CS153: Compilers
Lecture 16:
Local Optimization II

Stephen Chong
https://www.seas.harvard.edu/courses/cs153
Announcements

• Project 4 out
  • Due today!

• Project 5 out
  • Due Tuesday Nov 13 (19 days)

• Project 6 out
  • Due Tuesday Nov 20 (26 days)
Today

- Monadic form
- Implementation of some local optimizations
Monadic Form

- We will put programs into **monadic form**
  - A syntactic form that lets us easily distinguish side-effecting expressions from pure expressions
  - Enable simpler implementations of optimizations
  - Take CS152 to find out why it’s called monadic form!

- Recall: assume that variable names are distinct
datatype operand =
   (* small, pure expressions, okay to duplicate *)
   Int of int | Bool of bool | Var of var

and value =
   (* larger, pure expressions, okay to eliminate *)
   Op of operand
   | Fn of var * exp
   | Pair of operand * operand
   | Fst of operand | Snd of operand
   | Primop of primop * (operand list)

and exp =
   (* control & effects: deep thoughts needed here *)
   Return of operand
   | LetValue of var * value * exp
   | LetCall of var * operand * operand * exp
   | LetIf of var * operand * exp * exp * exp
Converting to Monadic Form

- Similar to lowering to MIPS:
  - Operands are either variables or constants.
    - Means we don't have to worry about duplicating operands since they are pure and aren't big.
  - We give a (unique) name to more complicated terms by binding it with a `let`
    - That will allow us to easily find common sub-expressions.
    - The uniqueness of names ensures we don't run into capture problems when substituting.
  - We keep track of those expressions that are guaranteed to be pure.
    - Makes doing inlining or dead-code elimination easy.
  - We flatten out `let`-expressions.
    - More scope for factoring out common sub-expressions.
Example

\((x+42+y) \times (x+42+z)\)

```
let t1 = (let t2 = x+42
t3 = t2+y in t3)
t4 = (let t5 = x+42
t6 = t5+z in t6)
t7 = t1*t4
in t7
```

```
let t2 = x+42
t3 = t2+y
t4 = t6
t7 = t1*t4
in t7
```

```
let t2 = x+42
t3 = t2+y
t6 = t2+z
t4 = t6
t7 = t1*t4
in t7
```
Some General ML Equations

• Optimizations in essence rewrite expressions according to equivalences

• E.g.,
  • 1. let x = v in e == e[x↦v]
  • 2. (fun x -> e) v == let x = v in e
  • 3. let x = (let y = e₁ in e₂) in e₃
     ==
     let y = e₁ in let x = e₂ in e₃
  • 4. e₁ e₂
     ==
     let x=e₁ in let y=e₂ in x y
  • 5. (e₁,...,eₙ) ==
     let x₁=e₁ ... xₙ=eₙ in (x₁,...,xₙ)
What About Metrics?

• We should rewrite when we improve the program

• E.g.,
  
  1. $3 + 4 \geq 7$
  
  2. $(\text{fun } x \rightarrow e) \ v \geq \ \text{let } x = v \ \text{in } e$
  
  3. $\text{let } x = v \ \text{in } e \geq e$
     (when $x$ doesn't occur in $e$)
  
  4. $\text{let } x = v \ \text{in } e \ ??? \ e[x\mapsto v]$
Let Reduce or Let Expand?

- Reducing `let x = v in e` to `e[x↦v]` is profitable when `e[x↦v]` is “no bigger”
  - e.g., when `x` does not occur in `e` (dead code elimination)
  - e.g., when `x` occurs at most once in `e`
  - e.g., when `v` is small (constant or variable) (constant & copy propagation)
  - e.g., when further optimizations reduce the size of the resulting expression.
Let Reduce or Let Expand?

• Expanding $e[x\mapsto v]$ to `let x = v in e` can be good for shrinking code (common sub-expression elimination)

• E.g., $(x*42+y) + (x*42+z)$ becomes

  \[
  \text{let } w = x*42 \text{ in } (w+y) + (w+z)
  \]
Reduction Algorithms

- **Constant folding**
  - reduce if's and arithmetic when args are constants

- **Operand propagation**
  - replace each `LetValue(x, Op(w), e)` with `e[x↦w]`
  - why can't we do `LetValue(x, v, e)` with `e[x↦v]`?

- **Common Sub-Value elimination**
  - replace each `LetValue(x, v, ...LetValue(y, v, e), ...)` with `LetValue(x, v, ...e[y↦x]...)`

- **Dead Value elimination**
  - When `e` doesn't contain `x`, replace `LetValue(x, v, e)` with `e`
let rec cfold_exp (e:exp) : exp =
  match e with
  | Return w -> Return w
  | LetValue(x,v,e) ->
    LetValue(x, cfold_val v, cfold_exp e)
  | LetCall(x,f,ws,e) ->
    LetCall(x,f,ws,cfold_exp e)
  | LetIf(x,Bool true,e1,e2,e)->
    cfold_exp (flatten x e1 e)
  | LetIf(x,Bool false,e1,e2,e)->
    cfold_exp (flatten x e2 e)
  | LetIf(x,w,e1,e2,e)->
    LetIf(x,w,cfold e1,cfold e2,cfold e)
Flattening

• Turn “let x = e1 in e2” into an exp

    and flatten (x:var) (e1:exp) (e2:exp):exp =
    match e1 with
    | Return w -> LetVal(x, Op w, e2)
    | LetVal(y,v,e') ->
        LetVal(y,v, flatten x e' e2)
    | LetCall(y,f,ws,e') ->
        LetCall(y,f,ws, flatten x e' e2)
    | LetIf(y,w,et,ef,ec) ->
        LetIf(y,w,et,ef, flatten x ec e2)
and cfold_val (v:value):value =
  match v with
  Fn(x,e) -> Fn(x,cfold_exp e)
  Primop(Plus,[Int i,Int j]) -> Op(Int(i+j))
  Primop(Plus,[Int 0,v]) -> Op(v)
  Primop(Plus,[v,Int 0]) -> Op(v)
  Primop(Minus,[Int i,Int j]) -> Op(Int(i-j))
  Primop(Minus,[v,Int 0]) -> Op(v)
  Primop(Lt,[Int i,Int j]) -> Op(Bool(i<j))
  Primop(Lt,[v1,v2]) ->
      if v1 = v2 then Op(Bool false) else v
  ...
  v -> v
let rec cprop_exp(env: var->oper option)(e: exp): exp =
match e with
| Return w -> Return (cprop_oper env w)
| LetValue(x, Op w, e) ->
  cprop_exp (extend env x (cprop_oper env w)) e
| LetValue(x, v, e) ->
  LetValue(x, cprop_val env v, cprop_exp env e)
| LetCall(x, f, w, e) ->
  LetCall(x, cprop_oper env f, cprop_oper env w,
           cprop_exp env e)
| LetIf(x, w, e1, e2, e) ->
  LetIf(x, cprop_oper env w,
        cprop_exp env e1, cprop_exp env e2,
        cprop_exp env e)
and cprop_oper env w =
  match w with
  | Var x ->
    (match env x with | None -> w | Some w2 -> w2)
  | _   -> w

and cprop_val env v =
  match v with
  | Fn(x,e) -> Fn(x,cprop_exp env e)
  | Pair(w1,w2) ->
    Pair(cprop_oper env w1, cprop_oper env w2)
  | Fst w -> Fst(cprop_oper env w)
  | Snd w -> Snd(cprop_oper env w)
  | Primop(p,ws) -> Primop(p,map (cprop_oper env) ws)
  | Op(_)   -> raise Impossible
let rec cse_exp(env:value->var option)(e:exp):exp =
  match e with
  | Return w -> Return w
  | LetValue(x,v,e) ->
    (match env v with
     | None -> LetValue(x,cse_val env v,
         cse_exp (extend env v x) e)
     | Some y -> LetValue(x,Op(Var y),cse_exp env e))
  | LetCall(x,f,w,e) -> LetCall(x,f,w,cse_exp env e)
  | LetIf(x,w,e1,e2,e) ->
    LetIf(x,w,cse_exp env e1,cse_exp env e2,
        cse_exp env e)
and cse_val env v =
  match v with
  | Fn(x,e) -> Fn(x,cse_exp env e)
  | v -> v
Dead Value Elimination (naive)

```ocaml
let rec dead_exp (e:exp) : exp =
  match e with
  | Return w -> Return w
  | LetValue(x,v,e) ->
    if count_occurs x e = 0 then dead_exp e
    else LetValue(x,v,dead_exp e)
  | LetCall(x,f,w,e) ->
    LetCall(x,f,w,dead_exp env e)
  | LetIf(x,w,e1,e2,e) ->
    LetIf(x,w,dead_exp env e1,
          dead_exp env e2,dead_exp env e)
```
Comments

- It’s possible to fuse constant folding, operand propagation, common value elimination, and dead value elimination into one giant pass.
  - one env to map variables to operands
  - one env to map values to variables
  - on way back up, return a table of use-counts for each variable.

- There are plenty of improvements:
  - e.g., sort operands of commutative operations so that we get more common sub-values.
  - e.g., keep an env mapping variables to values and use this to reduce fst/snd operations.
    - LetValue(x,Pair(w₁,w₂),…,LetValue(y,Snd(Op x),…)
      becomes LetValue(x,Pair(w₁,w₂),…,LetValue(y,Op w₂,…)}
Function Inlining

• Replace

\[
\text{LetValue}(f, \text{Fn}(x, e1)), \ldots \text{LetCall}(y, f, w, e2)
\]

... with

\[
\text{LetValue}(f, \text{Fn}(x, e1)), \ldots \text{LetValue}(y, \text{LetValue}(x, \text{Op w, e1}), e2) \ldots
\]

• Problems:
  • Monadic form doesn't have nested Let's!
    (so we must flatten out the nested let.)
  • Bound variables get duplicated
    (so we rename them as we flatten them out.)
When to Inline?

• Recall heuristics from last week:
  • Expand only function call sites that are called frequently
  • Expand only functions with small bodies
  • Expand functions that are called only once
    • Dead function elimination will remove the now unused function
Optimizations So Far...

- Constant folding
- Operand propagation
  - copy propagation: substitute a variable for a variable
  - constant propagation: substitute a constant for a variable
- Dead value elimination
- Common sub-value elimination
- Function inlining
Optimizing Function Calls

• We never eliminate `LetCall(x,f,w,e)` since the call might have effects.

• But if we can determine that `f` is a function without side effects, then we could treat this like a `LetVal` declaration.
  • Then we get cse, dce, etc. on function calls!
  • E.g., `fact(10000) + fact(10000)` becomes
    ```
    let t = fact(10000) in t + t
    ```

• In general, we won't be able to tell if `f` has effects.
  • Idea: use a modified type-inference to figure out which functions have side effects.
  • Idea 2: make the programmer distinguish between functions that have effects and those that do not.
Optimizing Conditionals

- if v then e else e becomes e
- if v then ...(if v then e1 else e2)... else e3 becomes
  if v then ...e1...else e3
- let x = if v then e1 else e2 in e3 becomes
  if v then let x=e1 in e3 else let x=e2 in e3
- if v then ...let x=v1... else ...let y=v1...
  becomes
  let z=v1 in if v then ...let x=z... else ...let y=z...
  (when vars(v1) defined before the if)
- let x=v1 in (if v then ...x... else ...(no x)...)
  becomes
  if v then (let x=v1 in ...x...) else ...(no x)...