive calls to `fold`.

How to address this? Enter the Danner et al. [2015].

Overview

- 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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Overview

Overview

- 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 × potential
- **cost**: steps to run a program
- **potential**: size of the result of evaluating program

How do anaylsis in a way that we have a formal connection between the program and the recurrence, is naturally higher-order, and compositional?

Previous work by Danner et al. [2013] and Danner et al. [2015] developed a method to address this. The overview of the approach is the programmer writes their programs in a source language. Then the source language program is translated to whats called a complexity language. The complexity language is essentially a language for expressing the recurrence for the complexity of the source language program. What is a complexity? A complexity is a pair of a cost and a potential. The cost is a bound on the steps required to run the program. The potential is the size of the result of evaluating the 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
```

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Source Language

Source Language

- Variant of System-T

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

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

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

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

So what does the source language look like? It is a variant of System T. If you're like me and can never remember the differences between the variants on the lambda calculus, System T is the simply typed lambda calculus with primitive recursion. The source language as structural recursion as well as some additional mechanisms to control the cost of programs. So we have `let` expressions to avoid recomputations of values and suspensions to evaluating expressions we don't actually need.
The source language also has programmer-defined datatypes. Here is an example of how you would define a list. The arguments to the constructor of a datatype must be strictly positive.

As an example of what the source language program looks like. Here is a recursive function that computes the length of a list in OCaml and the same function in the source language. The `rec` construct gives us structural recursion. It evaluates an expression to a value, and based on the constructor of the value, evaluates to the appropriate branch. Inside the branch we are given access to the arguments to the constructor as well as a delayed computation representing the result of the recursive call.

Sequential Cost Semantics

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

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

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

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Sequential Cost Semantics

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

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

$$\frac{}{\texttt{delay}(e) \downarrow^{0} \texttt{delay}(e)} \qquad \frac{e \downarrow^{n_0} \texttt{delay}(e_0) \qquad e_0 \downarrow^{n_1} v}{\texttt{force}(e) \downarrow^{n_0+n_1} v}$$

The semantics of the source language are given by operation cost semantics. Operational cost semantics are big step operational semantics but include a notion of steps to execute the program. The judgments are of the form e steps to v in n steps.

Complexity Language

- Source language without syntactic constructs for controlling costs
- Types

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

- Expressions

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

- No longer need mechanisms for controlling cost

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Complexity Language

Complexity Language

- Source language without syntactic constructs for controlling costs
- Types

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

- Expressions

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

- No longer need mechanisms for controlling cost

The complexity language is a language for recurrences for the cost and potential of a source language program. We add an additional type **C** for costs. The rest of the language is very similar to the source language except we no longer need the syntactic constructs for controlling the costs. So the complexity language does not have suspensions or let expressions.

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 $\| \cdot \|$:

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Translation

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 $\|\cdot\|$:

$$\begin{aligned}
\|\tau\| &= \mathbf{C} \times \langle\langle \tau \rangle\rangle \\
\langle\langle \mathtt{unit} \rangle\rangle &= \mathtt{unit} \\
\langle\langle \sigma \times \tau \rangle\rangle &= \langle\langle \sigma \rangle\rangle \times \langle\langle \tau \rangle\rangle \\
\langle\langle \sigma \to \tau \rangle\rangle &= \langle\langle \sigma \rangle\rangle \to \|\tau\| \\
\langle\langle \mathtt{susp}\ \tau \rangle\rangle &= \|\tau\| \\
\langle\langle \delta \rangle\rangle &= \delta
\end{aligned}$$

The translation function cross compiles source language programs to complexity language programs. So a source language program of type τ is translated to a complexity language program of type $\mathbf{C} \times \langle\langle \tau \rangle\rangle$. The $\mathbf{C}$ component is a bound on the steps to evaluate the program and $\langle\langle \tau \rangle\rangle$ is a complexity language expression for the size of the result of executing the program.

Abstractions are less straightforward. The cost of evaluating an abstraction is 0, since abstractions are in normal form. The potential of the translation of an abstraction is a function from potentials to complexities. Recall that a potential captures the cost of future uses of an expression. The cost of future uses of a function depends on the size of inputs you apply it to. So the potential of an abstraction is a function whose argument is a potential. Since applying a function has both a cost and a result. The codomain of the potential of the abstraction translation is a cost and a potential, aka a complexity.

Translation

Some cases of the translation function

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Translation

Some cases of the translation function

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

Here are some cases of the translation function to get a better idea of what is going on. In the variable case, the result of the translation is $\langle\langle 0, x\rangle\rangle$. The cost of evaluating the variable is 0 as it is already in normal form. The potential is the variable itself. The cost of translating a pair is the sum of the costs of translating the elements of the tuple. The potential is the pair of the potentials of the translations of the expressions.

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 = 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 []

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Fast Reverse - Specification and Implementation

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 = 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 []

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

I'll go through two examples. The first is a higher-order function. The function reverses a list but uses a higher-order function to do so in linear time. Here is the implementation in OCaml. You'll notice that this is in fact a higher-order fold, and that we can write the function using a fold like so.

Fast Reverse - Specification and Implementation

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Fast Reverse - Specification and Implementation

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

To give you an intuition into how this works, we the recursion results in nested functions. Each function takes a list of an argument and conses and element of to the list. The outermost function conses the first element of the original list and the innermost function conses the last element. Since the outermost function is evaluated first, the first element gets consed onto an empty list first and the last element gets consed on last, resulting in a reversed list. (DRAW PICTURE).

Fast Reverse - Translation

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

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

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

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Fast Reverse - Translation

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

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

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

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

This is the result of the translation into the complexity language. The first term is the complexity of the function `rev` itself. The cost is 0 and the potential is a function from potentials to complexities. We are interested in the analysis of applying the function `rev` to some list, so the translation of `rev xs` also shown.

Fast Reverse - Interpretation

We need to provide an interpretation for programmer-defined datatypes

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Fast Reverse - Interpretation

Fast Reverse - Interpretation

We need to provide an interpretation for programmer-defined datatypes

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

The translation of a term to the complexity language does not throw away any information. That is, there is no abstractions about the list. To do so we interpret the complexity language expression in a denotational semantics. Denotational semantics assign meanings to programs by interpreting types as sets and terms as elements of the set corrresponding to their type. Abstractions of type $\tau \to \sigma$ are interpreted as mathematical functions with domain τ and codomain σ. We use standard denotational semantics. The exception is programmer-defined datatypes. There are multiple interpretations of the size of a datatype. For example, if our natural numbers are fixed width integers we may want to interpret all natural numbers as having the same size. We may also interpret natural numbers as an inductively defined datatypes, where the interpretation of a natural number is the number itself. Another example is trees. We may want to interpret trees as their number of nodes. We may also want to interpret trees as their number of nodes and the maximum size of their label. Another possible interpretation is the height of the tree.

Here we interpret lists as their lengths. We need D because we need to be able to branch on a datatype in the

denotational semantics, so we introduce the sum type D representing the unfolding of the constructors.

Fast Reverse - Interpretation

Interpretation of the recursor

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

where

$$f_{Nil}(x) = (1, \lambda a.(0, a))$$

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

Extracting Cost Recurrences from Sequential and Parallel Functional Programs

Fast Reverse - Interpretation

Fast Reverse - Interpretation

Interpretation of the recursor

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

where

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

The interpretation of a structural recursion is a maximum over a *case* expression.

This is a recurrence for the function. We want the recurrence for the complexity of the function applied to some value.

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:

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

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

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:

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

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

So we let h be the result of applying g to the interpretation of some list a. In order to make the analysis easier we break the complexity recurrence into a recurrence for the cost and a recurrence for the potential. Then we can solve the recurrences to get closed form solutions. So the conclusion is the cost is linear in the length of the list and the potential is the sum of the two lists.

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`.