6.035 Project 3

Kyle Miller, Alec Thomson, Patrick Hulin, Steven Valdez

April 9, 2012

1 Overview

The purpose of this project was to add a series of dataflow optimizations to the unoptimized code generator produced at the end of the last project.

We chose to make use of The Higher-Order Optimization Library (Hoopl) for Haskell. This library provides fixed-point functions to interleave dataflow transfer functions and rewrite functions. Both forward and backward analysis are supported by Hoopl. We briefly explain the design and implementation of Hoopl in Section 3. To use Hoopl to perform dataflow analysis optimizations, we first had to dramatically modify both the Mid IR and Low IR used for unoptimized code generation. The changes to the IR are documented in Section 4.

We were able to implement several dataflow optimizations, including Constant Propagation (Section 5), Dead Code Elimination (Section 6), Unreachable Code Elimination (Section 6), Global Common Subexpression Elimination (Section 8), and Global Copy Propagation (Section 9).

2 Division of Work

The work was divided as follows: Kyle Miller rewrote the both the Mid and Low IR to be more compatible with Hoopl and x86 assembly, respectively. Kyle also helped design several optimizations. Alec Thomson used Hoopl to write dataflow optimizations for Constant Propagation, Dead Code Elimination, Global Constant Subexpression Elimination, and Global Copy Propagation. Steven Valdez refactored the Command Line Interface, Main.hs file, and helped implement Unreachable Code Elimination. Patrick Hulin created a testing framework that consisted of many automatic shell scripts and a function to produce valid C code from our IR for additional redundancy checks.

3 Hoopl

Hoopl is a Haskell Dataflow Analysis Library that provides several useful functions to interleave dataflow analysis and control flow graph rewriting. Hoopl requires its users to provide four pieces of information for a given dataflow problem: a *Node Type* defining the types of instructions that can be found inside basic blocks, a *Dataflow Lattice* describing the facts available at program nodes, a *Transfer Function* allowing facts to be updated at a program node, and a *Rewrite function* defining rules for rewriting the graph during rewrite passes.

Hoopl uses this information to interleave analysis and rewriting of a control flow graph. The basic process of Hoopl follows is to start at a provided set of entry points (with a map to initial facts) into the graph, perform the transfer function over a program statement to update the available fact, rewrite that same program statement if necessary, possibly recursively analyze and rewrite the newly rewritten statements, and continue on to the next statement. The Dataflow Lattice provided by the user includes a join function for Hoopl to join facts at merge points and a *bottom* value for Hoopl to use as the default fact to be merged with the incoming fact at a given node. Hoopl allows for both forward and backward analysis of the graphs, with similar information provided for both.

In our case, the node type provided to all of our Hoopl functions is type MidIRInst and represents an instruction in the Mid IR defined in IR2.hs. The dataflow lattices, transfer functions, and rewrite functions provided to Hoopl depend upon the exact dataflow problems at hand and are defined in greater detail in the sections below.

4 Changes to IR to Accomodate Hoopl

We redesigned the IRs so that Hoopl would be able to perform fixed-point computations on our control-flow graphs. We also designed an IR for manipulating control-flow graphs of x86 assembly (instead of manipulating raw strings as we had been doing in the code generator previously).

This involved rebuilding the AST to MidIR conversion and the code generator. Our new code generator is a rule-based system. To support the code generation, we extended the flat expressions to tree expresions. This lets the code generator detect when different addressing modes available to the x86 are being used.

Needless to say, this took a great amount of time, which is unsurprising considering that this was essentially doing the last project over agin.

5 Constant Propagation

Constant propagation was one of the easiest dataflow optimizations to implement using Hoopl. The idea behind constant propagation is to replace as many variables as possible with constants and fold constant expressions together into a single constant. The dataflow lattice, transfer function, and rewrite functions for constant propagation are described below. Constant Propagation is an example of Forward Dataflow Analysis and thus uses Hoopl's forward types.

5.1 Dataflow Lattice for Constant Propagation

The Dataflow lattice for constant propagation is shown in its entirety below:

```
type LitVal = (SourcePos, Int64)
-- Type and definition of the lattice
type ConstFact = Map.Map VarName (WithTop LitVal)
constLattice :: DataflowLattice ConstFact
constLattice = DataflowLattice { fact_name = "Constant Propagation Lattice"
                               , fact_bot = Map.empty
                               , fact_join = joinMaps (extendJoinDomain constFactAdd) }
    where constFactAdd _ (OldFact old) (NewFact new) = if new == old then (NoChange, PElem new)
                                                       else (SomeChange, Top)
```

The fact type used by the lattice is simply a map from variable names to either a known literal value or "Top". The join function for the lattice is a union of the two maps, and the "Bottom" of the lattice is the empty map. This allows the rewrite function to determine whether a variable use can be replaced by a literal at any given point in the graph.

5.2 Transfer Function for Constant Propagation

The transfer function for constant propagation is shown in its entirety below:

```
-- Analysis: variable equals a literal constant
varHasLit :: FwdTransfer Node ConstFact
varHasLit = mkFTransfer ft
    where
      ft :: MidIRInst e x -> ConstFact -> Fact x ConstFact
      ft (Label _ _) f = f
      ft (Enter _ _ args) f = Map.fromList (map (\a -> (a, Top)) args)
      ft (Store _ x (Lit pos k)) f = Map.insert x (PElem (pos, k)) f
      ft (Store _ x _) f = Map.insert x Top f
      ft (IndStore _ _ _) f = f
      ft (Call _ x _ _) f = Map.insert x Top f
      ft (Callout _ x _ _) f = Map.insert x Top f
      ft (Branch _ l) f = mapSingleton l f
      ft (CondBranch _ (Var pos x) tl fl) f
          = mkFactBase constLattice [ (tl, Map.insert x (PElem (pos, bTrue)) f)
                                    , (fl, Map.insert x (PElem (pos, bFalse)) f) ]
      ft (CondBranch _ _ tl fl) f
          = mkFactBase constLattice [ (tl, f)
                                    , (fl, f) ]
      ft (Return _ _) f = mapEmpty
      ft (Fail _) f = mapEmpty
```

The transfer function simply checks to see if a variable is being assigned a literal value. If so, the fact records that that variable corresponds to that literal. If the variable is being assigned any other value (from another variable, a memory location, or the return value of a call or callout), the variable is mapped to "Top" to indicate that its value is unknown. For branches or other nodes that are "closed" on exit, the transfer function provides a fact for each subsequent branch.

5.3 Rewrite Functions for Constant Propagation

Constant Propagation makes use of two rewrite functions, one to replace any literal valued variable uses with the literal they map to, and one to perform "Constant Folding" of expressions. The two rewrite functions are included in their entirety below:

```
-- Rewrite function: replace constant variables
constProp :: forall m. FuelMonad m => FwdRewrite m Node ConstFact
constProp = mkFRewrite cp
    where
      cp :: Node e x -> ConstFact -> m (Maybe (Graph Node e x))
      cp node f
             = return $ liftM insnToG $ mapVN (lookup f) node
      lookup :: ConstFact -> VarName -> Maybe MidIRExpr
      lookup f x = case Map.lookup x f of
                     Just (PElem (pos, v)) -> Just $ Lit pos v
                     _ -> Nothing

-- Rewrite function: Fold constant expressions together
simplify :: forall m f . FuelMonad m => FwdRewrite m Node f
simplify = deepFwdRw simp
    where
      simp :: forall e x. Node e x -> f -> m (Maybe (Graph Node e x))
      simp node _ = return $ liftM insnToG $ s_node node
      s_node :: Node e x -> Maybe (Node e x)
      s_node (CondBranch pos (Lit _ x) tl fl)
          = Just $ Branch pos (if intToBool x then tl else fl)
      s_node n = (mapEN . mapEE) s_exp n
      s_exp (BinOp pos OpDiv expr (Lit _ 0))
          = Nothing
      s_exp (BinOp pos OpMod expr (Lit _ 0))
          = Nothing
      s_exp (BinOp pos op (Lit _ x1) (Lit _ x2))
          = Just $ Lit pos $ (binOp op) x1 x2
      s_exp (UnOp pos op (Lit _ x))
          = Just $ Lit pos $ (unOp op) x
      s_exp (Cond pos (Lit _ x) expt expf)
          = Just $ (if intToBool x then expt else expf)
      s_exp (Cond pos _ expt expf)
          | expt == expf  = Just expt
      s_exp _ = Nothing
      binOp OpAdd = (+)
      binOp OpSub = (-)
      binOp OpMul = (*)
      binOp OpDiv = div
      binOp OpMod = rem
      binOp CmpLT = \x y -> boolToInt $ x < y
      binOp CmpGT = \x y -> boolToInt $ x > y
      binOp CmpLTE = \x y -> boolToInt $ x <= y
      binOp CmpGTE = \x y -> boolToInt $ x >= y
      binOp CmpEQ = \x y -> boolToInt $ x == y
      binOp CmpNEQ = \x y -> boolToInt $ x /= y
      unOp OpNeg = negate
      unOp OpNot = boolToInt . not . intToBool
```

The rewrite function provided to Hoopl is the composition of these two rewrite functions that effectively performs the Constant Propagation Optimization.

6 Dead Code Elimination

Dead Code Elimination was a relatively simple backwards dataflow optimization. The idea was to move backwards through the control flow graph, recording the uses of variables. If the rewrite function comes across an assignment to a variable that has no uses in the future, the assignment can be eliminated from the control flow graph. The dataflow lattice, transfer function, and rewrite function of Dead Code Elimination are described below.

6.1 Dataflow Lattice for Dead Code Elimination

```
type Live = S.Set VarName
liveLattice :: DataflowLattice Live
liveLattice = DataflowLattice { fact_name = "Live variables"
                              , fact_bot = S.empty
                              , fact_join = add
                              }
    where add _ (OldFact old) (NewFact new) = (ch, j)
              where  j = new ‘S.union‘ old
                     ch = changeIf (S.size j > S.size old)
```

As described briefly above, the fact type for Dead Code Elimination is simply a set of variable names representing the “Live” variables at a given program point. The join function is again a simple union and the “Bottom” of the lattice is the empty set.

6.2 Transfer Function for Dead Code Elimination

```
liveness :: BwdTransfer MidIRInst Live
liveness = mkBTransfer live
    where live :: MidIRInst e x -> Fact x Live -> Live
          live (Label _ _) f = f
          live n@(Enter _ _ args) f = addUses (f S.\\ (S.fromList args)) n
          live n@(Store _ x _) f = addUses (S.delete x f) n
          live n@(IndStore _ _ _) f = addUses f n
          live n@(Call _ x _ _) f = addUses (S.delete x f) n
          live n@(Callout _ x _ _) f = addUses (S.delete x f) n
          live n@(Branch _ l) f = addUses (fact f l) n
          live n@(CondBranch _ _ tl fl) f = addUses (fact f tl ‘S.union‘ fact f fl) n
          live n@(Return _ _) _ = addUses (fact_bot liveLattice) n
          live n@(Fail _) _ = addUses (fact_bot liveLattice) n

          fact :: FactBase Live -> Label -> Live
          fact f l = fromMaybe S.empty $ lookupFact l f

          addUses :: Live -> MidIRInst e x -> Live
          addUses = fold_EN (fold_EE addVar)
          addVar s (Var _ v) = S.insert v s
          addVar s _ = s
```

The transfer function makes use of an auxiliary function called “addUses” to gather every variable use from a given program statement and add those variables to the set of live variables. The “fold_EN” function performs a fold operation over the expressions in a node and is defined along with several other auxiliary functions in OptSupport.hs.

6.3 Rewrite Function for Dead Code Elimination

```
deadAsstElim :: forall m . FuelMonad m => BwdRewrite m MidIRInst Live
deadAsstElim = mkBRewrite d
    where
      d :: MidIRInst e x -> Fact x Live -> m (Maybe (Graph MidIRInst e x))
      d (Store _ x _) live
          | not (x ‘S.member‘ live) = return $ Just emptyGraph
      d _ _  = return Nothing
```

As you can see, the rewrite function for Dead Code Elimination is very simple and only looks for assignments to dead variables, which it then eliminates by returning an empty graph.

7 Empty Block Elimination

Empty Block Elimination, another relatively simple backward analysis, was used to work backwards and skip over Empty Blocks who only served to jump the dataflow onto another block. After analyzing the lattice, which consisted of the previously seen Label, the optimization rewrote any conditional and unconditional branches to skip over any direct empty blocks, in order to reduce code size.

7.1 Dataflow Lattice for Empty Block Elimination

```
type LastLabel = Maybe Label
lastLabelLattice :: DataflowLattice LastLabel
lastLabelLattice = DataflowLattice { fact_name = "Last Labels"
                                   , fact_bot = Nothing
                                   , fact_join = add
                                   }
  where add _ (OldFact o) (NewFact n) = (c, n)
          where c = changeIf (o /= n)
```

7.2 Transfer Function for Empty Block Elimination

```
lastLabelness :: BwdTransfer MidIRInst LastLabel
lastLabelness = mkBTransfer f
  where f :: MidIRInst e x -> Fact x LastLabel -> LastLabel
        f (Branch _ l) k =
          case lookupFact l k of
            Just l’ -> l’ ‘mplus‘ Just l
            Nothing -> Just l
        f (Label _ l) (Just l')
            | l == l'  = Nothing
            | otherwise = Just l'
        f (Label _ l) Nothing = Nothing
        f (Enter _ l _) k = Nothing

        f _ k = Nothing
```

7.3 Rewrite Function for Empty Block Elimination

```
lastLabelElim :: forall m . FuelMonad m => BwdRewrite m MidIRInst LastLabel
lastLabelElim = deepBwdRw ll
  where
    ll :: MidIRInst e x -> Fact x LastLabel -> m (Maybe (Graph MidIRInst e x))
    ll (Branch p l) f =
      return $ case lookupFact l f of
          Nothing -> Nothing
          Just mm -> mm >>= (Just . mkLast . (Branch p))
    ll (CondBranch p ce tl fl) f
        | tl == fl = return $ Just $ mkLast $ Branch p tl
        | otherwise =
            return $ do tl' <- fun tl
                        fl' <- fun fl
                        guard $ [tl', fl'] /= [tl, fl]
                        Just $ mkLast $ CondBranch p ce tl' fl'
        where
          fun :: Label -> Maybe Label
          fun l = fromJust (lookupFact l f) `mplus` (Just l)
--    ll (Enter p l args) (Just l')
--        = return $ Just (mkFirst (Enter p l args)
--                            <*> mkLast (Branch p l'))
    ll _ f = return Nothing
```

8 Global Common Subexpression Elimination

Global Common Subexpression Elimination, another forward analysis, was one of the most complex dataflow optimizations performed by our compiler. To get Global CSE working with Hoopl, we had to perform a few preliminary steps. First, the control flow graph needed to be "Flattened" so that any expression encountered by the CSE analysis would either be of the form "BinOp VarOrLit VarOrLit", "UnOp VarOrLit", or "VarOrLit". We performed this step in a simple Hoopl forward analysis and rewrite defined in Flatten.hs. Next, because Hoopl interleaves rewrites followed by analysis of those rewrites, we needed a way for the CSE transfer and rewrite functions to distinguish between "Non Temporary" variables that existed before the CSE analysis began, and "Temporary" variables introduced by the CSE analysis. To do this, we performed another forward analysis of the graph with no rewrite function that returned a set of all the variable names contained in the graph. That set is then given to the CSE transfer function and rewrite function at creation time so the functions can easily distinguish between temp and non-temp variables.

With these steps complete, the CSE analysis works by replacing an expression assignment to a non-temp variable with an expression assignment to a temp variable followed copying that temp variable to the original non-temp variable. For example:

The expression:

```
x := y + z
```

becomes

```
t1 := y + z
x := t1
```

The transfer function then handles mapping available expressions to the temps they're stored in and the rewrite function uses that information to replace common subexpressions with references to temps. The complete dataflow lattice, transfer function, and rewrite function for Global CSE are described below.

8.1 Dataflow Lattice for CSE

```
type ExprFact = WithBot (Map.Map MidIRExpr VarName)
exprLattice :: DataflowLattice ExprFact
exprLattice = DataflowLattice { fact_name = "Global CSE Lattice"
                              , fact_bot = Bot
                              , fact_join = intersectMaps }
    where intersectMaps :: Label -> OldFact ExprFact -> NewFact ExprFact -> (ChangeFlag, ExprFact)
          intersectMaps _ (OldFact old) (NewFact new)
              = case (old, new) of
                  (old', Bot) -> (NoChange, old')
                  (Bot, new') -> (SomeChange, new')
                  (PElem oldMap, PElem newMap) -> (ch, PElem j)
                      where j = Map.mapMaybeWithKey f oldMap
                            f k v = case Map.lookup k newMap of
                                      Just v' -> if v == v'
                                                 then Just v
                                                 else Nothing
                                      Nothing -> Nothing
                            ch = changeIf (Map.size j /= Map.size oldMap)
```

The fact type for CSE is a map from available expressions to the temporary variables that contain the computed values of the expressions. Since the bottom for available expression analysis is the set of all possible expressions in the program, we define the map type as "WithBottom" so we have a distinct "Bottom" value to represent this theoretical set. The join function for the dataflow lattice is then the intersection of two maps, with identical expressions being removed if they map to separate temporaries.

8.2 Transfer Function for CSE

```
exprAvailable :: S.Set VarName -> FwdTransfer MidIRInst ExprFact
exprAvailable nonTemps = mkFTransfer ft
    where
      ft :: MidIRInst e x -> ExprFact -> Fact x ExprFact
      ft (Label _ _) f = f
      ft (Enter _ _ args) f = foldl (flip invalidateExprsWith) f args
      ft (Store _ x expr) f = handleAssign x expr f
      ft (IndStore _ _ _) f = destroyLoads f
      ft (Call _ x _ _) f = invalidateExprsWith x f
      ft (Callout _ x _ _) f = invalidateExprsWith x f
      ft (Branch _ l) f = mapSingleton l f
      ft (CondBranch _ _ tl fl) f
          = mkFactBase exprLattice [ (tl, f)
                                   , (fl, f) ]
      ft (Return _ _) f = mapEmpty
      ft (Fail _) f = mapEmpty
      handleAssign :: VarName -> MidIRExpr -> ExprFact -> ExprFact
      handleAssign x expr f = if isTemp nonTemps x
                              then newFact
                              else case expr of
                                     (Var pos varName)
                                         | isTemp nonTemps varName
                                       -> invalidateExprsWith x f
                                     _ -> f
          where newFact = PElem newMap
                newMap = Map.insert expr x lastMap
                lastMap :: Map.Map MidIRExpr VarName
                lastMap = case f of
                            Bot -> Map.empty
                            PElem oldMap -> oldMap
      invalidateExprsWith :: VarName -> ExprFact -> ExprFact
      invalidateExprsWith _ Bot = Bot
      invalidateExprsWith x (PElem oldMap) = PElem newMap
          where newMap = Map.mapMaybeWithKey f oldMap
                f k v = if containsVar x k
                        then Nothing
                        else Just v
      destroyLoads :: ExprFact -> ExprFact
      destroyLoads Bot = Bot
      destroyLoads (PElem oldMap) = PElem newMap
          where newMap = Map.mapMaybeWithKey f oldMap
                f k v = if containsLoad k
                        then Nothing
                        else Just v
```

The transfer function primarily looks for temps and maps any expressions they correspond to into the map of available expressions. The transfer function also handles removing any available expressions that get invalidated by a variable being assigned to or memory being written (which invalidates load expressions).

8.3 Rewrite Function for CSE

```
cseRewrite :: forall m. (FuelMonad m, UniqueNameMonad m) => S.Set VarName
-> FwdRewrite m MidIRInst ExprFact
cseRewrite nonTemps = deepFwdRw cse
    where
      cse :: MidIRInst e x -> ExprFact -> m (Maybe (Graph MidIRInst e x))
      cse (Store _ x (Var _ v)) f
          | isTemp nonTemps v = return Nothing
      cse (Store pos x expr) f
          | not $ isTemp nonTemps x
          = case lookupExpr expr f of
              Just varName -> return $ Just $ insnToG $ Store pos x (Var pos varName)
              Nothing -> do tempName <- genUniqueName "cse"
                            let tempAssign = insnToG $ Store pos (MV tempName) expr
                                varAssign = insnToG $ Store pos x (Var pos (MV tempName))
                                newGraph = tempAssign <*> varAssign
                            return $ Just newGraph
      cse _ f = return Nothing
```

The rewrite function performs two duties. First, if it encounters an assign to a non-temporary by anything other than a temporary variable, it creates a new temporary representing the assignment expression and replaces the original assignment to the non-temporary with an assignment from the new temporary. The second duty of the rewrite function is to see if an available expression is being mapped to a non-temporary variable, and if so replace the available expression with the temporary that contains its computed value.

9 Global Copy Propagation

Global Copy Propagation was implemented in a very similar way to constant propagation without the need for any deep folding rewrite function. Another forward analysis, the idea of copy propagation is to record definitions of variables that are simply copies from other variables and replace any use of the new variable that the definition reaches with the old variable. The dataflow lattice, transfer function, and rewrite function for Global Gopy Propagation are included in their entirety below.

9.1 Dataflow Lattice for Copy Propagation

```
type VarInfo = (SourcePos, VarName)
type CopyFact = Map.Map VarName (WithTop VarInfo)
copyLattice :: DataflowLattice CopyFact
copyLattice = DataflowLattice { fact_name = "Copy Propagation Lattice"
                              , fact_bot = Map.empty
                              , fact_join = joinMaps (extendJoinDomain copyFactAdd) }
```

```
    where copyFactAdd _ (OldFact old) (NewFact new) = if new == old then (NoChange, PElem new)
                                                      else (SomeChange, Top)
```

The fact type for Copy Propagation is a map from a variable name to another variable name that variable is a copy of. As in constant propagation, the bottom is an empty map and the join function is a simple map union.

9.2 Transfer Function for Copy Propagation

```
-- Analysis: Whether a variable is a copy of another at a given point
varIsCopy :: FwdTransfer MidIRInst CopyFact
varIsCopy = mkFTransfer ft
    where
      ft :: MidIRInst e x -> CopyFact -> Fact x CopyFact
      ft (Label _ _) f = f
      ft (Enter _ _ args) f = Map.fromList (map (\a -> (a, Top)) args)
      ft (Store _ x (Var pos v)) f = removeBindingsTo x $ Map.insert x (PElem (pos, v)) f
      ft (Store _ x _) f = removeBindingsTo x $ Map.insert x Top f
      ft (IndStore _ _ _) f = f
      ft (Call _ x _ _) f = removeBindingsTo x $ Map.insert x Top f
      ft (Callout _ x _ _ ) f = removeBindingsTo x $ Map.insert x Top f
      ft (Branch _ l) f = mapSingleton l f
      ft (CondBranch _ _ tl fl) f
             = mkFactBase copyLattice [ (tl, f)
                                      , (fl, f) ]
      ft (Return _ _) f = mapEmpty
      ft (Fail _) f = mapEmpty
      removeBindingsTo :: VarName -> CopyFact -> CopyFact
      removeBindingsTo x oldMap = newMap
          where newMap = Map.mapMaybe f oldMap
                f (PElem (pos, v)) = if v == x
                                     then Nothing
                                     else Just $ PElem (pos, v)
                f v = Just v
```

The transfer function performs two duties. First, if it encounters a copy from one variable to another, it adds a mapping between the variables to the new fact. Second, if it encounters an assignment to a variable that exists in the map already (whether as a key or a value), it invalidates the mapping.

9.3 Rewrite Function for Copy Propagation

```
-- Rewrite function: Replace copied variables with the original version
copyProp :: forall m. FuelMonad m => FwdRewrite m MidIRInst CopyFact
copyProp = mkFRewrite cp
    where
      cp :: MidIRInst e x -> CopyFact -> m (Maybe (Graph Node e x ))
      cp node f
             = return $ liftM insnToG $ mapVN (copyLookup f) node
      copyLookup :: CopyFact -> VarName -> Maybe MidIRExpr
      copyLookup f x = case Map.lookup x f of
                         Just (PElem (pos, v)) -> Just $ Var pos v
                         _ -> Nothing
```

The rewrite function for Copy Propagation is almost identical to the initial rewrite for constant propagation. If it encounters a use of a variable that it has a mapping for in its fact, it simply replaces the variable with the original copy.

10 Testing

We created a small testing system to allow for rapid creation of tests. The syntax is a subset of Decaf, in that the desired output is contained within comments of the program. The Hello World example is shown below.

```
//> Hello, World!

class Program {
  void main() {
    callout("printf", "Hello, World!\n");
  }
}
```