

Algebraic Optimization

CS153: Compilers

Greg Morrisett

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., $\text{if true then } s \text{ else } t \implies s$
- Strength reduction
 - e.g., $x*2 \implies x+x$, $x \text{ div } 8 \implies x \gg 3$
- Inlining, constant propagation, copy propagation, dead-code elimination, etc. (beta reduction):
 - e.g., $\text{let val } x = 3 \text{ in } x + x \text{ end} \implies 3 + 3$
- Common sub-expression elimination (beta expansion):
 - e.g., $(\text{length } x) + (\text{length } x) \implies$
 $\text{let val } i = \text{length } x \text{ in } i+i \text{ end}$

More optimizations:

- Loop invariant removal:

```
for (i=0; i<n; i+=s*10) ... ==>
int t = s*10; for (i=0; i<n; i+=t) ...
```

- Loop interchange:

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    s += A[j][i];    ==>
```

```
for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    s += A[j][i];
```

More optimizations:

- Loop fusion, deforestation:
 - e.g., $(\text{map } f)(\text{map } g \ x) \implies \text{map } (f \circ g) \ x$
 - e.g., $\text{foldl } (\text{op } +) \ 0 \ (\text{map } f \ x) \implies \text{foldl } (\text{fn } (y,a) \Rightarrow (f \ y)+a) \ 0 \ x$
- Uncurrying:
 - $\text{let val } f = \text{fn } x \Rightarrow \text{fn } y \Rightarrow x + y \text{ in } \dots f \ a \ b \dots \implies \text{let val } f = \text{fn } (x,y) \Rightarrow x+y \text{ in } \dots f(a,b) \dots$
- Flattening/unboxing:
 - $\text{let val } f = \text{fn } (x,y) \Rightarrow x+y \text{ in } \dots f(a,b) \dots \implies \text{let val } f = \text{fn } \{|x,y|\} \Rightarrow x+y \text{ in } \dots f\{|a,b|\} \dots$
 - $\text{let val } x = ((a,b),(c,d)) \text{ in } \dots \#1(\#2 \ x) \dots \implies \text{let val } x = (a,b,c,d) \text{ in } \dots \#3 \ x \dots$

When is it safe to rewrite?

When can we safely replace e_1 with e_2 ?

1. when $e_1 \equiv e_2$ 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:

$$(\mathbf{let\ } \mathbf{x} = \mathbf{e}_1 \mathbf{\ in\ } \mathbf{e}_2) =?= (\mathbf{e}_2[\mathbf{e}_1/\mathbf{x}])$$

where $\mathbf{e}_2[\mathbf{e}_1/\mathbf{x}]$ is \mathbf{e}_2 with \mathbf{e}_1 substituted for \mathbf{x}

When does this equation hold?

- give some positive examples?
- give some negative examples?

Some Negatives:

```
let val x = print "hello" in x+x end
```

```
let val x = print "hello" in 3 end
```

```
let val x = raise Foo in 3 end
```

```
let val x = ref 3  
in  
    x := !x + 1; !x  
end
```

For ML:

$$(\text{let } \mathbf{x} = \mathbf{e}_1 \text{ in } \mathbf{e}_2) =?= (\mathbf{e}_2[\mathbf{e}_1/\mathbf{x}])$$

Holds for sure when \mathbf{e}_1 has no observable effects.

Observable effects include:

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

In Particular:

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

where v is drawn from the subset of expressions:

$v ::= i$ (* constants *)
| x (* variables *)
| $v \text{ op } v$ (* binops of vals *)
| (v_1, \dots, v_n) (* tuples of vals *)
| $\#i \ v$ (* select of a val *)
| $D \ v$ (* constructors *)
| $\text{fn } x \Rightarrow e$ (* functions *)
| $\text{let val } x = v_1 \text{ in } v_2$

Another Problem

```
let val x = foo ()
    val y = x+x
    val x = bar ()
in
    y * y
end
```

```
let val x = foo ()
    val x = bar ()
in
    (x+x) * (x+x)
end
```

Variable Capture

- When substituting a value v for a variable x , 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

```
let val x = foo()  
    val y = x+x  
    val z = bar()  
in  
    y * y  
end
```

```
let val x = foo()  
    val z = bar()  
in  
    (x+x) * (x+x)  
end
```

Some General ML Equations

1. $\text{let } x = v \text{ in } e == e[v/x]$

1. $(\text{fn } x \Rightarrow e) v == \text{let } x = v \text{ in } e$

2. $\text{let } x = (\text{let } y = e_1 \text{ in } e_2) \text{ in } e_3 ==$
 $\text{let } y = e_1 \text{ in let } x = e_2 \text{ in } e_3$

3. $e_1 e_2 == \text{let } x=e_1 \text{ in let } y=e_2 \text{ in } x y$

4. $(e_1, \dots, e_n) ==$
 $\text{let } x=e_1 \quad x=e_2 \quad \dots \quad x=e_n \text{ in } (x \quad x \quad \dots \quad x)$

What about metrics?

1. $3 + 4 \geq 7$
2. $(\text{fn } x \Rightarrow e) v \geq \text{let } x = v \text{ in } e$
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 \geq **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:

$(x*42+y) + (x*42+z) \quad \dashrightarrow$

$\text{let } w = x*42$

$\text{in } (w+y) + (w+z)$

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
```

```
| If 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:

$(x+42+y) * (x+42+z) \implies$

```
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 \implies

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

in t7

Reduction Algorithms:

- Constant folding
 - reduce if's and arithmetic when args are constants
- Operand propagation
 - replace each $\text{LetValue}(x, \text{Op}(w), e)$ with $e[w/x]$.
 - why can't we do $\text{LetValue}(x, v, e)$ with $e[v/x]$?
- Common Sub-Value elimination
 - replace each $\text{LetValue}(x, v, \dots \text{LetValue}(y, v, e), \dots)$ with $\text{LetValue}(x, v, \dots e[x/y] \dots)$
- Dead Value elimination
 - When e doesn't contain x , replace $\text{LetValue}(x, v, e)$ with e .

Constant Folding

```
fun cfold_exp (e:exp) : exp =
  case e of
    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,cprop_exp env 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)
and flatten x (Return w) e = LetVal(x,Op w,e)
| flatten x (LetValue(y,v,e1)) e2 =
    LetValue(y,v,flatten x e1 e2)
| flatten x (LetCall(y,f,ws,e1)) e2 =
    LetCall(y,f,ws,flatten x e1 e2)
| flatten x (LetIf(y,w,e1,e2),e3) e4 =
    LetIf(y,w,e1,e2,flatten x e3 e4)
```

Constant Folding Contd.

```
and cfold_val (v:value):value =
  case v of
    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

```
fun cprop_exp (env:var->oper option) (e:exp) : exp =
  case e of
    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)
```

Operand Propagation Contd.

```
and cprop_oper env w =
  case w of
    Var x =>
      (case env x of NONE => w | SOME w2 => w2)
  | _ => w

and cprop_val env v =
  case v of
    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

```
fun cse_exp (env:value->var option) (e:expr) : exp =
  case e of
    Return w => Return w
  | LetValue(x,v,e) =>
    (case env v of
      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 (Fn(x,e)) = Fn(x,cse_exp env e)
    | cse_val env v = v
```

Dead Value Elimination (Naïve)

```
fun dead_exp (e:exp) : exp =  
  case e of  
    Return w => Return w  
  | LetValue(x,v,e) =>  
      if occurs e x = 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.
$$\text{LetValue}(x, \text{Pair}(w1, w2), \dots, \text{LetValue}(y, \text{Snd}(\text{Op } x), \dots))$$
$$\Rightarrow \text{LetValue}(x, \text{Pair}(w1, w2), \dots, \text{LetValue}(y, \text{Op } w1, \dots))$$

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.
 - functional data structures!
- 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
```

```
| If 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 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:

- $\text{if } v \text{ then } e \text{ else } e \rightarrow e$
- $\text{if } v \text{ then } \dots(\text{if } v \text{ then } e_1 \text{ else } e_2)\dots \text{ else } e_3 \rightarrow$
 $\text{if } v \text{ then } \dots e_1 \dots \text{ else } e_3$
- $\text{let } x = \text{if } v \text{ then } e_1 \text{ else } e_2 \text{ in } e_3 \rightarrow$
 $\text{if } v \text{ then let } x=e_1 \text{ in } e_3 \text{ else let } x=e_2 \text{ in } e_3$
- $\text{if } v \text{ then } \dots \text{let } x=v_1 \dots \text{ else } \dots \text{let } y=v_1 \dots \rightarrow$
 $\text{let } z=v_1 \text{ in if } v \text{ then } \dots \text{let } x=z \dots \text{ else } \dots \text{let } y=z \dots$
(when $\text{vars}(v_1)$ defined before the if)
- $\text{let } x=v_1 \text{ in if } v \text{ then } \dots x \dots \text{ else } \dots (\text{no } x) \dots \rightarrow$
 $\text{if } v \text{ then let } x=v_1 \text{ in } \dots x \dots \text{ else } \dots (\text{no } x) \dots$

Optimizing Loops

LetRec([(f₁,x₁,e₁),..., (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:

```
datatype arith =  
  I of int | Add of arith*arith
```

- And a simpler target:

```
datatype 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
```

```
fun tomonadic (a:arith) : exp =  
  case a of  
    I(i) => Return(Int i)  
  | Add(a,b) =>  
    let val x = fresh_var()  
        val (da,wa) = split(tomonadic a)  
        val (db,wb) = split(tomonadic b)  
    in  
      join (da @ db @ [(x,PrimApp(Plus,[wa,wb]))]),  
           Var x)  
    end
```

Where...

```
fun split (e:exp) : (var * value) list * operand =  
  case e of  
    Return w => ([],w)  
  | Let(x,v,e) =>  
    let val (ds,w) = split e  
    in ((x,v)::ds,w)  
  end
```

```
fun join (ds:var*value list,w:operand) : exp =  
  case ds of  
    [] => 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.

```
fun tomonadic (a:arith) : exp =  
  case a of  
    I(i) => Return(Int i)  
  | Add(a,b) =>  
    let val x = fresh_var()  
        val (da,wa) = split(tomonadic a)  
        val (db,wb) = split(tomonadic b)  
    in  
      join (da @ db @ [(x,PrimApp(Plus,[wa,wb]))]),  
           Var x)  
    end
```

Avoiding Splits and Joins:

Don't bother joining until the end:

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

Problems:

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

- Appends are causing us to be quadratic.

Accumulator Based:

Accumulate reversed declarations:

```
fun tom (a:arith) (ds: (var*value) list) :  
    (var*value) list * oper =  
  case a of  
    I(i) => ([], Int i)  
  | Add(a,b) =>  
    let val x = fresh_var()  
        val (da,wa) = tom ds a  
        val (db,wb) = tom da b  
    in  
      ((x, PrimApp(Plus, [wa,wb])) :: db,  
       Var x)  
    end
```

```
fun tomonadic(a:arith):exp = revjoin(tom a)
```

Problems:

```
fun tom (a:arith) (ds: (var*value) list) :  
    (var*value) list * oper =  
  case a of  
    I(i) => ([], Int i)  
  | Add(a,b) =>  
    let val x = fresh_var()  
        val (da,wa) = tom ds a  
        val (db,wb) = tom da b  
    in  
      ((x, PrimApp(Plus, [wa,wb])) :: db,  
       Var x)  
    end
```

- 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 = Let (x1, v1,  
        Let (x2, v2, ...  
            Let (xn, vn, Return w) ... ) )
```

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

```
split e = (h, w)
```

```
where h is Let (x1, v1,  
               Let (x2, v2, ...  
                   Let (xn, vn, [0] ) ... ) )
```

Plugging Holes

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

```
plug (Let (x1, v1,  
          Let (x2, v2, ...  
              Let (xn, vn, [o] ) ... ) ) )  
      (Let (y1, z1,  
            Let (y2, z2, ...  
                Let (yn, zn, [o] ) ... ) ) ) =
```

```
Let (x1, v1,  
     Let (x2, v2, ...  
         Let (xn, vn,  
             (Let (y1, z1,  
                 Let (y2, z2, ...  
                     Let (yn, zn, [o] ) ... ) ) ) ... ) ) )
```

Recoding:

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

```
datatype 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
```

```
val hole : holy_exp =  
  fn e => e
```

```
plug_final (h:holy_exp) (w:operand) =  
  h (Return w)      (* = h o Return *)
```

```
plug (h1:holy_exp) (h2:holy_exp) =  
  fn e => h1 (h2 (e)) (* = h1 o h2 *)
```

Tom revisited:

```
val hole : holy_exp = fn e => e
val plug : holy_exp -> holy_exp -> holy_exp
  fn ha => fn hb => (fn e => ha(hb(e)))
fun tom (a:arith) : holy_exp * operand =
  case a of
    I(i) => (hole, Int i)
  | Add(x,b) =>
    let val x = fresh_var()
        val (ha,wa) = tom a
        val (hb,wb) = tom b
    in
      (plug ha
       (plug hb
        (fn e => (Let(x,PrimApp(Plus,[wa,wb]),e))),
        Var x)
      end
```

Tom Simplified:

```
fun tom (a:arith) : (exp->exp) * operand =
  case a of
    I(i) => (fn e=>e,Int i)
  | Add(x,b) =>
    let val x = fresh_var()
        val (ha,wa) = tom a
        val (hb,wb) = tom b
    in
      (fn e =>
        ha(hb(Let(x,PrimApp(Plus,[wa,wb]),e))),
        Var x)
    end
  end
fun tomonadic(a:arith) =
  let val (h,w) = tom a
  in h (Return w)
  end
```

Accumulator-Based:

```
fun tom (a:arith) (ds: holy_exp) : holy_exp * oper =  
  case a of  
    I(i) => (ds, Int i)  
  | Add(a,b) =>  
    let val x = fresh_var()  
        val (da,wa) = tom ds a  
        val (db,wb) = tom da b  
    in  
      (fn e => db(Let(x, PrimApp(Plus, [wa,wb]), e)),  
        Var x)  
    end
```

One more step...

Instead of:

```
tom : arith -> (exp->exp) -> (exp->exp) * operand
```

- The `(exp->exp)` *argument* represents the declarations given so far, whereas the `(exp->exp)` *result* represents the append of the declarations of `arith` to the declarations given so far.

The code given to you has the form:

```
tom : arith -> (operand->exp) -> exp
```

- The `(operand->exp)` argument is a holey-expression that represents how the rest of the surrounding expression should be built.

Even Simpler... (CPS)

```
fun tom (a:arith) (ds:operand->exp) =  
  case a of  
    I(i) => ds(Int i)  
  | Add(a,b) =>  
    let val x = fresh_var() in  
      tom a (fn wa =>  
        tom b (fn wb =>  
          LetVal(x, PrimApp(Plus, [wa, wb]), ds x)))  
    end  
fun tomonadic (a:arith) : exp = tom a Return
```

Example:

```
fun tom (a:arith) (ds:operand->exp) =  
  case a of  
    I(i) => ds(Int i)  
  | Add(a,b) =>  
    let val x = fresh_var() in  
      tom a (fn wa =>  
        tom b (fn wb =>  
          LetVal(x, PrimApp(Plus, [wa, wb]), ds x)))  
    end  
fun tomonadic (a:arith) : exp = tom a Return  
  
tomonadic(I 31) =  
tom(I 31) Return = Return(Int 31)
```

Next Example:

```
fun tom (a:arith) (ds:operand->exp) =
  case a of
    I(i) => ds(Int i)
  | Add(a,b) =>
    let val x = fresh_var() in
      tom a (fn wa =>
        tom b (fn wb =>
          LetVal(x,PrimApp(Plus,[wa,wb]),ds x)))
    end
end
```

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

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

Example Continued:

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

```
tom (I 31) (fn wa =>  
  tom (I 42) (fn wb =>  
    LetVal("x1", PrimApp(Plus, [wa, wb]),  
      Return "x1"))))
```

=

```
tom (I 42) (fn wb =>  
  LetVal("x1", PrimApp(Plus, [Int 31, wb]),  
    Return "x1"))
```

Example Continued:

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

```
tom (I 42) (fn 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.sml for the real code.
- It has to deal with many more cases but has the same basic structure.

```
fun tom (a:arith) (ds:operand->exp) =
  case a of
    I(i) => ds(Int i)
  | Add(a,b) =>
    let val x = fresh_var() in
      tom a (fn wa =>
        tom b (fn wb =>
          LetVal(x, PrimApp(Plus, [wa, wb]), ds x))
        )
    end
  end
fun tomonadic (a:arith) : exp = tom a Return
```