Extraction of Cost Recurrences from Sequential and Parallel Functional Programs

Justin Raymond
Wesleyan University

Abstract

Complexity analysis aims to predict the resources, most often time and space, which a program requires. Traditional complexity analysis is not compositional, often does not address higher order functions, and there is no formal relation between programs and the equations that describe their complexity. We build on previous work by [2] and [1] which formalizes the extraction of cost recurrences from higher order functional programs. We use the formalization to analyze the time complexity of higher order functional programs. We also demonstrate the flexibility of the method by extending it to parallel cost semantics.

Complexity Analysis

The higher order function `fold` reduces a list to a single element using a combining function.

$$\texttt{fold} = \lambda f\ z\ xs.\texttt{rec}(xs, \texttt{Nil} \mapsto z,$$
$$\texttt{Cons} \mapsto (x, xs', r).f\ x\ \texttt{force}(r))$$

To analyze the complexity of `fold`, we write down a recurrence $T(n)$ which reflects the number of steps to execute the `fold` applied to a function f, a seed value z, and a list xs.

$$T(n) = c_f + T(n-1)$$

The closed form solution is $T(n) = c_f\ n$. The method we used to write the recurrence for the program was informal. The analysis is not composable: we cannot use our results to analyze `map f ∘ fold g z`. The analysis also assumes the cost of applying f to each item in the list is constant.

Higher Order Complexity Analysis

Instead of considering the complexity of a program as a cost, we consider it as a pair of a cost and a potential. The cost is a bound on the steps required to run the program. The potential is a measure of the cost of future use of the program.

$$\|\tau\| = \mathbf{C} \times \langle\langle \tau \rangle\rangle$$

Fig. 1: The type of the translation function.

To formally extract a recurrence from a program, we translate it from its original "source" language into a "complexity" language. We then interpret the complexity language program in a denotational semantics to obtain the recurrence.

$$\|\langle e_0, e_1 \rangle\| = \langle \|e_0\|_c + \|e_1\|_c, \langle \|e_0\|_p \|e_1\|_p \rangle \rangle$$

Fig. 2: The translation of a tuple.

This is a formal process for extracting a recurrence for the cost of executing a program, and the recurrence is a bound on the cost of executing the program.
Since the analysis of a program results in a measure of future use of the program, we can compose analysis of programs to analyze the composition of the programs as well analyze higher-order programs.

References

[1] 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*. Vol. abs/1506.01949. 2015.

[2] 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*. ACM Press, 2013, pp. 25–34. DOI: 10.1145/2428116.2428123.

Parallel Complexity Analysis

To analyze the complexity of a parallel program we use a specification that is agnostic to the details of how to schedule subcomputations, that is the analysis does not depend on the number of processors available. Instead of representing costs as natural numbers, we represent a cost as a graph which describes the dependencies between subcomputations of the program.

$$\mathbf{C} = 0 \mid 1 \mid \mathbf{C} \oplus \mathbf{C} \mid \mathbf{C} \otimes \mathbf{C}$$

Fig. 3: Cost graph definition

Two measures can be extracted from the cost graph: work, the total amount of steps to execute a program, and span, the length of the critical path of the program execution. The predicted running time of a program with work w and span s on p processors is given by Brent's Theorem:

$$O(max(\frac{w}{p}, s))$$

The translation rules must be altered to produce cost graphs instead of natural numbers.

$$\|\langle e_0, e_1 \rangle\| = \langle \|e_0\|_c \otimes \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p \rangle \rangle$$

Fig. 4: Parallel Translation of a tuple

The steps to execute the source language program is bounded by the recurrence in the translated complexity language program.

$$\text{If } \gamma \vdash e : \tau, \text{ then } e \sqsubseteq \|e\|$$

Fig. 5: Bounding Theorem

$\sqsubseteq$ is a binary logical relation stating the cost of executing the program e is bounded by the cost of $\|e\|$.