

The expressions we want to support

```
data Exp =
  Num Int           -- constants
| Dat Int [Exp]    -- nameless constructors
| Swi Exp [(Int, [String], Exp)] -- pattern matching
| BOp OpKind Exp Exp -- arithmetic operators
| Var String       -- variables
| App Exp Exp      -- application
| Lam String Exp   -- abstraction

data SCS = Map String ([String], Exp) -- super-combinators

data OpKind = Add | Sub | Mul | Div
```

We start with ...

```
data Exp =
    Num Int           -- constants
  | Var String       -- variables
  | App Exp Exp      -- application

type SCS = Map String ([String], Exp) -- super-combinators

parseSCS :: Parser (Map String ([String], Exp))
```

Example

```
f x y = x ;
main = f 1 2 ;
```

The “Three instruction machine”

```
type CPtr = Int           -- a code pointer
type HPtr = Int           -- a heap pointer

data Data =
  FObj CPtr [HPtr]        -- a function object
| IObj Int                -- a constant

data MS = MS
  { frame :: [HPtr]       -- arguments
  , stack :: [HPtr]       -- stack
  , pc    :: [Instr]      -- prog. counter
  , heap  :: Map HPtr Data -- heap
  , count :: Int          -- counter for heap objects
  , code  :: Map CPtr [Instr] -- code (does not change)
  }
```

Example

Running

■ `f x y = g x x`

may have the following state before jumping to the body of `g`:

```
{ frame = [xp, yp]
, stack = [xp, xp]
, pc     = "call to gc" : ...
, heap  = { xp -> FObj xc xenv, yp -> FObj yc yenv, ...}
, count = ...
, code  = { gc -> ..., xc -> ..., yc -> ..., ...}
}
```

The TIM monad

```
type TimM a = StateT MS IO a
```

All MS fields `x` have a `setX`, `getX`, and `modifyX`.

```
x          :: A
getX      :: TimM a
setX      :: A -> TimM ()
modifyX   :: (A -> A) -> TimM ()
```

There is also

- `createId` :: TimM Int
- `findHeap` :: HPtr -> TimM Data
- `findCode` :: CPtr -> TimM Data

Details ...

```
createId = do  
  modifyCount (1+)  
  getCount
```

```
findHeap n = do  
  hp <- getHeap  
  return $ (M.!) hp n
```

```
findCode n = do  
  bls <- getCode  
  return $ (M.!) bls n
```

Running the program

```
type TimM a = StateT MS IO a

step :: Instr -> TimM ()
step = ...

run :: TimM ()
run = do
  instr <- getPC
  case instr of
    []   -> return ()
    i:is -> do
      setPC is
      step i      -- Note: may overwrite PC
      run
```

Example 2.0 – intuition

compose2 $f\ g\ x = f\ (g\ x\ x)$

```
{ frame = [fp, gp, xp]
, heap = { fp -> FObj fc fenv
          , gp -> FObj gc yenv
          , xp -> FObj xc xenv
          , ... }
}
```

- fp, gp, and xp are dynamic arguments
- the FObj-s represent λ -terms
- fc, gc, and xc are pointers to code
 - ... sub-terms of the program

Example 2.1 – argument is prepared

compose2 $f\ g\ x = f\ (g\ x\ x)$
w $f\ g\ x = g\ x\ x$

```
{ frame = [fp, gp, xp]
, stack = [ wp ]
, heap = { fp -> FObj fc fenv
          , gp -> FObj yc yenv
          , xp -> FObj xc xenv
          , wp -> FObj wc [fp, gp, xp]
          , ...}
}
```

Example 2.2 – function f is called

compose2 $f\ g\ x = f\ (g\ x\ x)$
w $f\ g\ x = g\ x\ x$

```
{ frame = [ wp ]
, stack = []
, heap = { fp -> FObj fc fenv
          , gp -> FObj yc yenv
          , xp -> FObj xc xenv
          , wp -> FObj wc [fp, gp, xp]
          , ... }
, pc = fc
}
```

Instructions

To implement these transitions we need to

- 1 push values onto the stack,
- 2 jump into the function, and
- 3 move values from the stack to the frame.

```
data Instr =
  Push Addr      -- push values onto the stack
| Enter Addr     -- jump into the function
| Take Int       -- move values from the stack to the frame

data Addr =
  Arg Int        -- an argument (from the frame)
| Label CPtr    -- a constant code pointer
| IntConst Int  -- an integer constant
```

Actually ...

A frame is a pointer to the heap, where the arguments can be found in a FObj. This will become important later. Now it is just an indirection to the args.

```
data MS = MS
  { frame :: HPtr           -- not [HPtr]
  , stack :: [HPtr]
  , pc    :: [Instr]
  , heap  :: Map HPtr Data
  , code  :: Map CPtr [Instr]
  , count :: Int
  }
```

*) there are more omissions ...

Implementation

```
step (Push (Arg k)) = do  
  x ← findFrame k  
  modifyStack (x :)
```

```
step (Push (IntConst n)) = do  
  ip ← createId  
  modifyHeap (M.insert ip $ IObj n)  
  modifyStack (ip :)
```

```
findFrame n = do  
  fr ← getFrame  
  FObj _ fr ← findHeap fr  
  return $ fr !! n
```

Implementation

```
step (Push (Label l)) = do
  np      <- createId
  fr      <- getFrame
  FObj _ as <- findHeap fr
  modifyHeap (M.insert np $ FObj l as)
  modifyStack (np :)
```

```
findHeap n = do
  hp <- getHeap
  return $ (M.!) hp n
```

Implementation

```
step (Take n) = do
  st      <- getStack
  fr      <- getFrame
  FObj c _ <- findHeap fr
  modifyHeap (M.insert fr $ FObj c (take n st))
  modifyStack (drop n)
```

```
step (Enter (Label l)) = do
  c      <- findCode l
  np     <- createId
  fr     <- getFrame
  FObj _ as <- findHeap fr
  modifyHeap (M.insert np $ FObj l as)
  setFrame np
  setPC c
```

```
findCode n = do
  bls <- getCode
  return $ (M.!) bls n
```

```
step (Enter (Arg k)) = do
  fa      <- findFrame k
  FObj n f <- findHeap fa
  c       <- findCode  n
  setFrame fa
  setPC c
```

```
step (Enter (IntConst n)) = do
  ip <- createId
  modifyHeap (M.insert ip $ IObj n)
  modifyStack (ip :)
```


Compiling by hand.

- `f x y = g x x;`

```
| f = [ Take 2, Push (Arg 0), Push (Arg 0)  
      , Enter (Label gc) ]
```

Compiling by hand.

- $f\ x\ y = g\ x\ x;$

```
f = [ Take 2, Push (Arg 0), Push (Arg 0)
      , Enter (Label gc) ]
```

- $w\quad f\ g\ x = g\ x\ x;$
compose2 $f\ g\ x = f\ (g\ x\ x);$

```
w = [ Take 3, Push (Arg 2)
      , Push (Arg 2), Enter (Arg 1) ]

w' = [ Push (Arg 2), Push (Arg 2)
       , Enter (Arg 1) ]

compose2 = [ Take 3, Push (Label w' c)
             , Enter (Arg 0) ]
```

Basic structure of the compiler

- `compile` :: `Exp` -> [`Instr`]

Basic structure of the compiler

- `compile :: Exp -> [Instr]`
- `compile :: Exp -> (CPtr, Map CPtr [Instr])`

Basic structure of the compiler

- `compile :: Exp -> [Instr]`
- `compile :: Exp -> (CPtr, Map CPtr [Instr])`
- monadic computation

The Compile Monad

```
data CompState = CompState      -- current:
  { instr :: [Instr]           -- instruction gen. buffer
  , cur   :: CPtr              -- current sc. id
  , env   :: Map String Addr   -- var to addr mapping
  , scs   :: Map CPtr [Instr]  -- sc-s; equivalent of code
  , cnt   :: Int               -- counter for creating new sc
  }

type CompileM a = StateT CompState IO a
```

1 Compile a program

```
compileSCS :: Map String ([String], Exp) -> CompileM ()
```

2 Compile a single super-combinator

```
oneSC :: String -> ([String], Exp) -> CompileM ()
```

3 Compute the result of an expression

```
compileR :: Exp -> CompileM ()
```

4 Compile an argument expression

```
compileL :: Exp -> CompileM Addr
```

```
compileSCS :: Map String ([String], Exp) -> CompileM ()  
compileSCS m = do  
  mapM_ addScToEnv (M.keys m)  
  iterM_ oneSC m  
  
addScToEnv :: String -> CompileM ()  
addScToEnv x = do  
  i <- newID  
  addEnv x (Label i)  
  
iterM_ :: Monad m => (a -> b -> m c) -> Map a b -> m ()  
iterM_ f = M.foldWithKey comb (return ())  
  where comb k v r = r >> f k v >> return ()
```



```
oneSC :: String -> ([String], Exp) -> CompileM ()
```

```
oneSC name (args, body) = do
  addArgs args
  Label lab <- findEnv name
  flushInstr
  setCur lab
  setInstr [(Take $ length args)]
  compileR body
  flushInstr
```

```
addArgs :: [String] -> CompileM ()
```

```
addArgs = addArgsAcc 0
  where addArgsAcc i [] = return ()
        addArgsAcc i (n:ns) = do
          addEnv n (Arg i)
          addArgsAcc (i+1) ns
```

```
compileR :: Exp -> CompileM ()  
compileR (Var n) = do  
  a <- findEnv n  
  addInstr $ Enter a  
  
compileR (App f x) = do  
  a <- compileL x  
  addInstr $ Push a  
  compileR f
```

```
compileL :: Exp -> CompileM Addr
```

```
compileL (Var n) =  
  findEnv n
```

```
compileL i = do
```

```
  flushInstr
```

```
  fo <- getCur
```

```
  fn <- newID
```

```
  setCur fn
```

```
  compileR i
```

```
  flushInstr
```

```
  setCur fo
```

```
  return $ Label fn
```

```
compToTimState :: CompState -> MS
compToTimState cs = MS
  { frame = 0
  , stack = []
  , pc    = reverse $ (M.!) (scs cs) main_l
  , heap  = M.insert 0 (FObj main_l []) M.empty
  , code  = scs cs
  , count = cnt cs
  }
where
  Label main_l = (M.!) (env cs) "main"
```

Putting it all together

```
main :: IO ()
main = do
  file:_ <- getArgs
  fc     <- readFile file
  let sc = parseResult parseSCS fc
  s      <- execStateT (compileSCS sc) startCompState
  let ts = compToTimState s
  s2     <- execStateT run ts
  printf "main = %s\n" (show $ returnValue s2)
where
  returnValue ms =
    case M.lookup (head $ stack ms) (heap ms) of
      Just n -> n
      _     -> undefined
```

Mapping to common hardware

```
data MS = MS
  { frame :: HPtr           -- Data *frame;
  , stack :: [HPtr]        -- Data *stack;
  , count :: Int           -- int count;
  , heap  :: Map HPtr Data  -- memory: 0 .. count
  , pc    :: [Instr]       -- the pc/ip register
  , code  :: Map CPtr [Instr] -- int *C;
  }
```

- Instructions can be converted to x86 instructions.
- Program starts by jumping to `C[m]`.
- Only GC is not doable as single instruction.

Additional features: Lambda abstraction

```

data Exp =
  Num Int           -- constants
| Var String       -- variables
| App Exp Exp      -- application
| Lam String Exp   -- abstraction

data SCS = Map String ([String], Exp) -- super-combinators

```

Example

```
f x = g (\ y -> add y x) 2;
```

1 Transform into super-combinators

```
f x = g (f' x) 2;
```

```
f' x y = add y x;
```

2 or, extend FObj-s.

Additional features: arithmetic operators

```
data Exp =
  Num Int           -- constants
| BOp OpKind Exp Exp -- arithmetic operators
| Var String       -- variables
| App Exp Exp      -- application

data SCS = Map String ([String], Exp) -- super-combinators

data OpKind = Add | Sub | Mul | Div
```

Example

```
f = x * (1 + y);
```

- First normalize x , then $1 + y$. Then add.


```
compileR (BOp op x y) = do
  compileS y
  compileS x
  addInstr $ BOp op
```

```
data Instr =
  ...
  | BOp OpKind
```

- Add the normal value on the stack

```
compileS :: Exp -> CompileM ()
```

```
step (BinOp op) = do
  x:y:st <- getStack
  IObj xv <- findHeap x
  IObj yv <- findHeap y
  n      <- createId
  modifyHeap (M.insert n $ IObj $ doOp op xv yv)
  setStack (n:st)
where
  doOp Add = (+)
  doOp Sub  = (-)
  doOp Mul  = (*)
  doOp Div  = div
```

```
compileS :: Exp -> CompileM ()
```

```
compileS (Num i) =  
  addInstr $ Push (IntConst i)
```

```
compileS (BOp op x y) = do  
  compileS y  
  compileS x  
  addInstr $ BinOp op
```

Var and App cases

```
compileS x = do
  n <- newID
  addInstr $ Stash n
  compileR x
  flushInstr
  setCur n
```

```
data Instr =
  ...
  | Stash CPtr
  | Retrieve
```

- **Append** `Retrieve` to every code section.

```
data DumpElem =
    Continue (HPtr, CPtr, [HPtr])

data MS = MS
  { ...
  , dump  :: [DumpElem]
  }

step (Stash c) = do
  fr <- getFrame
  st <- getStack
  modifyDump (Continue (fr, c, st) : )
  setStack []
```

```
step Retrieve = do
  dmp <- getDump
  case dmp of
    Continue (fr, c, st) : dmp -> do
      r:_ <- getStack
      c' <- findCode c
      setFrame fr
      setPC c'
      setStack (r:st)
      setDump dmp
    _ ->
      setPC []
```

What about applicative order?

- `compileS` is the usual case, not `compileR`
- `compileS` for variables can sometimes be optimized
 - `f x = g (2+x);`
- Too few arguments — modify `step` (Take `n`)
 - `f h = h 2;`
 - `g x y = x + y;`
 - `main = f (g 3);`

Additional features: algebraic data types

```
data Exp =  
  ...  
  | Dat Int [Exp]  
  | Swi Exp [(Int, [String], Exp)]
```

Example

```
x = <1> 2;  
f = case x of  
  <0>   -> 0  
  | <1> z -> z+1 ;
```



```
data Data =
    FObj CPtr [HPtr]
  | IObj Int
  | CObj Int [HPtr]
```

```
data Instr =
    ...
  | Case Int CPtr
  | Data Int Int
```

```
compileR (Dat n as) = do
  cas <- mapM compileL as
  mapM_ (addInstr . Push) $ reverse cas
  addInstr $ Data n $ length as
```

```
step (Data n m) = do
  st <- getStack
  dp <- createId
  modifyHeap (M.insert dp $ CObj n (take m st))
  modifyStack (:) dp . drop m
```

Intuition

Example

```
x = <1> 2;
```

```
f = case x of
```

```
  <0>  -> 0
```

```
  | <1> z -> z+1 ;
```

```
xc = [Push (IntConst 2), Data 1 1, Retrieve]
```

```
fc = ``compileS (Var x)`` ++ [  
  Case 0 c0,  
  Case 1 c1  
]
```

```
c0 = [Push (IntConst 0), Retrieve]
```

```
c1 = [Push (IntConst 1) ] ++  
  ``compileS (Var z)`` ++  
  [BinOp Add, Retrieve]
```

```
step (Case i lc) = do
  x:_      <- getStack
  CObj i' ap <- findHeap x
  if i==i' then do
    cp      <- findCode lc
    fr      <- getFrame
    FObj c f <- findHeap fr
    modifyHeap (M.insert fr $ FObj c (f++ap))
    setPC cp
  else return ()

compileR (Swi e cs) = do
  compileS e
  mapM_ caseR cs

compileS (Dat n as) = do
  cas <- mapM compileL as
  mapM_ (addInstr . Push) $ reverse cas
  addInstr $ Data n $ length as
```

```

caseR (i, as, e) = do
  oc    <- getCur
  oe    <- getEnv
  nc    <- newID
  let ma = M.fold maxarg (-1) oe
  addInstr $ Case i nc
  flushInstr
  setCur nc
  modifyEnv $ addArgs as (ma+1)
  compileR e
  flushInstr
  setCur oc
  setEnv oe

where
  addArgs [] _ e = e
  addArgs (x:xs) m e = addArgs xs (m+1) $ M.insert x (Arg m) e

  maxarg (Arg n) y = max n y
  maxarg _ y = y

```

Additional features: graph reduction

Example (Haskell)

```
import System.IO.Unsafe
```

```
f :: Int -> Int
```

```
f x = x+x
```

```
main = print $ f (unsafePerformIO $ putStrLn "!" >> return 1)
```

```
*Main> main
```

```
!
```

```
2
```

- normal vs. applicative order
- tree vs. graph reduction

```
data Instr =  
    ...  
    | UpdateMarker  
  
data DumpElem =  
    Continue (HPtr, CPtr, [HPtr])  
    | Update  
  
step UpdateMarker = do  
    modifyDump (Update : )  
    return ()
```

```
step Retrieve = do
  dmp <- getDump
  case dmp of
    Continue _ -> do
      ...
    Update :dmp -> do
      fr  <- getFrame
      vp:_ <- getStack
      v   <- findHeap vp
      modifyHeap (M.insert fr v)
      setDump dmp
      modifyPC (Retrieve :)
  _ ->
    ...
```

- This is the reason why we defined `frame` as we did.


```
compileL (Num i) =  
  return $ IntConst i
```

```
compileL (Var n) =  
  findEnv n
```

```
compileL i = do  
  fo <- getCur  
  fn <- newID  
  flushInstr  
  setCur fn  
  addInstr $ UpdateMarker  
  compileR i  
  flushInstr  
  setCur fo  
  return $ Label fn
```

- Some `FObj`-s will be now overwritten ...

```
step (Enter (Arg k)) = do
  fa <- findFrame k
  fo <- findHeap fa
  case fo of
    FObj n f -> do
      setFrame fa
      c <- findCode n
      setPC c
    _ ->
      modifyStack ( fa : )
```

Graph reduction

- Original TIM put update markers on the `stack`.
- This allows to decide if to update on the caller side.
- Complicated and ugly design.
- Unclear if the optimization is that important.

Summary

- Compile super-combinators into instructions.
- Few instructions — each reasonably simple.
- Low-level: computation on the stack.
- High-level: mostly of re-arranging `Data`-stacks.
- Efficiency loss compared to C:
 - boxed values
 - small functions/lot of jumping around
 - function parameters on a stack instead of registers
 - generating a lot of heap objects