Optimal Code Generation for Expression Trees using Haskell

Ganesh Bhambarkar - 153050072
Sushant Shambharkar - 153050081

November 2015

1 Introduction

This project involves generation of optimal code for the machine using Aho-Johnson algorithm given an expression tree and a machine model. Bruno and Sethi show that generating optimal code is difficult, even if the target machine has only one register; specifically they show that the problem of generating optimal code for straight-line sequence of assignment statements is NP-complete.[2]

On the other hand , if we restrict the class of inputs to straight line programs with no common sub-expression, optimal code generation becomes considerably easier.[2]

2 Problem Definition

Input to the algorithm is a machine model and expression tree. Machine model contains number of registers and a set of instructions that machine supports. Expression tree is a sequence of assignment statements represented as tree.

3 Algorithm

The Aho-Johnson algorithm produces machine code for a given expression tree with minimum cost. The algorithm is from dynamic programming class of algorithms and uses memoization to store the previously computed results, which might be useful in future.

Width : The width of a program is the maximum number of registers live at any instruction. A program of width w (but possibly using more than w registers) can always be rearranged into an equivalent program using exactly w registers.[3]

Strongly Contiguous : A program is strongly contiguous if width of that program is minimum.

Strong Normal Form : A program $P(P_1 J_1 P_2 J_2 ... P_{S-1} J_{S-1} P_S)$ is in strong normal form, if each P_i is strongly contiguous. J_i are store instructions.[3]

The algorithm is divided into three phases.

3.1 Phase 1 : Computing Cost Array for each subtree

Given an expression tree (T), a set of instructions (I) and number of registers (n), this phase calculates the cost array, optimal instruction and optimal permutation for each node in T. *Phase 1* uses a subroutine *cover(E,S)* and calculates cost array, optimal instructions and optimal permutations.

Number of registers: 2

Memory sequences: $m1, m2, m3 \dots$

Instructions:
$m \leftarrow r_1$ (Cost 1)
$r_1 \leftarrow m$ (Cost 1)
$r_1 \leftarrow c$ (Cost 1)
$r_1 \leftarrow r_1 + r_2$ (Cost 2)
$r_1 \leftarrow r_1 * r_2$ (Cost 2)
$r_1 \leftarrow ind(r_1)$ (Cost 1)
$r_1 \leftarrow ind(r_1 + m)$ (Cost 4)

Figure 1: Machine model

cover(E,S) : Given an instruction and a node in an expression tree, *cover(E,S)* returns True or False depending on whether the instruction(E) covers the node(S) or not. If the instruction(E) covers the node(S) then cover will place into two (initially empty) sets, *regset* and *memset*, the subtrees which need to be computed in registers and memory, respectively.

Cost Array : This is an array of size $n + 1$ at each node.

- $C_j(S), j \neq 0$: is the minimum cost of evaluating S with a strong normal form program using j registers.
- $C_0(S)$: cost of evaluating S with strong normal form program in a memory location.[3]

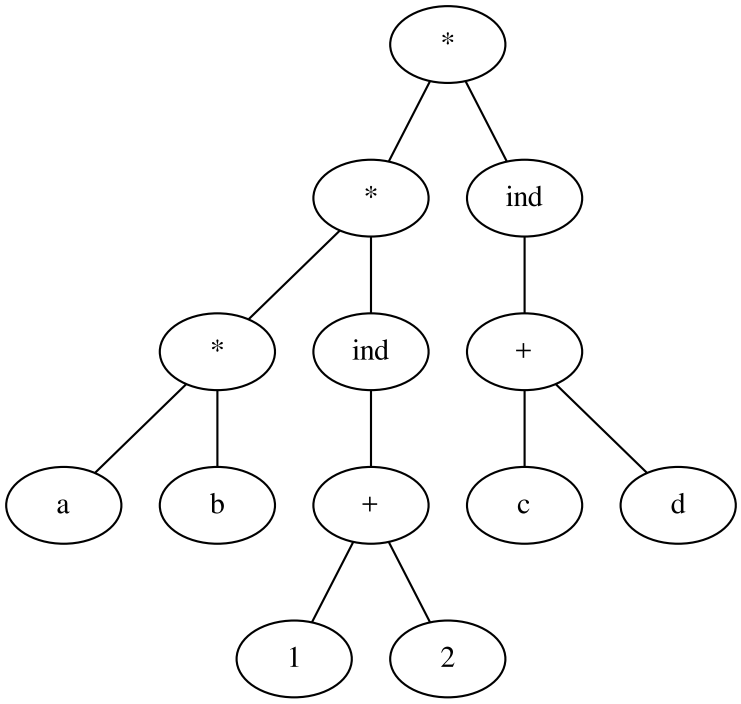

Figure 2: Input expression tree

Optimal Instruction : From all the instructions which cover this node, optimal instruction gives the minimum cost. This is defined for each C_j.

Optimal Permutation : The order in which the regset of the optimal instruction should be evaluated so that the cost will be optimum. This is defined for each C_j.

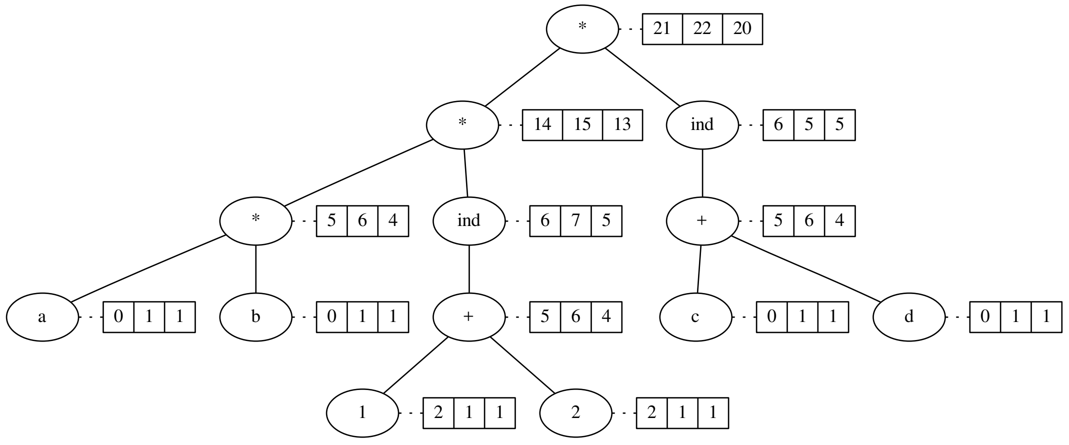

Figure 3: Output of phase 1 (Showing cost arrays only)

3.2 Phase 2 : Determining the subtrees to be stored

Phase 2 walks over the tree T, making use of the $C_j(S)$ arrays computed in Phase 1 to create a sequence of vertices $x_i, \dots, x_s$ of T. These vertices are the roots of those subtrees of T which must be computed into memory; x_s is the root of T. These subtrees are so ordered that for all i, x_i never dominates any x_j for $j > i$. Thus if one subtree S requires some of its subtrees to be computed into memory, these subtrees will be evaluated before S is evaluated. *Phase 2* uses *mark(T,n)*.

mark(E,S) : Given a expression tree with costs (T) and number of registers(n), *mark(T,n)* returns list of nodes in the given tree which need to be stored in memory for optimal cost.

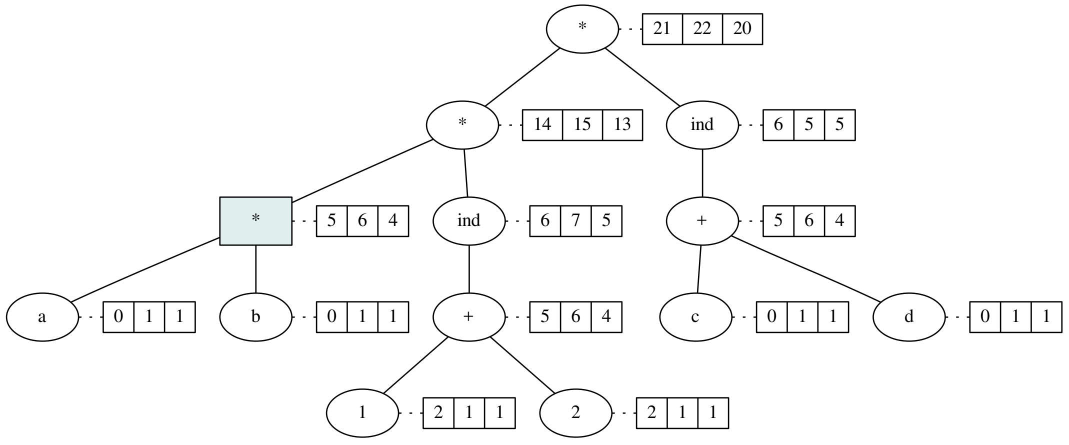

Figure 4: Phase 2 output

3.3 Phase 3 : Code Generation

This phase makes a series of walks over subtrees of T, generating code. These walks start at the vertices $x_i, \ldots, x_s$ computed in Phase 2. After each walk, we generate a store into a distinct temporary memory location m_i, and rewrite node x_i, to make it behave like an input variable m_i in later walks that might encounter it. *Phase 3* uses *code(S,j)* to generate code.

code(S,j) : Given a node(S) in expression tree which need to be stored in memory and number of registers(j), *code(S,j)* returns sequence of instruction which results in optimal cost for give node(S).

Generated code:

```
r1 ← b
r2 ← a
r2 ← (r2 * r1)
m1 ← r2
r1 ← 2
r2 ← 1
r2 ← (r2 + r1)
r2 ← (ind r2)
r1 ← m1
r1 ← (r1 * r2)
r2 ← c
r2 ← (ind (r2 + d))
r1 ← (r1 * r2)
```

4 Implementation

In this section we will describe how we implemented the Aho-Johnson algorithm in Haskell.

The final distributable after compilation is an executable called `Main`.

The usage of the program is:

`Main <number-of-registers> <instructions-file> <expression-tree-file> <output-file>`

4.1 Definitions

```haskell
type RegisterName = Int
type MemoryName = String
type Op = String
type Cost = Int

type RegSet = [Walk]
type MemSet = [Walk]

type Walk = [Int]

data Annotation = Annotation
    { costs :: [Cost]
    , instructions :: [Instruction]
    , permuts :: [[Int]]
    } deriving (Show, Read)

data Expr = Fork Op [Expr]
          | Register RegisterName
          | Constant Int
          | Memory MemoryName deriving (Show, Read, Eq)

data AnnotatedExpr = ForkAnnotated Op [AnnotatedExpr] Annotation
                   | RegisterAnnotated RegisterName Annotation
                   | ConstantAnnotated Int Annotation
                   | MemoryAnnotated MemoryName Annotation deriving
     (Show, Read)
```

```haskell

data Instruction = RegisterAssign RegisterName Expr Cost
                 | Load RegisterName MemoryName Cost
                 | LoadConstant RegisterName Int Cost
                 | Store MemoryName RegisterName Cost
                 | NoInstruction deriving (Read, Eq)
```

4.1.1 Sample instructions

Instruction	Haskell representation
$m \leftarrow r_1$ (Cost 4)	`Store "m" 1 4`
$r_1 \leftarrow m$ (Cost 4)	`Load 1 "m" 4`
$r_1 \leftarrow 2$ (Cost 5)	`LoadConstant 1 2 5`
$r_1 \leftarrow r_1 + r_2$ (Cost 4)	`RegisterAssign 1 (Fork "+" [Register 1, Register 2]) 4`
$r_1 \leftarrow ind(r_1 + m)$ (Cost 3)	`RegisterAssign 1 (Fork "ind" [Fork "+" [Register 1, Memory "m"]]) 3`

4.1.2 Sample expression

```haskell
Fork "*" [Fork "*" [Fork "*" [Memory "a", Memory "b"], Fork "ind" [
    Fork "+" [Constant 1, Constant 2]]], Fork "ind" [Fork "+" [
    Memory "c", Memory "d"]]]
```

4.2 Algorithm

```
instructions ← read the provided instructions file;
expressionTree ← read the expression tree from provided file;
n ← number of registers;
verify the number of registers;
verify instruction set;
verify expression tree;
generate PDF file for expressionTree using GraphViz[1] tool (dot
command);
initTree ← initAnnotatedExprTree n expressionTree;
costTree ← phase1 n instructions initTree;
generate PDF file for costTree;
markedNodes ← phase2 n costTree;
generate PDF file for costTree showing marked nodes;
generatedCode ← phase3 numberOfRegisters costTree marked;
```

Algorithm 1: Main flow of the implementation

4.3 Important functions

```haskell
cover :: Instruction -> Expr -> (Bool, RegSet, MemSet)

initAnnotatedExprTree :: Int -> Expr -> AnnotatedExpr

--Call phase1AtNode recursively in preorder
phase1 :: Int -> [Instruction] -> AnnotatedExpr -> AnnotatedExpr
```

```haskell

--Calculate costs at this node only, no recursion
phase1AtNode :: Int -> [Instruction] -> AnnotatedExpr -> AnnotatedExpr

phase2 :: Int -> AnnotatedExpr -> [Walk]
phase2 n tree = mark n n [] [] tree

--Recursive implementation of mark
mark :: Int -> Int -> Walk -> [Walk] -> AnnotatedExpr -> [Walk]

phase3 :: Int -> AnnotatedExpr -> [Walk] -> [Instruction]

--Called in phase3 for each marked node
code :: Walk -> Int -> AnnotatedExpr -> [RegisterName] -> [Instruction] -> (RegisterName, [Instruction], [RegisterName])
```

5 Conclusion

Implementing recursive algorithms in Haskell is really easy, and this algorithm is no exception. Tracking down bugs during development was really fast. As opposed to imperative languages, where due to side effects, bugs can occur anywhere (outside the function which actually has the bug).

- The time required by Phase 1 is an, where a is a constant depending
  1. linearly on the size of the instruction set
  2. exponentially on the arity of the machine, and
  3. linearly on the number of registers in the machine
- Time required by Phase 2 and 3 is proportional to n.

References

[1] Graphviz. http://www.graphviz.org.

[2] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, July 1976.

[3] Amitabha Sanyal. Code generation. https://www.cse.iitb.ac.in/~uday/courses/cs324-08/code-generation.pdf.