Data structure inference based on source code

Aleksander Balicki

September 6, 2011

Abstract

Contents

1 Introduction

1.1 Example imperative language

We define an imperative language, on which we will show examples of the algorithm.

2 Data structure inference

2.1 Comparison of the complexities

Asymptotical complexity of an operation we store as a pair of type:

$$AsymptoticalComplexity = Int \times Int, \tag{1}$$

where

$$(k,\ l)\ means\ O(n^k \log^l n). \tag{2}$$

The reason to choose such a type is that it's easier to compare than the general case (we can do a lexicographical comparison of the two numbers) and it distincts most of the data structure operation complexities.

Sometimes we have to use some qualified complexities:

$$ComplexityType = \{Normal,\ Amortized,\ Amortized\ Expected,\ Expected\} \tag{3}$$

The overall complexity can be seen as a type:

$$Complexity = AsymptoticalComplexity \times ComplexityType \tag{4}$$

Here we can also use a lexicographical comparison, but we have to say that

$$Normal > Amortized, \tag{5}$$
$$Amortized > Expected, \tag{6}$$
$$Expected > Amortized\ Expected, \tag{7}$$
$$\tag{8}$$

and that $>$ is transitive.

We also always choose the smallest asymptotic-complexity-wise complexity. For example, we have a search operation on a splay tree. It's $O(n)$, but $O(\log n)$ amortized, so it's represented as $((0, 1), Amortized)$.

2.2 Choosing the best data structure

We define a set $DataStructureOperations$. We can further extend this set, but for now assume that

$$DataStructureOperations = \{Insert,\ Update,\ Delete,\ FindMax,\ DeleteMax,\ \ldots\}. \tag{9}$$

Each of the $DataStructureOperations$ elements symbolizes an operation you can accomplish on a data structure.

The type

$$DataStructure \subset DataStructureOperations \times Complexity \tag{10}$$

represents a data structure and all of the implemented operations for it, with their complexities.

When trying to find the best suited data structure for a given program P, we look for data structure uses in P. Let $DSU(P)$ be the set of $DataStructureOperations$ elements, that are used somewhere in the source code of P.

We define a parametrized comparison operator for data structures $<_{DSU(P)}$ defined as:

$$d_1 <_{DSU(P)} d_2 \Leftrightarrow o \in DSU(P) \wedge \tag{11}$$

$$|\{(o, c_1) \in d_1 | (o, c_2) \in d_2 \wedge c_1 < c_2\}| < \{(o, c_2) \in d_2 | (o, c_1) \in d_1 \wedge c_2 < c_1\} \tag{12}$$

If we fix P, we have a preorder on data structures induced by $<_{DSU(P)}$ and we can sort those data structures using this order. The maximum element is the best data structure for the task.

3 Extensions of the idea

3.1 Second extremal element

If we want to find the maximal element in a heap, we just look it up in $O(1)$, that's what heaps are for. If we want to find the minimal element we can use a min-heap. What happens if we want to find the max and the min element in one program? How to modify our framework to handle this kind of situations?

$$DataStructureOperations = \{\ldots\ FindFirstExtremalElement,\ DeleteFirstExtremalElement, FindSecond \tag{13}$$

Now we can add two complexity costs to the data structure definition, the cheaper one can be used primarily, and the second one can be used in above situations.

3.2 Big load

change in the algorithm

3.3 Data structure modifications

max elem cache

3.4 Different element types

Currently the framework works only for integer elements. We can extend it to each type that has a compare function, because there's no difference if the types are numerical or not. If we analyzed haskell, we would use types that are in the Ord class, if we analyzed C, we would ask the programmer to write an acompanying cmp function to the type and pass the pointer to the function that creates the data structure.

3.5 Linked data structures

If we wanted to keep records (structs) in our data structure and find an element by one field and some other time by some other field, we would have to do this:

3.6 Transforming datastructures on-line

what it said

3.7 Upper bound on the element count

so we can choose between malloc and static allocation

3.8 Outer-world input

detecting scanf and sockets and so on

3.9 Minimal element count treshold

4 Program

4.1 Recommendation mode

prints recommendations

4.2 Advice mode

prints advice

4.3 Compile mode

linkes appropriate lib

4.4 Typechecker