A Compiler for a Stack-Based Language

V. E. McHale

September 25, 2021

Abstract
We present a compiler for a stack-based language targeting assembly, written in the functional style.

Contents

1 Introduction 2
  1.1 Compiler ......................................................... 2

2 Language 2
  2.1 Type System ................................................... 3
  2.2 Pattern Matching .............................................. 3
  2.3 ABI ............................................................. 3

3 Frontend 4
  3.1 Lexer ............................................................ 4
    3.1.1 Interning Identifiers .................................... 4
  3.2 Happy .......................................................... 4
  3.3 Type Assignment .............................................. 4
    3.3.1 Annotation ................................................ 5
    3.3.2 Renamer .................................................... 5
    3.3.3 Unorthodoxies .......................................... 5
  3.4 Pattern-Match Exhaustiveness Checker ..................... 5
  3.5 Monomorphization ............................................. 5
    3.5.1 Inliner ..................................................... 5

4 Backend 6
  4.1 Intermediate Representation ............................... 6
    4.1.1 Sizing .................................................... 8
    4.1.2 Byte Copying ............................................. 8
    4.1.3 Optimization ............................................ 8
  4.2 Abstract Assembly ........................................... 8
  4.3 Instruction Selection ...................................... 9
  4.4 Register Allocation ........................................ 10
    4.4.1 Control Flow ............................................ 10
    4.4.2 Liveness Analysis ...................................... 10
    4.4.3 Linear Allocator ....................................... 11
1 Introduction

Stack-based languages are obscure with little attention to implementation in the literature, though standard techniques work. Here I present a compiler front to back, pointing to algorithms and sources.

This paper is also part confession: monomorphization as I have done it is sublunary.

1.1 Compiler

The compiler is implemented in Haskell, depending heavily on the containers library \[1\] for immutable data structures. It has both an Aarch64 backend and an x86-64 backend and can be used as a cross-compiler.

All code for the compiler can be found on Hackage: \[\text{https://hackage.haskell.org/package/kempe}\].

2 Language

The source language is part of the reason our compiler will be so concise. It is a stack-based language made friendlier by static types (à la Joy \[12\]).

\[
gcd : \text{Int Int } \rightarrow \text{Int}
\]
\[
=:\begin{array}{l}
\text{[ dup 0 =}
\text{ } \\
\text{\hspace{1cm}if( drop}
\text{ } \\
\text{\hspace{2cm}, dup dip(%) swap gcd )}
\text{ ]}
\end{array}
\]

Everything in Kempe is unoriginal \[6, 7\] if arcane.
2.1 Type System

Kempe supports algebraic (non-recursive) data types, including higher-kinded types, as well as a few integer types (Int, Word). Types can be universally quantified.

Function application is replaced with concatenation, viz.

\[
\frac{\Gamma \vdash x : \alpha_1 \cdots \alpha_n - \beta_1 \cdots \beta_m \gamma_1 \cdots \gamma_k \quad \Gamma \vdash y : \gamma_1 \cdots \gamma_k - \delta_1 \cdots \delta_l}{\Gamma \vdash xy : \alpha_1 \cdots \alpha_n - \beta_1 \cdots \beta_m \delta_1 \cdots \delta_l} \quad \text{(CONCAT)}
\]

which handles the fact of multiple return values.

Moreover, a stack of types can be generalized on the left; this allows e.g. dup to be applied to 2 3.

\[
\frac{\Gamma \vdash x : \alpha_1 \cdots \alpha_n - \beta_1 \cdots \beta_m}{\Gamma \vdash x : a\alpha_1 \cdots \alpha_n - a\beta_1 \cdots \beta_m} \quad \text{(GENERALIZE)}
\]

2.2 Pattern Matching

Pattern matching in Kempe is somewhat atypical; a match clause strips the constructor off the stack but doesn’t bind anything:

```
type Maybe a { Just a | Nothing }

isJust : (Maybe a) -- Bool
  =: [ { case
              | Just -> drop True
              | Nothing -> False
        }
  ]
```

2.3 ABI

Kempe has its own ABI. All values are passed on the stack; a pointer to the Kempe stack is kept in a platform-specific register.

Representation of algebraic data types is straightforward: one begins with a tag for the constructor, then the data in question, in the order of the fields in question. Padding is added if needed so that all data of the same type has the same size.

As an example, consider

```
type These a { This a | That b | These a b }
```

Then a False That of type ((These Int) Bool) would be encoded like so:

```
0 1 2 3 4 5 6 7 8 9
1 0 padding
```

while False 0 These would be:

```
0 1 2 3 4 5 6 7 8 9
2 0x00000000 0
```
3 Frontend

3.1 Lexer

The lexer is written with Alex [8], a lexer generator.

We use the monadUserState-bytestring wrapper, which is more efficient than working with Strings.

3.1.1 Interning Identifiers

To aid interning identifiers, we have the type

```haskell
data Name a = Name { name :: Text
                     , unique :: !Int
                     , loc :: a
}
```

This will be essential for performance later on, as we will be able use IntMaps and IntSets over Maps and Sets.

3.2 Happy

The parser is written with Happy [9], a parser generator. This is more performant than parser combinators, which are commonly used in tutorials [5, 4].

3.3 Type Assignment

kc uses constraint-based typing [11] p. 322], which has two phases: one in which types are assigned and constraints dispatched, and a second in which constraints are solved by unification [11] p. 327].

If we have 0 dup (for instance), dup would be assigned type a -- a a, and the constraint a == 0 would be collected. Constraints are stored as a list of paired types [(KempeTy a, KempeTy a)], with each pair (KempeTy a, KempeTy a) being taken to mean equality. In this way, no atoms need type annotations; this is absolutely essential for stack-based programming, where basics like swap have polymorphic types (i.e. a b -- b a).

In the crude unification algorithm, we pick off a constraint from the rest (C') and form a new context C'':

- If the constraint X = Y holds (e.g. Int = Int), discard it and proceed (C'' := C').
- If the constraint is X = a for a a free variable, substitute X for all occurrences of a in C'', C'' := C'[a → X]. We also need to store the substitution [a → X] if we are going to annotate atoms with type information.
- If the constraint is XY = ZW, then add constraints X = Z and Y = W and proceed (C'' := (X = Z) : (Y = W) : C').
- In all other cases (e.g. Int = Bool), fail.

In practice, we have to be somewhat subtler as performing renaming on every step is expensive. So instead we finish the step by updating a UnifyMap with any new substitutions and only perform renamings when we are inspecting a type variable.
3.3.1 Annotation

For type assignment (as opposed to type checking), a third step writes in type annotations
with results from unification.

assignTy :: Atom c b -> Atom (StackType ()) (StackType ())

The Atom type is higher-kinded; we can annotate atoms yet reuse the underlying ADT.

data Atom c b = AtName b (Name b)
  | Case b (NonEmpty (Pattern c b, [Atom c b]))
  | If b [Atom c b] [Atom c b]
  | Dip b [Atom c b]
  ...

3.3.2 Renamer

Since type variables can be bound by implicit universal quantification, we need a renamer
for types.

This is most efficient in a stateful style; we use a state monad. Reassignments are
tracked in the monad, and substitutions are performed all at once when the type variables
are inspected down the line.

3.3.3 Unorthodoxies

Two unorthodoxies in type-checking: first, Kempe uses concatenation in place of function
application. If we have atoms x y, x may have multiple return values; we must stitch
together the stack-in types of y and stack-out types of x.

Second, it is stacks of types that must unify against one another (recall the GENERALIZE
rule). So when matching types such as Int -- Int and Int Int -- Int Int (for instance)
we must be permissive; we expand Int -- Int to a Int -- a Int.

3.4 Pattern-Match Exhaustiveness Checker

The pattern-match exhaustiveness checker will be necessary as we do not want to bother
with pattern match fail code.

Because of how pattern matching works in Kempe, this turns out to be quite simple: we
check that each constructor or least one wildcard is present.

3.5 Monomorphization

All functions must resolve to be monomorphic at the call site, so that sizes can be assigned.

Every non-recursive function is inlined before monomorphization, so we only need one
pass.

3.5.1 Inliner

Our stack language does not bind any variables, so we do not need to rename when inlining.
4 Backend

The compiler is a traditional pipeline architecture.

4.1 Intermediate Representation

The intermediate representation is also a stack machine, but replaces case- and if-statements with labels and jumps and translates higher-level constructs into memory moves in an architecture-independent way.

The IR is loosely based on Appel [3].

```haskell
data Temp = Temp64 !Int |
          Temp8 !Int |
          DataPointer |
          deriving (Eq)

data Stmt = Labeled Label |
            Jump Label -- corresponds to a register later in the pipeline |
            CJump Exp Label Label |
            MJump Exp Label |
            CCall MonoStackType BSL.ByteString |
            KCall Label -- ^ KCall is a jump to a Kempe procedure |
            WrapKCall ABI MonoStackType BSL.ByteString Label |
            MovTemp Temp Exp -- ^ Put e in temp |
            MovMem Exp Int64 Exp -- ^ Store e2 at address given by e1 |
            Ret |

data Exp = ConstInt Int64 |
          ConstInt8 Int8 |
          ConstTag Word8 -- ^ Used to distinguish constructors of a sum type |
          ConstWord Word
```
Translation from our frontend AST to IR is by replacing each atom with a series of 
Stmts (there are no loops in Kempe).

writeAtom :: SizeEnv
    -> Atom (ConsAnn MonoStackType) MonoStackType
    -> TempM [Stmt]

The SizeEnv contains size information we will need, and TempM is a monad giving us 
free unique labels and Temps.

type TempM = State TempSt

data TempSt = TempSt { labels :: [Label]
    , tempSupply :: [Int]
    }

runTempM :: TempM a -> a
runTempM = flip runState (TempSt [1..] [1..])

nextTemps :: TempSt -> TempSt
nextTemps (TempSt ls ts) = TempSt ls (tail ts)
getTemp :: TempM Int
getTemp = gets (head . tempSupply) <*> modify nextTemps
...

4.1.1 Sizing
Sizing is needed to translate stack manipulation functions (say, dup) to memory moves —
dup will have a different implementation when applied to a Bool vs. an Int.
  Sizing is fairly straightforward and notably depends on type inference.

4.1.2 Byte Copying
Builtins such as dup and swap builtins are implemented by generating code to copy the
appropriate number of bytes; there is no recourse to memcpy or the like, and moreover
the compiler does not generate any loops. This leads to problematic code size but the
implementation is simpler.

4.1.3 Optimization
kc needs to perform tail-call optimization as that is the facility for looping in Kempe.

4.2 Abstract Assembly
The IR is translated into an abstract assembly with infinitely many registers.
  That Kempe is a concatenative language has made our lives easier - there are no bound
variables in our translation to a series of (assembly) steps.

data Addr reg = Reg reg
  | AddRRPlus reg reg
  | AddRCPlus reg Int64
deriving (Eq)

data Cond = Eq
  | Neq
  | Geq
  | Lt
  | Gt
  ...

data Arm reg a = Branch a Label
  | BranchLink a Label
  | BranchZero a reg Label
  | AddRR a reg reg reg
  | AddRC a reg reg Int64
  | SubRC a reg reg Int64
  | SubRR a reg reg reg
  | MulRR a reg reg reg
  | MulSubRRR a reg reg reg reg
  | CSet a reg Cond
| Neg a reg reg
... deriving (Functor)

```haskell
data AbsReg = DataPointer
  | AllocReg !Int
  | CArg0 -- x0
  | CArg1
  | CArg2
  | CArg3
  | CArg4
  | CArg5
  | CArg6
  | CArg7 -- x7
  | LinkReg -- so we can save before/after branch-links
  | StackPtr -- so we can save in translation phase
 deriving (Eq)
```

Arm is a higher-kinded type, parametric in the register type. This will allow us to reuse the algebraic data types across the abstract assembly phase and register allocation.

```
allocRegs :: [Arm AbsReg Liveness] -> [Arm ArmReg ()]
```

### 4.3 Instruction Selection

Instruction selection is inspired by maximal munch [3]. For both Aarch64 and x86-64, we use two functions:

- `irEmit :: SizeEnv -> IR.Stmt -> WriteM [Arm AbsReg ()]`
- `evalE :: IR.Exp -> IR.Temp -> WriteM [Arm AbsReg ()]`

`evalE` puts expressions in a Temp recursively; `WriteM` is a monad for fresh labels and registers, like `TempM` mentioned previously.

As an example, our implementation with pattern matching:

```
-- e.g. (mjump (=b (mem [1] (* (reg datapointer) (int 0))) (tag 0x0)) kmp2)
irEmit _ (IR.MJump (IR.EqByte e (IR.ConstTag 0)) l) = do
  { r <- allocTemp64
    ; eEval <- evalE e r
    ; pure $ eEval ++ [BranchZero () (toAbsReg r) l]
  }

irEmit _ (IR.MJump e l) = do
  { r <- allocTemp64
    ; eEval <- evalE e r
    ; pure $ eEval ++ [BranchNonzero () (toAbsReg r) l]
  }
```

This is somewhat lacking compared to maximal munch because we only match on one Stmt at a time (rather than a list of Stmt), but in any case we can see how pattern matching picks more efficient idioms.
4.4 Register Allocation

Register allocation depends on conventional compiler analyses:

- Arm AbsReg ()
- Arm AbsReg ControlAnn
- Arm AbsReg Liveness
- Linear Allocator

In all transformations we finish with annotated instructions, rather than an information table; this is the approach used in GHC [10].

4.4.1 Control Flow

Control flow is defined by an instruction’s predecessor and successors; an instruction like `BranchZero` would have its target and the next instruction as successors.

In this analysis, we first give each node (instruction) a label and then in a second pass annotate each instruction with it with the labels of its successors using the map built up in the first pass.

```haskell
data ControlAnn = ControlAnn { node :: !Int,
                                  succ :: [Int]
                                }

mkControlFlow :: [Arm AbsReg ()] -> [Arm AbsReg ControlAnn]
```

4.4.2 Liveness Analysis

Liveness analysis is based on Appel [3 p. 212-215].

There is one subtlety here: we use `Ints` for `AbsReg`s so that we can use `IntSets` during liveness analysis rather than `Set AbsReg`s. This gives a substantial performance boost.

We can implement

```haskell
uses :: Arm AbsReg a -> IntSet
defs :: Arm AbsReg a -> IntSet
```

functions which tells us which registers are used and defined by a particular instruction.

Then define
Liveness \{ \text{ins} :: !\text{IS.IntSet}, \text{out} :: !\text{IS.IntSet} \}

which is associated with an \text{Arm AbsReg a}. We proceed in a stateful/iterative way:

1. Initialize each node with \text{ins} and \text{out} empty.
2. Iterate over all nodes, updating them with \text{def} and \text{uses} as well as previous \text{ins} and \text{outs} annotations:
   
   \[
   \text{ins'} \ n = \text{uses} \ n \triangledown (\text{out} \ n \setminus \text{defs} \ n)
   \]
   
   \[
   \text{out'} \ n = \text{unions} (\text{fmap} \ \text{ins} \ (\text{succ} \ n))
   \]
3. End when all the \text{ins} and \text{outs} are the same after one iteration.

This will run faster when iterate backwards over the nodes, i.e. begin with the last assembly instruction. See Appel [3, p. 214-215] for proof of correctness and explanation of backwards iteration.

4.4.3 Linear Allocator

We use a linear scan register allocator, which is easy to implement and performant. Because of how the stack-based IR is generated, we never have to spill.

With liveness analysis annotations:

1. A register needs to be allocated when it is live out at a node but not live in.
2. If an abstract register is live in and live out at a node, look up which register we’ve assigned it to.
3. A register may be freed when it is live out at a node but not live in.

See CS 143 lecture notes [2] for a picture of how this works.

References


A Renamer

-- Make sure there are no cycles in the renames map!
replaceUnique :: Unique -> TypeM a Unique
replaceUnique u@(Unique i) = do
  rSt <- gets renames
  case IM.lookup i rSt of
    Nothing -> pure u
    Just j -> replaceUnique (Unique j)

renameIn :: KempeTy a -> TypeM a (KempeTy a)
renameIn b@TyBuiltin{} = pure b
renameIn n@TyNamed{} = pure n
renameIn (TyApp l ty ty') = TyApp l <$> renameIn ty <*> renameIn ty'
renameIn (TyVar l (Name t u l')) = do
  u' <- replaceUnique u
  pure $ TyVar l (Name t u' l')

withName :: Name a -> TypeM a (Name a, TyState a -> TyState a)
withName (Name t (Unique i) l) = do
  m <- gets maxU
  let newUniq = m+1
  maxULens .= newUniq
  pure (Name t (Unique newUniq) l, over renamesLens (IM.insert i (m+1)))

-- freshen the names in a stack so there aren't overlaps
-- in quantified variables
renameStack :: StackType a -> TypeM a (StackType a)
renameStack (StackType qs ins outs) = do
  newQs <- traverse withName (S.toList qs)
  let (newNames, localRenames) = unzip newQs
  newBinds = thread localRenames
  withTyState newBinds $ StackType (S.fromList newNames) <$> traverse renameIn ins <*> traverse renameIn outs
B Unification

type UnifyMap = IM.IntMap (KempeTy ()

inContext :: UnifyMap -> KempeTy () -> KempeTy ()
inContext um ty'@(TyVar _ (Name _ (Unique i) _)) =
case IM.lookup i um of
  Just ty@TyVar{} -> inContext (IM.delete i um) ty -- prevent cyclic lookups
  Just ty -> ty
  Nothing -> ty'
inContext _ ty'@TyBuiltin{} = ty'
inContext _ ty'@TyNamed{} = ty'
inContext um (TyApp l ty ty') = TyApp l (inContext um ty) (inContext um ty')

-- | Perform substitutions before handing off to 'unifyMatch'
unifyPrep :: UnifyMap
  -> [(KempeTy (), KempeTy ())]
  -> Either (Error ()) (IM.IntMap (KempeTy ()))
unifyPrep _ [] = Right mempty
unifyPrep um ((ty, ty'):tys) =
  let ty'' = inContext um ty
      ty''' = inContext um ty'
in unifyMatch um $ (ty'', ty'''):tys

unifyMatch :: UnifyMap
  -> [(KempeTy (), KempeTy ())]
  -> Either (Error ()) (IM.IntMap (KempeTy ()))
unifyMatch _ [] = Right mempty
unifyMatch um ((ty@(TyBuiltin _ b0), ty'@(TyBuiltin _ b1)):tys)
  | b0 == b1 = unifyPrep um tys
  | otherwise = Left (UnificationFailed () ty ty')
unifyMatch um ((ty@(TyNamed _ n0), ty'@(TyNamed _ n1)):tys)
  | n0 == n1 = unifyPrep um tys
  | otherwise = Left (UnificationFailed () (void ty) (void ty'))
unifyMatch um ((ty@(TyNamed _ _), TyVar _ (Name _ (Unique k) _)):tys)
  = IM.insert k ty <$> unifyPrep (IM.insert k ty um) tys
unifyMatch um ((TyVar _ (Name _ (Unique k) _), ty@(TyNamed _ _)):tys)
  = IM.insert k ty <$> unifyPrep (IM.insert k ty um) tys
unifyMatch um ((ty@(TyBuiltin{}), TyVar _ (Name _ (Unique k) _)):tys)
  = IM.insert k ty <$> unifyPrep (IM.insert k ty um) tys
unifyMatch um ((TyVar _ (Name _ (Unique k) _), ty@(TyBuiltin _ _)):tys)
  = IM.insert k ty <$> unifyPrep (IM.insert k ty um) tys
unifyMatch um ((TyVar _ (Name _ (Unique k) _), ty@(TyVar _ _)):tys)
  = IM.insert k ty <$> unifyPrep (IM.insert k ty um) tys
unifyMatch _ ((ty@TyBuiltin{}, ty'@TyNamed{}):_)
  = Left (UnificationFailed () ty ty')
unifyMatch _ ((ty@TyNamed{}, ty'@TyBuiltin{}):_)

13
C  Byte Copying

See how the dup and swap builtins are implemented to get a flavor of translating from Atoms to our IR:

writeAtom env _ (AtBuiltin (is, _) Dup) =
  let sz = size’ env (last is) in
  pure $  
    copyBytes 0 (-sz) sz
  ++ [ dataPointerInc sz ] -- move data pointer over sz bytes
writeAtom env _ (AtBuiltin ([i0, i1], _) Swap) =
  let sz0 = size’ env i0
      sz1 = size’ env i1
      in
    pure $  
      copyBytes 0 (-sz0 - sz1) sz0 -- copy i0 to end of the stack
      ++ copyBytes (-sz0 - sz1) (-sz1) sz1 -- copy i1 to where i0 used to be
      ++ copyBytes (-sz0) 0 sz0 -- copy i0 at end of stack to its new place
writeAtom _ _ (AtBuiltin _ Swap) = error "Ill-typed swap!"

copyBytes :: Int64 -- ^ dest offset
-> Int64 -- ^ src offset
-> Int64 -- ^ Number of bytes to copy
-> [Stmt]
copyBytes off1 off2 b =
  let (b8, b1) = b ‘quotRem’ 8
      in copyBytes8 off1 off2 b8 ++ copyBytes1 (off1 + b8 * 8) (off2 + b8 * 8) b1
-- | Copy bytes 8 at a time. Note that @b@ must be divisible by 8.
copyBytes8 :: Int64
-> Int64
-> Int64 -- ^ @b@ (number 8-byte chunks to copy)
-> [Stmt]
copyBytes8 off1 off2 b =
  let is = fmap (8*) [0..(b - 1)] in
  [ MovMem (dataPointerPlus (i + off1)) 8 (Mem 8 $ dataPointerPlus (i + off2))
    | i <- is ]

copyBytes1 :: Int64 -> Int64 -> Int64 -> [Stmt]
copyBytes1 off1 off2 b =
  [ MovMem (dataPointerPlus (i + off1)) 1 (Mem 1 $ dataPointerPlus (i + off2))
    | i <- [0..(b-1)] ]

dataPointerDec :: Int64 -> Stmt
dataPointerDec i =
  MovTemp
  DataPointer
  (ExprIntBinOp IntMinusIR (Reg DataPointer) (ConstInt i))

dataPointerInc :: Int64 -> Stmt
dataPointerInc i =
  MovTemp
  DataPointer
  (ExprIntBinOp IntPlusIR (Reg DataPointer) (ConstInt i))

dataPointerPlus :: Int64 -> Exp
dataPointerPlus off =
  if off > 0
    then ExprIntBinOp IntPlusIR (Reg DataPointer) (ConstInt off)
    else ExprIntBinOp IntMinusIR (Reg DataPointer) (ConstInt (negate off))

D Control Flow

As the Haskell implementation of control flow analysis does not follow directly from the explanation above, the control flow module is reproduced below to fill in some details:

module Kempe.Asm.Arm.ControlFlow ( mkControlFlow
, ControlAnn (..)
) where

import Control.Monad.State.Strict (State, evalState, gets, modify)
import Data.Bifunctor (first, second)
import Data.Functor (($>))
import qualified Data.IntSet as IS
import qualified Data.Map as M
import Data.Semigroup ((<>))
import Kempe.Asm.Arm.Type
import Kempe.Asm.Type
-- map of labels by node

type FreshM = State (Int, M.Map Label Int)

runFreshM :: FreshM a -> a
runFreshM = flip evalState (0, mempty)

mkControlFlow :: [Arm AbsReg ()] -> [Arm AbsReg ControlAnn]
mkControlFlow instrs = runFreshM (broadcasts instrs <*> addControlFlow instrs)

getFresh :: FreshM Int
getFresh = gets fst <*> modify (first (+1))

lookupLabel :: Label -> FreshM Int
lookupLabel l = gets
  (M.findWithDefault
   (error "Internal error in control-flow graph: node label not in map.") l
    . snd)

broadcast :: Int -> Label -> FreshM ()
broadcast i l = modify (second (M.insert l i))

singleton :: AbsReg -> IS.IntSet
singleton = maybe IS.empty IS.singleton . toInt

-- | Can’t be called on abstract registers i.e. ‘DataPointer’
-- It allows us to use an 'IntSet' for liveness analysis.
toInt :: AbsReg -> Maybe Int
toInt (AllocReg i) = Just i
toInt _ = Nothing

fromList :: [AbsReg] -> IS.IntSet
fromList = foldMap singleton

addrRegs :: Addr AbsReg -> IS.IntSet
addrRegs (Reg r) = singleton r
addrRegs (AddRRPlus r r') = fromList [r, r']
addrRegs (AddRPlus r _) = singleton r

-- | Annotate instructions with a unique node name and a list of all possible
-- destinations.
addControlFlow :: [Arm AbsReg ()] -> FreshM [Arm AbsReg ControlAnn]
addControlFlow [] = pure []
addControlFlow ([Label l]:asms) = do
  i <- lookupLabel l
  ; (f, asms') <- next asms
  ; pure (Label (ControlAnn i (f [])) IS.empty IS.empty) l : asms')
addControlFlow ((BranchCond _ l c):asms) = do
  { i <- getFresh
    ; (f, asms') <- next asms
    ; l_i <- lookupLabel l
    ; pure (BranchCond (ControlAnn i (f [l_i]) IS.empty IS.empty) l c : asms')
  }

addControlFlow ((BranchLink _ l):asms) = do
  { i <- getFresh
    ; nextAsms <- addControlFlow asms
    ; l_i <- lookupLabel l
    ; pure (BranchLink (ControlAnn i [l_i] IS.empty IS.empty) l : nextAsms)
  }

addControlFlow (Ret{}:asms) = do
  { i <- getFresh
    ; nextAsms <- addControlFlow asms
    ; pure (Ret (ControlAnn i [] IS.empty IS.empty) : nextAsms)
  }

addControlFlow (asm:asms) = do
  { i <- getFresh
    ; (f, asms') <- next asms
    ; pure ((asm $> ControlAnn i (f []) (uses asm) (defs asm)) : asms')
  }

uses :: Arm AbsReg ann -> IS.IntSet
uses (MovRR _ _ r) = singleton r
uses (AddRR _ _ r r') = fromList [r, r']
uses (SubRR _ _ r r') = fromList [r, r']
-- since MovRK only affects 16 bits, it depends on the previous r to be live!
uses (MovRK _ r _ _) = singleton r
... uses (Store _ r a) = singleton r <> addrRegs a
uses _ = mempty

defs :: Arm AbsReg ann -> IS.IntSet
defs (MovRR _ _ r _) = singleton r
defs (MovRC _ _ r _) = singleton r
defs (LoadLabel _ _ r _) = singleton r
... defs _ = mempty

next :: [Arm AbsReg ()] -> FreshM ([Int] -> [Int], [Arm AbsReg ControlAnn])
next asms = do
  nextAsms <- addControlFlow asms
  case nextAsms of
    [] -> pure (id, [])
    (asm:_) -> pure ((node (ann asm) :], nextAsms)
broadcasts :: [Arm reg ()] -> FreshM [Arm reg ()]
broadcasts [] = pure []
broadcasts (asm@(Label _ l):asms) = do
  \{ i <- getFresh \\
  ; broadcast i l \\
  ; (asm :) <$> broadcasts asms \\
  \}
broadcasts (asm:asms) = (asm :) <$> broadcasts asms

E  Liveness Analysis

As the liveness analysis module is 60 or so lines, it is reproduced below:

module Kempe.Asm.Liveness ( Liveness
    , reconstruct
  ) where

import       Data.Copointed
import qualified Data.IntMap.Lazy as IM
import qualified Data.IntSet    as IS
import       Data.Semigroup     ((<>))
import       Kempe.Asm.Type

emptyLiveness :: Liveness
emptyLiveness = Liveness IS.empty IS.empty

initLiveness :: Copointed p => [p ControlAnn] -> LivenessMap
initLiveness = IM.fromList .
  fmap (\asm -> let x = copoint asm in (node x, (x, emptyLiveness)))

type LivenessMap = IM.IntMap (ControlAnn, Liveness)

-- | All program points accessible from some node.
succNode :: ControlAnn --^ 'ControlAnn' associated w/ node @n@
       -> LivenessMap
       -> [Liveness] --^ 'Liveness' associated with 'succNode' @n@
succNode x ns =
  let conn  = conn x
      in fmap (snd . flip lookupNode ns) conn;

lookupNode :: Int -> LivenessMap -> (ControlAnn, Liveness)
lookupNode = IM.findWithDefault
  (error "Internal error: failed to look up instruction")

done :: LivenessMap -> LivenessMap -> Bool
done n0 n1 = and $
  zipWith (\(_, l) (_, l') -> l == l') (IM.elems n0) (IM.elems n1)
-- n0, n1 must have same length

-- order in which to inspect nodes during liveness analysis
-- don't need to reverse because thread goes in opposite order
inspectOrder :: Copointed p => [p ControlAnn] -> [Int]
inspectOrder = fmap (node . copoint)

reconstruct :: (Copointed p, Functor p) => [p ControlAnn] -> [p Liveness]
reconstruct asms = fmap (fmap lookupL) asms
  where l = mkLiveness asms
        lookupL x = snd $ lookupNode (node x) l

mkLiveness :: Copointed p => [p ControlAnn] -> LivenessMap
mkLiveness asms = liveness is (initLiveness asms)
  where is = inspectOrder asms

liveness :: [Int] -> LivenessMap -> LivenessMap
liveness is nSt =
  if done nSt nSt'
    then nSt
    else liveness is nSt'
  where nSt' = iterNodes is nSt

iterNodes :: [Int] -> LivenessMap -> LivenessMap
iterNodes is = thread (fmap stepNode is)
  where thread = foldr (.) id

stepNode :: Int -> LivenessMap -> LivenessMap
stepNode n ns = IM.insert n (c, Liveness ins' out') ns
  where (c, l) = lookupNode n ns
        ins' = usesNode c <> (out l IS.\defsNode c)
        out' = IS.unions (fmap ins (succNode c ns))

This module is architecture-independent, as both Aarch64 and x86-64 abstract assembly
are instances of Copointed:

module Data.Copointed ( Copointed (..)
) where

class Copointed p where
copoint :: p a -> a

Implementation for the Arm type:

data Arm reg a = Branch { ann :: a, label :: Label }
  | BranchLink { ann :: a, label :: Label }
...

instance Copointed (Arm reg) where
  copoint = ann