Algebraic Optimization

CS4410: Spring 2013
Optimization:

Want to rewrite code so that it's:
- faster, smaller, consumes less power, etc.
- while retaining the "observable behavior"
- usually: input/output behavior
- often need analysis to determine that a given optimization preserves behavior.
- often need profile information to determine that a given optimization is actually an improvement.

Often have two flavors of optimization:
- high-level: e.g., at the AST-level (e.g., inlining)
- low-level: e.g., right before instruction selection (e.g., register allocation)
Some algebraic optimizations:

- **Constant folding (delta reductions):**
  - e.g., $3+4 \implies 7$, $x*1 \implies x$
  - e.g., if true then s else t $\implies s$

- **Strength reduction**
  - e.g., $x*2 \implies x+x$, $x \div 8 \implies x \gg 3$

- **Inlining, constant propagation, copy propagation, dead-code elimination, etc. (beta reduction):**
  - e.g., let val x = 3 in x + x end $\implies 3 + 3$

- **Common sub-expression elimination (beta expansion):**
  - e.g., $(\text{length } x) + (\text{length } x) \implies$
    let val i = length x in i+i end
More optimizations:

• Loop invariant removal:

  \[
  \text{for } (i=0; i<n; i+=s*10) \ldots \implies \\
  \text{int } t = s*10; \text{ for } (i=0; i<n; i+=t)\ldots
  \]

• Loop interchange:

  \[
  \text{for } (i=0; i<n; i++) \\
  \text{ for } (j=0; j<n; j++) \\
  \quad s += A[j][i]; \implies \\
  \text{for } (j=0; j<n; j++) \\
  \text{ for } (i=0; i<n; i++) \\
  \quad s += A[j][i];
  \]
More optimizations:

• Loop fusion, deforestation:
  – e.g., \((\text{map } f)(\text{map } g \; x)\) ==> \(\text{map } (f \circ g) \; x\)
  – e.g., \(\text{foldl } (+) \; 0 \; (\text{map } f \; x)\) ==> 
    \(\text{foldl } (\text{fn } (y,a) => (f \; y)+a) \; 0 \; x\)

• Uncurrying:
  – let val \(f = \text{fn } x => \text{fn } y => x + y\) in 
    …\(f\; a\; b\)…  ==> 
    let val \(f = \text{fn } (x,y) => x+y\) in 
    …\(f(a,b)\)…

• Flattening/unboxing:
  – let val \(x = ((a,b),(c,d))\) in 
    …\#1(#2 \; x)…  ==> 
    let val \(x = (a,b,c,d)\) in 
    …\#3 \; x…
When is it safe to rewrite?

When can we safely replace $e_1$ with $e_2$?

1. when $e_1 = e_2$ from an input/output point of view.

2. when $e_1 \leq e_2$ from our improvement metrics (e.g., performance, space, power)
I/O Equivalence

• Consider let-reduction:
\[(\text{let } x = e_1 \ \text{in } e_2) =?= (e_2[e_1/x])\]
where \(e_2[e_1/x]\) is \(e_2\) with \(e_1\) substituted for \(x\)

When does this equation hold?

– give some positive examples?
– give some negative examples?
Some Negatives:

let x = print "hello" in x+x

let x = print "hello" in 3

let x = raise Foo in 3

let x = ref 3
in
  x := !x + 1; !x
For ML:

\[(\text{let } x = e_1 \text{ in } e_2) =?= (e_2[e_1/x])\]

Holds for sure when \(e_1\) has no observable effects.

Observable effects include:

- diverging
- input/output
- allocating or reading/writing refs & arrays
- raising an exception
In Particular:

\[(\text{let } x = v \text{ in } e) \equiv (e[v/x])\]

where \(v\) is drawn from the subset of expressions:

\[v ::= i \quad (* \text{ constants } *)\]

| \(x\) \quad (* \text{ variables } *)
| \(v \op v\) \quad (* \text{ binops of vals } *)
| \((v_1,\ldots,v_n)\) \quad (* \text{ tuples of vals } *)
| \#i v \quad (* \text{ select of a val } *)
| D v \quad (* \text{ constructors } *)
| \text{fun } x \rightarrow e \quad (* \text{ functions } *)
| \text{let } x = v_1 \text{ in } v_2
Another Problem

\[
\begin{align*}
\text{let } x & = \text{foo()} \text{ in} \\
\text{let } y & = x+x \text{ in} \\
\text{let } x & = \text{bar()} \text{ in} \\
& \mathsf{y} \times \mathsf{y}
\end{align*}
\]

\[
\begin{align*}
\text{let } x & = \text{foo()} \text{ in} \\
\text{let } x & = \text{bar()} \text{ in} \\
& (x+x) \times (x+x)
\end{align*}
\]
Variable Capture

• When substituting a value \( v \) for a variable \( y \), we must make sure that none of the free variables in \( v \) is accidentally captured.

• A simple solution is to just rename all the variables so they are unique (throughout the program) before doing any reductions.

• Must be sure to preserve uniqueness.
Avoiding Capture

\[
\text{let } x = \text{foo()} \text{ in } \text{let } y = x+x \text{ in } \text{let } z = \text{bar()} \text{ in } y * y
\]

\[
\text{let } x = \text{foo()} \text{ in } \text{let } z = \text{bar()} \text{ in } (x+x) * (x+x)
\]
Some General ML Equations

1. let x = v in e == e[v/x]

2. (fun x -> e) v == let x = v in e

3. let x =(let y = e_1 in e_2) in e_3 ==
   let y = e_1 in let x = e_2 in e_3

4. e_1 e_2 == let x=e_1 in let y=e_2 in x y

5. (e_1,...,e_n) ==
   let x_1=e_1 ... x_n=e_n in (x_1,...,x_n)
What about metrics?

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 $v$ doesn't occur in $e$)

4. $\text{let } x = v \text{ in } e =?= e[v/x]
Let reduce or expand?

The first direction:

```
let x = v in e ≥ e[v/x]
```

is profitable when $e[v/x]$ 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 expand?

The second direction:

\[ e[v/x] \geq \text{let } x = v \text{ in } e \]

can be good for shrinking code (common sub-expression elimination.)

For example:

\[(x*42+y) + (x*42+z) \rightarrow \]
\[\text{let } w = x*42\]
\[\text{in } (w+y) + (w+z)\]
How to do reductions?

Naïve solution:

iterate until no change
    find sub-expression that can be reduced
    and reduce it.

Many questions remain:
    For example, how do we find common sub-expressions?
Monadic Form:

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 thought needed here *)
    Return of operand
    | LetValue of var * value * exp
    | LetCall of var * operand * operand * exp
    | LetIf of var * operand * exp * exp * exp
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:

\[(x+42+y) \times (x+42+z) \implies\]

\[
\text{let } t1 = (\text{let } t2 = x+42 \\
    \quad t3 = t2+y \text{ in } t3) \\
    t4 = (\text{let } t5 = x+42 \\
    \quad t6 = t5+z \text{ in } t6) \\
    t7 = t1 \times t4 \\
\text{in } t7 \implies
\]

\[
\text{let } t2 = x+42 \quad \text{let } t2 = x+42 \\
    t3 = t2+y \quad t3 = t2+y \\
    t1 = t3 \quad t6 = t2+z \\
    t5 = x+42 \quad \implies t7 = t3 \times t6 \\
    t6 = t5+z \quad \text{in } t7 \\
    t4 = t6 \\
    t7 = t1 \times t4 \\
\text{in } t7
\]
Reduction Algorithms:

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

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

- **Common Sub-Value elimination**
  - replace each LetValue(x,v,…LetValue(y,v,e),…) with LetValue(x,v,…e[x/y]…)

- **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

and flatten (x:var) (e1:exp) (e2:exp):exp =
match e1 with
  | Return w -> LetVal(x,Op w,2)
  | LetValue(y,v,e1) ->
    LetValue(y,v,flatten x e1 e2)
  | LetCall(y,f,ws,e1) ->
    LetCall(y,f,ws,flatten x e1 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
Operand Propagation

```ocaml
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
Common Value Elimination

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 (Naïve)

```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 e)
  | LetIf(x,w,e1,e2,e) ->
    LetIf(x,w,dead_exp e1,
          dead_exp e2,dead_exp 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),…)
    => LetValue(x,Pair(w1,w2),…,LetValue(y,Op w2,…)}
Function Inlining:

Replace:

```
LetValue(f,Fn(x,e1),…LetCall(y,f,w,e2)…)  
with

LetValue(f,Fn(x,e1),…  
LetValue(y,LetValue(x,Op w,e1),e2)…)  
```

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?

- Certainly when f occurs at most once.
  - Not going to blow up the code since DVE will get rid of the original after inlining.
- We could try inlining at each call site, then reduce, and then see if the result is no worse than the original code.

- In practice, rarely done.
- Instead, just inline "small" functions.
  - e.g., map will be inlined by SML/NJ
Monadic Form:

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 thought needed here *)
    Return of operand
| LetValue of var * value * exp
| LetCall of var * operand * operand * exp
| LetIf of var * operand * exp * exp * exp
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 completely 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!
• To what expressions can f be bound?
  – Lambda, a call, Fst x, Snd x, Hd x, etc.
  – 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 \( e \) → \( e \)
- if \( v \) then \( \ldots (if \ v \ then \ e_1 \ else \ e_2) \ldots \) else \( e_3 \) → 
  if \( v \) then \( \ldots e_1 \ldots \) else \( e_3 \)
- let \( x = if \ v \ then \ e_1 \ else \ e_2 \) in \( e_3 \) → 
  if \( v \) then let \( x = e_1 \) in \( e_3 \) else let \( x = e_2 \) in \( e_3 \)
- if \( v \) then \( \ldots let \ x = v_1 \ldots \) else \( \ldots let \ y = v_1 \ldots \) → 
  let \( z = v_1 \) in if \( v \) then \( \ldots let \ x = z \ldots \) else \( \ldots let \ y = z \ldots \) (when \( \text{vars}(v_1) \) defined before the \( if \))
- let \( x = v_1 \) in if \( v \) then \( \ldots x \ldots \) else \( \ldots (no \ x) \ldots \) → 
  if \( v \) then let \( x = v_1 \) in \( \ldots x \ldots \) else \( \ldots (no \ x) \ldots \)
Optimizing Loops

LetRec([(f_1,x_1,e_1),…,(f_n,x_n,e_n)],e)

• Loop invariant removal:
  – if e_i = …let x=v in…
  – and if vars(v) are defined before the LetRec
  – then we can hoist the definition out of the loop.

• e.g.,
  val z = 42
  fun f x = (…z*31…) →
  val t = z*31
  fun f x = (…t…)
Other Algebraic Laws?

If f and g have no effects, then:

- \( \text{map } f = \text{foldr } (\text{fn } (x,a) \Rightarrow (f x)::a) \ [\] \)
- \( \text{filter } f = \text{foldr } (\text{fn } (x,a) \Rightarrow \text{if } f x \text{ then } x::a \text{ else } a) \ [\] \)
- \( (\text{foldr } f \ u) \circ (\text{map } g) = \text{foldr } (\text{fn } (x,a) \Rightarrow f(g x,a)) \ u \)
- \( (\text{foldr } f \ u) \circ (\text{filter } g) = \)
  \( \text{foldr } (\text{fn } (x,a) \Rightarrow \text{if } g x \text{ then } f(x,a) \text{ else } a) \ u \)

So any (pure) foldr combined with any sequence of (pure) filters and maps can be reduced to a single traversal of the list!

This generalizes to any inductive datatype!
Getting into Monadic Form

• Lots of optimizations are simplified by translating into monadic form.
• How do we (efficiently) get ML code into monadic form?
• Let's first consider a simpler source:
  ```ml
  type arith =
    I of int | Add of arith * arith
  ```
• And a simpler target:
  ```ml
  type exp =
    Return of operand
    | Let of var * value * exp
  ```
Very Naïve way:

val split : exp -> (var * value) list * operand
val join : (var * value) list * operand -> exp

let rec tomonadic (a:arith) : exp =
  match a with
  | I(i) -> Return(Int i)
  | Add(a,b) ->
    let x = fresh_var() in
    let (da,wa) = split(tomonadic a) in
    let (db,wb) = split(tomonadic b) in
    join (da @ db @ [((x,PrimApp(Plus,[wa,wb])))], Var x)
let rec split (e: exp): (var * value) list * operand =
    match e with
    | Return w -> ([], w)
    | Let(x, v, e) ->
        let (ds, w) = split e
        in ((x, v) :: ds, w)

let rec join (ds: var * value list, w: operand) : exp =
    match ds with
    | [] -> Return w
    | (x, v) :: rest -> Let(x, v, join(rest, w))
Problems:

• Expensive to split/join on each compound expr.
• Must generalize split/join to return a declaration list that covers all of the other cases beyond values.

let rec tomonadic (a:arith) : exp =
  match a with
  | I(i) -> Return(Int i)
  | Add(a,b) ->
    let x = fresh_var() in
    let (da,wa) = split(tomonadic a) in
    let (db,wb) = split(tomonadic b) in
    join (da @ db @ [(x,PrimApp(Plus,[wa,wb]))], Var x)
Avoiding Splits and Joins:

Don't bother joining until the end:

```ocaml
let rec tom (a:arith) : (var*value) list * oper =
  match a with
  | I(i) => ([], Int i)
  | Add(a,b) =>
    let x = fresh_var() in
    let (da,wa) = tom a in
    let (db,wb) = tom b in
    (da @ db @ [(x,PrimApp(Plus,[wa,wb]))], Var x)
end

let tomonadic(a:arith):exp = join(tom a)
```
let rec tom (a:arith) : (var*value) list * oper =
    match a with
    | I(i) -> ([],Int i)
    | Add(a,b) ->
        let x = fresh_var() in
        let (da,wa) = tom a in
        let (db,wb) = tom b in
        (da @ db @ [(x,PrimApp(Plus,[wa,wb]))],
         Var x)

• Appends are causing us to be quadratic.
let rec tom (a:arith) (ds: (var*value) list) : (var*value) list * oper =
  match a with
  | I(i) -> (ds, Int i)
  | Add(a,b) ->
    let x = fresh_var() in
    let (da,wa) = tom ds a in
    let (db,wb) = tom da b
    in
    ((x, PrimApp(Plus, [wa, wb])) :: db,
     Var x)

fun tomonadic(a:arith):exp = revjoin(tom a)
let rec tom (a: arith) (ds: (var*value) list) : (var*value) list * oper =
  match a with
  | I(i) -> (ds, Int i)
  | Add(a, b) ->
    let x = fresh_var() in
    let (da, wa) = tom ds a in
    let (db, wb) = tom da b
    in
    ((x, PrimApp(Plus, [wa, wb]))::db,
     Var x)

• Still have to generalize to cover all of the other Let cases beyond values (e.g., Call, If, etc.)
What we wish we could do...

e = \text{Let}(x_1,v_1,
                   \text{Let}(x_2,v_2,\ldots
                       \text{Let}(x_n,v_n,\text{Return } w)\ldots))

Imagine we could split an expression e into a "hole-y" expression and the Return'ed operand:

\begin{align*}
\text{split } e &= (h, w) \\
\text{where } h \text{ is } \text{Let}(x_1,v_1,
                   \text{Let}(x_2,v_2,\ldots
                       \text{Let}(x_n,v_n, [o] )\ldots))
\end{align*}
Plugging Holes

Imagine we could plug another expression (with a hole) into the "hole":

\[
\text{plug } (\text{Let}(x_1,v_1, \\
\text{Let}(x_2,v_2, \\
\text{Let}(x_n,v_n,[o])...)) \\
(\text{Let}(y_1,z_1, \\
\text{Let}(y_2,z_2, \\
\text{Let}(y_n,z_n,[o])...))) = \\
\text{Let}(x_1,v_1, \\
\text{Let}(x_2,v_2, \\
\text{Let}(x_n,v_n, \\
(\text{Let}(y_1,z_1, \\
\text{Let}(y_2,z_2, \\
\text{Let}(y_n,z_n,[o])...))...)))...)
\]
Recoding:

```ocaml
val hole : holy_exp
val plug : holy_exp -> holy_exp -> holy_exp
val plug_final : holy_exp * operand -> exp
let rec tom (a:arith) : holy_exp * operand =
  match a with
  | I(i) -> (hole ,Int i)
  | Add(a,b) ->
    let x = fresh_var() in
    let (ha,wa) = tom a in
    let (hb,wb) = tom b
    in
    (plug ha
     (plug hb(Let(x,PrimApp(Plus,[wa,wb]), hole))),
      Var x)

let tomonadic(a:arith):exp = plug_final(tom a)
```
Implementing Hole-y Expr's

• How to implement holy expressions?

val hole : holy_exp
val plug : holy_exp -> holy_exp -> holy_exp
val plug_final : holy_exp * operand -> exp
We've already seen one option:

type decl =
    Vald of var * value
| Calld of var * operand * operand
| Ifd of var * exp * exp

type holy_exp = decl list
A Clever Option...

type holy_exp = exp -> exp

let hole : holy_exp =
  fun e -> e

let plug (h1:holy_exp)(h2:holy_exp) =
  fun e -> h1(h2(e)) (* = h1 o h2 *)

let plugFinal(h:holy_exp)(w:operand) =
  h (Return w) (* = h o Return *)
Tom revisited:

```ocaml
let hole : holy_exp = fun e -> e
let plug : holy_exp -> holy_exp -> holy_exp
    fun ha -> fn hb -> (fun e -> ha(hb(e)))
let rec tom (a:arith) : holy_exp * operand = 
    match a with
    | I(i) -> (hole,Int i)
    | Add(x,b) ->
        let x = fresh_var() in
        let (ha,wa) = tom a in
        let (hb,wb) = tom b in
        (plug ha (plug hb
                      (fun e -> (Let(x,PrimApp(Plus,[wa,wb]),e)),
                           Var x))
```
Tom Simplified:

let rec tom (a:arith) : (exp->exp) * operand =
   match a with
   | I(i) -> (fun e -> e,Int i)
   | Add(x,b) ->
      let x = fresh_var() in
      let (ha,wa) = tom a in
      let (hb,wb) = tom b
      in
      (fun e ->
       ha(hb(Let(x,PrimApp(Plus,[wa,wb]),e))),
       Var x)
   end

let tomonadic(a:arith) =
   let(h,w) = tom a in h (Return w)
let rec tom(a:arith)(ds:holy_exp):holy_exp * oper =
  match a with
  | I(i) -> (ds,Int i)
  | Add(a,b) =>
    let x = fresh_var() in
    let (da,wa) = tom ds a in
    let (db,wb) = tom da b
    in
    (fun e -> db(Let(x,PrimApp(Plus,[wa,wb]),e)),
     Var x)
One more step...

Instead of:

\[
tom : \text{arith} \rightarrow (\text{exp} \rightarrow \text{exp}) \rightarrow (\text{exp} \rightarrow \text{exp}) \ast \text{operand}
\]

- The \((\text{exp} \rightarrow \text{exp})\) argument represents the declarations given so far, whereas the \((\text{exp} \rightarrow \text{exp})\) result represents the append of the declarations of \text{arith} to the declarations given so far.

The code given to you has the form:

\[
tom : \text{arith} \rightarrow (\text{operand} \rightarrow \text{exp}) \rightarrow \text{exp}
\]

- The \((\text{operand} \rightarrow \text{exp})\) argument is a holey-expression that represents how the rest of the surrounding expression should be built.
let rec tom (a:arith) (ds:operand->exp) =
  match a with
  | I(i) -> ds(Int i)
  | Add(a,b) ->
    let x = fresh_var() in
    tom a (fun wa ->
      tom b (fun wb ->
        LetVal(x,PrimApp(Plus,[wa,wb]),ds x)))

let tomonadic (a:arith) : exp =
  tom a (fun v -> Return v)
Example:

```ml
let rec tom (a:arith) (ds:operand->exp) =
  match a with
  | I(i) -> ds(Int i)
  | Add(a,b) ->
    let x = fresh_var() in
    tom a (fun wa ->
      tom b (fun wb ->
        LetVal(x,PrimApp(Plus,[wa,wb]),ds x))
    )

let tomonadic (a:arith) : exp =
  tom a (fun v -> Return v)
```

tomonadic(I 31) =
tom(I 31) Return = Return(Int 31)
Next Example:

```ocaml
let rec tom (a:arith) (ds:operand->exp) =
  match a with
  | I(i) -> ds(Int i)
  | Add(a,b) ->
    let x = fresh_var() in
    tom a (fun wa ->
      tom b (fun wb ->
        LetVal(x,PrimApp(Plus,[wa,wb]),ds x))
    )

tomonadic(Add(I 31, I 42)) =

tom(Add(I 31, I 42)) (fun v -> Return v) =
  tom (I 31) (fun wa ->
    tom (I 42) (fun wb ->
      LetVal("x1",PrimApp(Plus,[wa,wb]),Return "x1")))
```
Example Continued:

```plaintext
tom(Add(I 31, I 42)) (fun v -> Return v) =
    tom (I 31) (fun wa ->
        tom (I 42) (fun wb ->
            LetVal("x1", PrimApp(Plus, [wa, wb]), Return "x1"))
    )

tom (I 31) ds = ds(Int 31) so...

tom (I 31) (fun wa ->
    tom (I 42) (fun wb ->
        LetVal("x1", PrimApp(Plus, [wa, wb]),
            Return "x1"))
    )
= tom (I 42) (fun wb ->
    LetVal("x1", PrimApp(Plus, [Int 31, wb]),
        Return "x1"))
```
Example Continued:

tom (I 42) ds = ds(Int 42) so...

tom (I 42) (fun wb ->
  LetVal("x1",PrimApp(Plus,[Int 31,wb]),
    Return "x1"))
=
  LetVal("x1",PrimApp(Plus,[Int 31,Int 42]),
    Return "x1")
The Real Code

• See monadic.ml for the real code.
• It has to deal with many more cases but has the same basic structure.

let rec tom (a:arith) (ds:operand->exp) =
    match a with
    | I(i) -> ds(Int i)
    | Add(a,b) ->
        let x = fresh_var() in
        tom a (fun wa ->
                tom a (fun wb ->
                          LetVal(x,PrimApp(Plus,[wa,wb]),ds x)))
    let tomonadic (a:arith) : exp =
        tom a (fun v -> Return v)