50006 - Compilers - (Prof Kelly) Lecture 4

Oliver Killane

11/01/22

Lecture Recording

Lecture recording is available here

Unbounded Register Use

We will generate code for arithmetic expressions that:

- Assumes there will always be enough registers.
- Handles the case when we run out of registers.
- Translates expressions while minimising number of registers required.

```haskell
data Instruction
   = Add Reg Reg | Sub Reg Reg
     | Mul Reg Reg | Div Reg Reg -- Op r1 r2 -> r1 := r1 <Op> r2
     | AddImm Reg Int | SubImm Reg Int
     | MulImm Reg Int | DivImm Reg Int -- <Op>Imm r c -> r := r <Op> c
     | Load Reg Name      -- Load r1 n     -> r1 := mem[n]
     | LoadImm Reg Int    -- LoadImm r1 i -> r1 := i
     | Store Reg Name     -- Load r1 n     -> mem[n] := r1
     | Push Reg           -- Push r1       -> SP++; mem[SP] := r1
     | Pop Reg            -- Pop r2        -> r1 := mem[SP]; SP--
     | CompEq Reg Reg     -- CompEq r1 r2 -> r1 := r1 - r2 = 0 ? 1 : 0
     | JTrue Reg Label    -- JTrue r1 l    -> IF r1 = 1 THEN JUMP TO l
     | JFalse Reg Label   -- JFalse r1 l   -> IF r1 = 0 THEN JUMP TO l
     | Define Label       -- Assembler directive to set up label
```

We can take an input such as:

$$(100 * 3) + ((200 * 2) + 300) + (400 + (500 * 3))$$

	Instruction	Stack Slot			
		3	2	1	0
0	PushImm 100				100
1	PushImm 3			3	100
2	Mul			3	300
3	PushImm 200			200	300
4	PushImm 2		2	200	300
5	Mul		2	400	300
6	PushImm 300		300	400	300
7	Add		300	700	300
8	Add		300	700	1000
9	PushImm 400		300	400	1000
10	PushImm 500		500	400	1000
11	PushImm 3	3	500	400	1000
12	Mul	3	1500	400	1000
13	Add	3	1500	1900	1000
14	Add	3	1500	1900	2900

We can use the placement of values in the stack (relative to the initial stack pointer) to assign registers. Assigning each slot to a register.

We want to provide the translator with the register to place the result in, it can use any higher registers.

	Instruction	Register			
		R3	R2	R1	R0
0	LoadImm R0 100				100
1	LoadImm R1 3			3	100
2	Mul R0 R1			3	300
3	LoadImm R1 200			200	300
4	LoadImm R2 2		2	200	300
5	Mul R1 R2		2	400	300
6	LoadImm R2 300		300	400	300
7	Add R1 R2		300	700	300
8	Add R0 R1		300	700	1000
9	LoadImm R1 400		300	400	1000
10	LoadImm R2 500		500	400	1000
11	LoadImm R3 3	3	500	400	1000
12	Mul R2 R3	3	1500	400	1000
13	Add R1 R2	3	1500	1900	1000
14	Add R0 R1	3	1500	1900	2900

Register Improvements

We could improve our generated code if there were instructions available for in place constant application.

IA32 with Immediate Operand

```
movl $3, %eax
imull $3, %eax
addl $4, %eax
```

Our Simple Assembly

```
PushImm R0 3
PushImm R1 3
Mul R0 R1
PushImm R1 4
Add R0 R1
```

	Instruction	Register			
		R3	R2	R1	R0
0	LoadImm 0 3				3
1	MulImm 0 100				300
3	LoadImm 1 2			2	300
4	MulImm 1 200			400	300
5	AddImm 1 300			700	300
6	Add 0 1			700	1000
7	LoadImm 1 3			3	1000
8	MulImm 1 500			1500	1000
9	AddImm 1 400			1900	1000
10	Add 0 1			1900	2900

Code for Translation

```haskell
translateOp :: Op -> (Int -> Int -> Instruction)
translateOp Plus = Add
translateOp Minus = Sub
translateOp Times = Mul
translateOp Divide = Div

translateOpImm :: Op -> (Int -> Int -> Instruction)
translateOpImm Plus = AddImm
translateOpImm Minus = SubImm
translateOpImm Times = MulImm
translateOpImm Divide = DivImm

transExp :: Exp -> Reg -> [Instruction]
transExp (Const n) r = [LoadImm r n]
transExp (Ident id) r = [Load r id]

-- Only allow for - unary operato (e.g -3)
transExp (Unop Minus e) r = transExp e r ++ [MulImm r (-1)]
transExp (Unop _ _) _
  = error "(transExp) Only '-' unary operator supported"

-- As * and + are commutative, the order does not matter
transExp (BinOp Times (Const n) e) r = transExp e r ++ [MulImm r n]
transExp (BinOp Plus (Const n) e) r = transExp e r ++ [AddImm r n]

-- Can run left hand, then do right hand with immediate operand
transExp (BinOp op e (Const n)) r = transExp e r ++ [translateOpImm op r n]

-- General case for two expressions
transExp (BinOp op e1 e2) r
  = transExp e1 r ++ transExp e2 (r+1) ++ [translateOp op r (r+1)]
```

Bounded Number of Registers

Accumulator Machine

Has a single register (**Accumulator**) upon which arithmetic instructions can be applied.

```haskell
data Instruction = Add | Sub | Mul | Div --
  | AddImm Int | SubImm Int | MulImm Int | DivImm Int    --
  | CompEq          -- CompEq   -> Acc :=
  | Push            -- Push     -> SP--; mem[SP] := Acc
  | Pop             -- Pop      -> Acc := mem[SP]; SP++
  | Load Name       -- Load n   -> Acc := mem[n]
  | LoadImm Int     -- Load i   -> Acc := i
  | Store Name      -- Store n  -> mem[n] := Acc
  | Jump Label      -- Jump l   -> PC := l
  | JTrue Label     -- JTrue l  -> IF Acc = 1 THEN JUMP TO l
  | JFalse Label    -- JFalse l -> IF Acc = 0 THEN JUMP TO l
  | Define Label    -- Assembler directive to set up label
```

	Instruction	Acc	Stack 1	Stack 0
0	LoadImm 500	500		
1	MulImm 3	1500		
3	AddImm 400	1900		
4	Push	1900		1900
5	LoadImm 200	200		1900
6	MulImm 2	400		1900
7	AddImm 300	700		1900
8	Push	700	700	1900
9	LoadImm 100	100	700	1900
10	MulImm 3	300	700	1900
11	Add	1000	700	1900
12	Add	2900	700	1900

```haskell
transOpImm :: Op -> (Int -> Instruction)
transOpImm Plus = AddImm
transOpImm Minus = SubImm
transOpImm Times = MulImm
transOpImm Divide = DivImm

transOp :: Op -> Instruction
transOp Plus = Add
transOp Minus = Sub
transOp Times = Mul
transOp Divide = Div

transExp :: Exp -> [Instruction]
transExp (Const n) = [LoadImm n]
transExp (Ident x) = [Load x]
transExp (Unop Minus e) = transExp e ++ [MulImm (-1)]

-- Can only use Minus unary operator (e.g -3)
transExp (Unop _ _)
  = error "(transExp) Only '-' unary operator supported"

-- If constant on the left, can use immediate operand
transExp (BinOp op e (Const n)) = transExp e ++ [transOpImm op n]

-- With commutative operator, can switch order to use immediate operand
transExp (BinOp Times (Const n) e) = transExp e ++ [MulImm n]
transExp (BinOp Plus (Const n) e) = transExp e ++ [AddImm n]

-- General case for two expressions
transExp (BinOp op e1 e2) = transExp e2 ++ Push : transExp e1 ++ [transOp op]
```

Limited Register Set

One solution is to combine the register and accumulator strategies

```haskell
data Instruction
   = Add Reg Reg | Sub Reg Reg
    | Mul Reg Reg | Div Reg Reg -- Op r1 r2 -> r1 := r1 <Op> r2
    | AddImm Reg Int | SubImm Reg Int
    | MulImm Reg Int | DivImm Reg Int -- <Op>Imm r c -> r := r <Op> c
```

```
     | AddStack Reg | SubStack Reg
     | MulStack Reg | DivStack Reg -- <Op>Imm r c -> r := r <Op> mem[SP]; SP--
     | Load Reg Name     -- Load r1 n     -> r1 := mem[n]
     | LoadImm Reg Int   -- LoadImm r1 i -> r1 := i
     | Store Reg Name    -- Load r1 n     -> mem[n] := r1
     | Push Reg          -- Push r1       -> SP++; mem[SP] := r1
     | Pop Reg           -- Pop r2        -> r1 := mem[SP]; SP--
     | CompEq Reg Reg    -- CompEq r1 r2 -> r1 := r1 - r2 = 0 ? 1 : 0
     | JTrue Reg Label   -- JTrue r1 l    -> IF r1 = 1 THEN JUMP TO l
     | JFalse Reg Label -- JFalse r1 l    -> IF r1 = 0 THEN JUMP TO l
     | Define Label      -- Assembler directive to set up label
```

- When free register sremain, use the register machine strategy.
- When the limit is reached (one register left to use as accumulator), switch to accumulator strategy.

This results in most expressions using full benefit of registers, while very large expressions can still be correctly executed.

```
translateOp :: Op -> (Int -> Int -> Instruction)
translateOp Plus = Add
translateOp Minus = Sub
translateOp Times = Mul
translateOp Divide = Div

translateOpImm :: Op -> (Int -> Int -> Instruction)
translateOpImm Plus = AddImm
translateOpImm Minus = SubImm
translateOpImm Times = MulImm
translateOpImm Divide = DivImm

translateOpStack :: Op -> (Int -> Instruction)
translateOpStack Plus = AddStack
translateOpStack Minus = SubStack
translateOpStack Times = MulStack
translateOpStack Divide = DivStack

maxReg :: Int
maxReg = 10

transExp :: Exp -> Reg -> [Instruction]

-- No need to check maxreg as we only use one register (a reg or the last one)
transExp (Const n) r = [LoadImm r n]
transExp (Ident id) r = [Load r id]
transExp (Unop Minus e) r = transExp e r ++ [MulImm r (-1)]
transExp (Unop _ _) _
  = error "(transExp) Only '-' unary operator supported"

-- As * and + are commutative, the order does not matter, only use one register so no
    ↪ need to check maxReg
transExp (BinOp Times (Const n) e) r = transExp e r ++ [MulImm r n]
transExp (BinOp Plus (Const n) e) r = transExp e r ++ [AddImm r n]

-- Can run left hand, then do right hand with immediate operand
transExp (BinOp op e (Const n)) r = transExp e r ++ [translateOpImm op r n]

-- General case for two expressions, we need to take into account the registers used.
transExp (BinOp op e1 e2) r
```

```
  | r == maxReg = transExp e2 r ++ Push r : transExp e1 r ++ [translateOpStack op r]
  | otherwise = transExp e1 r ++ transExp e2 (r+1) ++ [translateOp op r (r+1)]
```