Tutor's guide
for
Implementing functional languages: a tutorial

Simon L Peyton Jones
Department of Computing Science, University of Glasgow

and David R Lester
Department of Computer Science, University of Manchester

March 23, 2000

Contents

Preface

This Tutor's Guide contains answers to many of the exercises, including sufficient code to build an executable program for each of the versions presented in the main text.

You should read it in conjunction with the Installation Guide.

While the Tutor's Guide contains the Miranda code to solve almost all the exercises, it is very short on supporting text. We plan to improve it in the places that need it most, so please let us know where you looked for help and did not find it.

Chapter 1

The Core language

The first four sections of this chapter are essential, since they introduce the Core language itself.

The last two sections, which develop a pretty printer and parser for the Core language can be omitted if desired. We have included them for two reasons. Firstly, they each provide a non-trivial example of functional programming. Secondly, in a book like this we felt it was important to show the *entire* implementation.

The pretty printer uses the idea of abstract data types, and briefly introduces Miranda's notation for such ADTs.

1.1 Core language overview

1.2 Syntax of the Core language

1.3 Data types for the Core language

This section uses Miranda algebraic data type declarations and type synonyms for the first time.

1.4 A small standard prelude

1.5 A pretty-printer for the Core language

This section contains the first substantial programming exercises.

1.5.1 Pretty printing using strings

1.5.2 An abstract data type for pretty-printing

Solution to Exercise 1.2.

```
> iConcat = foldr iAppend iNil


> iInterleave sep []         = iNil
> iInterleave sep [seq]      = seq
> iInterleave sep (seq:seqs) = seq `iAppend` (sep `iAppend` iInterleave sep seqs)
```

Solution to Exercise 1.3.

Here is a pretty printer for the complete Core language (quick fix to put binary operators in infix position – doesn't include Exercise 1.8. also modified from previous version to put parentheses around lambda definition to delimit definition).

```
> pprProgram prog = iInterleave (iAppend (iStr " ;") iNewline) (map pprSc prog)
> -- Isn't this what you meant?


> pprSc (name, args, body)
>   = iConcat [ iStr name, iSpace, pprArgs args,
>               iStr " = ", iIndent (pprExpr body) ]


> pprExpr :: CoreExpr -> Iseq
> pprExpr (ENum n) = iNum n
> pprExpr (EVar v) = iStr v
> pprExpr (EAp (EAp (EVar "+") e1) e2)  = iConcat [pprAExpr e1, iStr " + ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "-") e1) e2)  = iConcat [pprAExpr e1, iStr " - ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "*") e1) e2)  = iConcat [pprAExpr e1, iStr " * ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "/") e1) e2)  = iConcat [pprAExpr e1, iStr " / ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "<") e1) e2)  = iConcat [pprAExpr e1, iStr " < ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "<=") e1) e2)  = iConcat [pprAExpr e1, iStr " <= ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "==") e1) e2)  = iConcat [pprAExpr e1, iStr " == ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "~=") e1) e2)  = iConcat [pprAExpr e1, iStr " ~= ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar ">=") e1) e2)  = iConcat [pprAExpr e1, iStr " >= ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar ">") e1) e2)  = iConcat [pprAExpr e1, iStr " > ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "&") e1) e2)  = iConcat [pprAExpr e1, iStr " & ", pprAExpr e2]
> pprExpr (EAp (EAp (EVar "|") e1) e2)  = iConcat [pprAExpr e1, iStr " | ", pprAExpr e2]
> pprExpr (EAp e1 e2) = iConcat [ pprExpr e1, iSpace, pprAExpr e2 ]


> pprExpr (ELet isrec defns expr)
>   = iConcat [  iStr keyword, iNewline,
>                iStr "  ", iIndent (pprDefns defns), iNewline,
>                iStr "in ", pprExpr expr ]
>     where
>     keyword | not isrec = "let"
>             | isrec = "letrec"


> pprExpr (ECase e alts)
>   = iConcat [  iStr "case ", pprExpr e, iStr " of", iNewline,
```

```
>                iStr "  ", iIndent (iInterleave iNl (map pprAlt alts)) ]
>     where
>     iNl = iConcat [ iStr ";", iNewline ]
>
>     pprAlt (tag, args, rhs)
>       = iConcat [        iStr "<", iNum tag, iStr "> ",
>                          pprArgs args, iStr " -> ",
>                          iIndent (pprExpr rhs) ]

> pprExpr (ELam args body)
>   = iConcat [  iStr "(\\", pprArgs args, iStr ". ", iIndent (pprExpr body),
>                iStr ")"]

> pprArgs args = iInterleave iSpace (map iStr args)

> pprAExpr e | isAtomicExpr e = pprExpr e
> pprAExpr e | otherwise = iConcat [iStr "(", pprExpr e, iStr ")"]
```

1.5.3 Implementing `iseq`

1.5.4 Layout and indentation

```
> flatten col ((INewline, indent) : seqs)
>   = '\n' : (space indent) ++ (flatten indent seqs)

> flatten col ((IIndent seq, indent) : seqs)
>   = flatten col ((seq, col) : seqs)

> flatten col ((IStr s, indent) : seqs)
>   = s ++ flatten (col + length s) seqs

> flatten col ((INil, indent) : seqs) = flatten col seqs

> flatten col ((IAppend seq1 seq2, indent) : seqs)
>   = flatten col ((seq1, indent) : (seq2, indent) : seqs)

> flatten col [] = ""
```

1.5.5 Operator precedence

1.5.6 Other useful functions on `iseq`

```
> iSpace = iStr " "
```

1.6 A parser for the Core language

1.6.1 Lexical analysis

This lexical analyser recognises comments from a double vertical bar to the end of the line, and recognises two-character operators. We begin with the same equations as before.

```
> clex (c:cs) | isWhiteSpace c = clex cs

> clex (c:cs) | isDigit c = num_token : clex rest_cs
>             where
>             num_token = c : takeWhile isDigit cs
>             rest_cs   = dropWhile isDigit cs

> clex (c:cs) | isAlpha c = var_tok : clex rest_cs
>             where
>             var_tok = c : takeWhile isIdChar cs
>             rest_cs = dropWhile isIdChar cs
```

Now the extra equation for handling a comment:

```
> clex ('|':'|':cs) = clex (dropWhile (/= '\n') cs)
```

And the equation for two-character operators:

```
> clex (c1:c2:cs) | ([c1, c2] `elem` twoCharOps) = [c1,c2] : clex cs
```

The remaining two equations are unchanged.

```
> clex (c:cs) = [c] : clex cs
> clex [] = []
```

1.6.2 Syntax analysis

1.6.3 Basic tools for parsing

1.6.4 Sharpening the tools

Solution to Exercise 1.12.

```
> pThen3 :: (a -> b -> c -> d)
>           -> Parser a -> Parser b -> Parser c -> Parser d
> pThen3 combine p1 p2 p3 toks
>   = [(combine v1 v2 v3, toks3) | (v1,toks1) <- p1 toks,
>                                  (v2,toks2) <- p2 toks1,
>                                  (v3,toks3) <- p3 toks2]
```

```
> pThen4 :: (a -> b -> c -> d -> e)
>           -> Parser a -> Parser b -> Parser c
>           -> Parser d -> Parser e
> pThen4 combine p1 p2 p3 p4 toks
>   = [(combine v1 v2 v3 v4, toks4) | (v1,toks1) <- p1 toks,
>                                     (v2,toks2) <- p2 toks1,
>                                     (v3,toks3) <- p3 toks2,
>                                     (v4,toks4) <- p4 toks3]
```

Solution to Exercise 1.13.

```
> pEmpty v toks = [(v, toks)]

> pOneOrMore p = pThen (:) p (pZeroOrMore p)
```

Solution to Exercise 1.14.

```
> pApply p f toks = [(f v, toks') | (v, toks') <- p toks]
```

Solution to Exercise 1.15.

```
> pOneOrMoreWithSep p psep = pThen (:) p (pOneOrMoreWithSep_c p psep)
>
> pOneOrMoreWithSep_c p psep
>   = (pThen discard_sep psep (pOneOrMoreWithSep p psep)) `pAlt`
>     (pEmpty [])
>     where
>     discard_sep sep vs = vs
```

Solution to Exercise 1.16.

```
> pSat pred []         = []
> pSat pred (tok:toks) | pred tok = [(tok,toks)]
>                      | otherwise = []
```

Solution to Exercise 1.17.

```
> pVar = pSat isVar
>        where
>        isVar s = isAlpha (head s) && s `notElem` keywords
```

Solution to Exercise 1.18.

```
> pNum = pSat (isDigit . head) `pApply` numval

> numval :: String -> Int
> numval = foldll (\a c -> 10 * a + ord c - ord '0') 0
```

1.6.5 Parsing the Core language

```
> mk_sc sc args eq rhs = (sc, args, rhs)

> pExpr = pLet `pAlt` (pCase `pAlt` (pLambda `pAlt` pExpr1))

> pLet = pThen4 mk_let
>               ((pLit "let") `pAlt` (pLit "letrec")) pDefns
>               (pLit "in") pExpr
>        where
>        mk_let keyword defns in' expr = ELet (keyword == "letrec") defns expr
>
> pDefns = pOneOrMoreWithSep pDefn (pLit ";")
> pDefn = pThen3 mk_defn pVar (pLit "=") pExpr
>         where
>         mk_defn var equals rhs = (var,rhs)

> pCase = pThen4 mk_case (pLit "case") pExpr (pLit "of") pAlters
>         where
>         mk_case case' e of' alts = ECase e alts
>
> pAlters = pOneOrMoreWithSep pAlter (pLit ";")
> pAlter = pThen4 mk_alt pTag (pZeroOrMore pVar) (pLit "->") pExpr
>          where
>          mk_alt tag args arrow rhs = (tag, args, rhs)
>
> pTag = pThen3 get_tag (pLit "<") pNum (pLit ">")
>        where
>        get_tag lb tag rb = tag

> pLambda = pThen4 mk_lam
>               (pLit "\\") (pOneOrMore pVar) (pLit ".") pExpr
>           where
>           mk_lam lam vars dot expr = ELam vars expr

> pExpr2 = pThen assembleOp pExpr3 pExpr2c
> pExpr2c = (pThen FoundOp (pLit "&") pExpr2) `pAlt` (pEmpty NoOp)

> pExpr3 = pThen assembleOp pExpr4 pExpr3c
> pExpr3c = (pThen FoundOp pRelop pExpr4) `pAlt` (pEmpty NoOp)

> pRelop = pSat (`elem` relops)
>          where
>          relops = ["<=", "<", ">=", ">", "==", "~="]
```

```
> pExpr4 = pThen assembleOp pExpr5 pExpr4c
> pExpr4c = (pThen FoundOp (pLit "+") pExpr4) `pAlt`
>           ((pThen FoundOp (pLit "-") pExpr5) `pAlt`
>            (pEmpty NoOp))

> pExpr5 = pThen assembleOp pExpr6 pExpr5c
> pExpr5c = (pThen FoundOp (pLit "*") pExpr5) `pAlt`
>           ((pThen FoundOp (pLit "/") pExpr6) `pAlt`
>            (pEmpty NoOp))

> pExpr6 = (pOneOrMore pAtomic) `pApply` mk_ap_chain
>          where
>          mk_ap_chain (fn:args) = foldll EAp fn args

> pAtomic = pConstr `pAlt`
>           (pBracExpr `pAlt`
>           ((pVar `pApply` EVar) `pAlt`
>           ((pNum `pApply` ENum))))
```

The extra parens in the definitions of `pExpr4`, `pExpr5` and `pAtomic` are there so that the compiler doesn't need to assume that `pAlt` is associative. The parens make the grouping explicit.

```
> pBracExpr = pThen3 mk_brack (pLit "(") pExpr (pLit ")")
>             where
>             mk_brack open expr close = expr

> pConstr = pThen4 pick_constr (pLit "Cons") (pLit "{") pTagArity (pLit "}")
>           where
>           pick_constr cons lbrack constr rbrack = constr
>           pTagArity = pThen3 mk_constr pNum (pLit ",") pNum
>           mk_constr tag comma arity = EConstr tag arity
```

Chapter 2

Template-instantiation

2.1 Mark 1: the basic machine

Solution to exercise 2.3 must exist for template1 to compile without error. (change Makefile to include template-tutor.src for template1.lhs)

```
> multFinal (_, m, d, _) | m == 0 && d == 0  = True
>                        | otherwise         = False
```

Exercises 2.4 to 2.7 are straightforward.

Solution to Exercise 2.9. If the proposed change is made, only the state *before* the one causing the error is printed. Both equations of the proposed `eval` return `state` at the head of its result list, but no list at all will be returned until `tiFinal` returns its result. We can get a bit more output by returning `state` as the head of the list, and using `tiFinal` to decide whether the rest of the list is empty (as the real `eval` does).

This strictness bug actually caught one of the authors out!

2.2 Mark 2: Let(rec) expressions

Here is the code for `instantiateLet`, which deals with both the recursive and non-recursive case.

```
> instantiateLet isrec defs body heap old_env
>  = instantiate body heap1 new_env
>    where
>    (heap1, extra_bindings) = mapAccuml instantiate_rhs heap defs
>
>    new_env = extra_bindings ++ old_env
>    rhs_env | isrec     = new_env
>            | otherwise = old_env
>
```

```
>    instantiate_rhs heap (name, rhs)
>     = (heap1, (name, addr))
>       where
>       (heap1, addr) = instantiate rhs heap rhs_env
```

The `mapAccuml` runs down the list, instantiating the right-hand side of each definition, returning the extra bindings created thereby, and of course the new heap. The old bindings are augmented with the extra bindings to create `new_env`, which is used in the instantiation of the body of the `let(rec)`.

Each right-hand side is instantiated in the environment `rhs_env`, which is defined to be `old_env` for the non-recursive case, and `new_env` for the recursive case. We have tied the recursive knot in Miranda! This is possible because the addresses produced for the new graphs do not depend on `rhs_env`.

2.3 Mark 3: Updating

Here is the code for updating, including the optimisation of Section 2.5.1

2.3.1 The new `scStep` function

First, we must redefine `scStep` to call `instantiateAndUpdate`. It drops one fewer addresses off the stack than before, because the root of the redex can now remain.

```
> scStep   :: TiState -> Name -> [Name] -> CoreExpr -> TiState
> scStep (stack, dump, heap, globals, stats) sc_name arg_names body
>  = (new_stack, dump, new_heap, globals, stats)
>    where
>    new_stack = drop (length arg_names) stack
>    root = hd new_stack
>    new_heap = instantiateAndUpdate body root heap (bindings ++ globals)
>    bindings = zip2 arg_names (getargs heap stack)
```

2.3.2 The new instantiation function

Next, we give the new definition for `instantiateAndUpdate`:

```
> instantiateAndUpdate (ENum n) upd_addr heap env
>  = hUpdate heap upd_addr (NNum n)
>
> instantiateAndUpdate (EAp e1 e2) upd_addr heap env
>  = hUpdate heap2 upd_addr (NAp a1 a2)
>    where
>    (heap1, a1) = instantiate e1 heap  env
>    (heap2, a2) = instantiate e2 heap1 env
```

```
>
> instantiateAndUpdate (EVar v) upd_addr heap env
>   = hUpdate heap upd_addr (NInd var_addr)
>     where
>     var_addr = aLookup env v
>                      (error ("Undefined name " ++ show v))
```

For the `let(rec)` case, notice the recursive call to `instantiateAndUpdate` when instantiating the body.

```
> instantiateAndUpdate (ELet isrec defs body) upd_addr heap old_env
>  = instantiateAndUpdate body upd_addr heap1 new_env
>    where
>    (heap1, extra_bindings) = mapAccuml instantiate_rhs heap defs
>
>    new_env = extra_bindings ++ old_env
>    rhs_env = if isrec then new_env else old_env
>
>    instantiate_rhs heap (name, rhs)
>     = (heap1, (name, addr))
>       where
>       (heap1, addr) = instantiate rhs heap rhs_env
```

The case for constructors is handled as before:

```
> instantiateAndUpdate (EConstr tag arity) upd_addr h b
>             = instantiateAndUpdateConstr tag arity upd_addr h b


> instantiateAndUpdateConstr tag arity upd_addr h b
>    = error "instantiateAndUpdateConstr: not implemented yet"
```

2.3.3 Changes to the state transitions

We need an extra equation in the definition of `dispatch`, to call `indStep` if we find an indirection:

```
> step state
>  = dispatch (hLookup heap (hd stack))
>    where
>    (stack, dump, heap, globals, stats) = state
>    dispatch (NInd a)                   = indStep state a
>    dispatch (NAp a1 a2)                = apStep  state a1 a2
>    dispatch (NSupercomb sc args body) = scStep  state sc args body
```

In the case of an indirection, we remove the address of the indirection from the stack and replace it with the address of the node to which it points.

```
> indStep :: TiState -> Addr -> TiState
> indStep (a : stack, dump, heap, globals, stats) a'
>  = (a' : stack, dump, heap, globals, stats)
```

2.3.4 New printing equations

We need a new case in `showNode` for indirections:

```
> showNode (NAp a1 a2) = iConcat [ iStr "NAp ", showAddr a1,
>                                  iStr " ",    showAddr a2
>                          ]
> showNode (NSupercomb name args body) = iStr ("NSupercomb " ++ name)
> showNode (NNum n) = (iStr "NNum ") `iAppend` (iNum n)
> showNode (NInd a) = (iStr "NInd ") `iAppend` (showAddr a)
```

2.4 Mark 4: Arithmetic

We present the completed code for this section.

2.4.1 New state transitions

```
> tiFinal ([sole_addr], [], heap, globals, stats)
>  = isDataNode (hLookup heap sole_addr)
>
> tiFinal ([], dump, heap, globals, stats) = error "Empty stack!"
> tiFinal state = False


> step state
>  = dispatch (hLookup heap (hd stack))
>    where
>    (stack, dump, heap, globals, stats) = state
>    dispatch (NNum n)                   = numStep  state n
>    dispatch (NInd a)                   = indStep  state a
>    dispatch (NAp a1 a2)                = apStep   state a1 a2
>    dispatch (NSupercomb sc args body) = scStep   state sc args body
>    dispatch (NPrim name prim)          = primStep state prim


> apStep :: TiState -> Addr -> Addr -> TiState
> apStep (stack, dump, heap, globals, stats) a1 a2
>  = ap_dispatch (hLookup heap a2)
>    where
>
>    ap_dispatch (NInd a3) = (stack, dump, heap', globals, stats)
>                            where heap' = hUpdate heap ap_node (NAp a1 a3)
```

```
>                                   ap_node = hd stack
>
>    ap_dispatch node = (a1 : stack, dump, heap, globals, stats)

> numStep (stack, stack':dump, heap, globals, stats) n
>  = (stack', dump, heap, globals, stats)
```

2.4.2 Definition for arithmetic primitives

```
> primNeg :: TiState -> TiState
> primNeg (stack, dump, heap, globals, stats)
>  | length args /= 1 = error "primNeg: wrong number of args"
>  | not (isDataNode arg_node) = ([arg_addr], new_stack:dump, heap, globals, stats)
>  | otherwise = (new_stack, dump, new_heap, globals, stats)
>    where
>    args = getargs heap stack              -- Should be just one arg
>    [arg_addr] = args
>    arg_node = hLookup heap arg_addr        -- Get the arg node itself
>    NNum arg_value = arg_node              -- Extract the value
>    new_stack = drop 1 stack                -- Leaves root of redex on top
>    root_of_redex = hd new_stack
>    new_heap = hUpdate heap root_of_redex (NNum (-arg_value))

> primArith (stack, dump, heap, globals, stats) op
>  | length args /= 2 = error "primArith: wrong number of args"
>  | not (isDataNode arg1_node) = ([arg1_addr], new_stack:dump, heap, globals, stats)
>  | not (isDataNode arg2_node) = ([arg2_addr], new_stack:dump, heap, globals, stats)
>  | otherwise = (new_stack, dump, new_heap, globals, stats)
>    where
>    args = getargs heap stack                        -- Should be just two args
>    [arg1_addr,arg2_addr] = args
>    arg1_node = hLookup heap arg1_addr
>    arg2_node = hLookup heap arg2_addr
>    NNum arg1_value = arg1_node
>    NNum arg2_value = arg2_node
>    new_stack = drop 2 stack
>    root_of_redex = hd new_stack
>    new_heap = hUpdate heap root_of_redex (NNum (op arg1_value arg2_value))
```

2.4.3 Printing

```
> showNode (NAp a1 a2) = iConcat [ iStr "NAp ", showAddr a1,
>                                  iStr " ",    showAddr a2
>                          ]
> showNode (NSupercomb name args body) = iStr ("NSupercomb " ++ name)
> showNode (NNum n) = (iStr "NNum ") `iAppend` (iNum n)
```

```
> showNode (NInd a) = (iStr "NInd ") `iAppend` (showAddr a)
> showNode (NPrim name prim) = iStr ("NPrim " ++  name)

> showState (stack, dump, heap, globals, stats)
>  = iConcat [ showStack heap stack, iNewline, showDump dump, iNewline ]

> showDump dump = iConcat [ iStr "Dump depth ", iNum (length dump) ]
```

2.5 Mark 5: Structured data

2.5.1 Conditionals

The Core-language definitions for the boolean operators are:

```
not x = if x False True
or x y = if x True y
xor x y = if x (not y) y
```

The extra `preludeDefs` are:

```
> extraPreludeDefs
>  = [
>      ("False",       [], EConstr 1 0),
>      ("True",        [], EConstr 2 0),
>
>      ("not",         ["x"],     EAp (EAp (EAp (EVar "if") (EVar "x"))
>                                         (EVar "False"))
>                                  (EVar "True")),
>      ("and",         ["x","y"], EAp (EAp (EAp (EVar "if") (EVar "x"))
>                                         (EVar "y"))
>                                  (EVar "False")),
>      ("or",          ["x","y"], EAp (EAp (EAp (EVar "if") (EVar "x"))
>                                         (EVar "True"))
>                                  (EVar "y")),
>      ("xor",         ["x","y"], EAp (EAp (EAp (EVar "if") (EVar "x"))
>                                         (EAp (EVar "not") (EVar "y")))
>                                  (EVar "y")),
>
>      ("MkPair",      [], EConstr 1 2),
>      ("fst",         ["p"], EAp (EAp (EVar "casePair") (EVar "p"))
>                                 (EVar "K")),
>      ("snd",         ["p"], EAp (EAp (EVar "casePair") (EVar "p"))
>                                 (EVar "K1")),
>
>      ("LCons",       [], EConstr 2 2),
>      ("Nil",         [], EConstr 1 0),
```

```
>      ("head",        ["xs"], EAp (EAp (EAp (EVar "caseList") (EVar "xs"))
>                                 (EVar "abort"))
>                              (EVar "K")),
>      ("tail",        ["xs"], EAp (EAp (EAp (EVar "caseList") (EVar "xs"))
>                                 (EVar "abort"))
>                              (EVar "K1"))
>    ]
```

Next comes new definitions of `primitive` and `primitives`:

```
> data Primitive = Negate
>                | Add | Subtract
>                | Multiply | Divide
>                | Greater | GreaterEq
>                | Less | LessEq
>                | Eq | NotEq
>              | PrimConstr Int Int          -- Tag and arity
>                | If
>                | PrimCasePair
>                | PrimCaseList
>                | Abort

> primitives :: ASSOC Name Primitive
> primitives = [  ("negate", Negate),
>                ("+", Add),           ("-", Subtract),
>                ("*", Multiply),      ("/", Divide),
>                (">", Greater),       (">=", GreaterEq),
>                ("<", Less),          ("<=", LessEq),
>                ("==", Eq),           ("~=", NotEq),
>                ("if", If),           ("casePair", PrimCasePair),
>                ("caseList", PrimCaseList),
>                ("abort", Abort)
>              ]
```

The new definition of `instantiateConstr`:

```
> instantiateConstr tag arity heap env
>  = hAlloc heap (NPrim "Cons" (PrimConstr tag arity))
> instantiateAndUpdateConstr tag arity upd_addr heap env
>  = hUpdate heap upd_addr (NPrim "Cons" (PrimConstr tag arity))
```

The new definition of `isDataNode` identifies `NData` as well as `NNum`:

```
> isDataNode :: Node -> Bool
> isDataNode (NNum n)    = True
> isDataNode (NData t c) = True
> isDataNode node        = False
```

`dispatch` gets an extra case:

```
> step state
>  = dispatch (hLookup heap (hd stack))
>    where
>    (stack, dump, heap, globals, stats) = state
>    dispatch (NNum n)                   = numStep  state n
>    dispatch (NInd a)                   = indStep  state a
>    dispatch (NAp a1 a2)                = apStep   state a1 a2
>    dispatch (NSupercomb sc args body) = scStep   state sc args body
>    dispatch (NPrim name prim)          = primStep state prim
>    dispatch (NData tag compts)         = dataStep state tag compts

> dataStep (stack, stack':dump, heap, globals, stats) tag compts
>  = (stack', dump, heap, globals, stats)
```

`primStep` is extended to handle the new primitives:

```
> primStep state Negate   = primNeg state
> primStep state Add      = primArith state (+)
> primStep state Subtract = primArith state (-)
> primStep state Multiply = primArith state (*)
> primStep state Divide   = primArith state (div)
>
> primStep state Greater    = primComp state (>)
> primStep state GreaterEq  = primComp state (>=)
> primStep state Less       = primComp state (<)
> primStep state LessEq     = primComp state (<=)
> primStep state Eq         = primComp state (==)
> primStep state NotEq      = primComp state (/=)
> primStep state (PrimConstr tag arity) = primConstr state tag arity
>
> primStep state If           = primIf state
> primStep state PrimCasePair = primCasePair state
> primStep state PrimCaseList = primCaseList state
>
> primStep state Abort = error "Program abort!"

> primArith state op = primDyadic state op'
>                      where op' (NNum n) (NNum m) = NNum (op n m)
> primComp state op  = primDyadic state op'
>                     where op' (NNum n) (NNum m) | op n m = NData 2 []
>                                                 | otherwise = NData 1 []

> primDyadic (stack, dump, heap, globals, stats) op
>  | length args /= 2 = error "primArith: wrong number of args"
```

```
>   | not (isDataNode arg1_node) = ([arg1_addr], new_stack:dump, heap, globals, stats)
>   | not (isDataNode arg2_node) = ([arg2_addr], new_stack:dump, heap, globals, stats)
>   | otherwise = (new_stack, dump, new_heap, globals, stats)
>     where
>     args = getargs heap stack               -- Should be just two args
>     [arg1_addr,arg2_addr] = args
>     arg1_node = hLookup heap arg1_addr
>     arg2_node = hLookup heap arg2_addr
>     new_stack = drop 2 stack
>     root_of_redex = hd new_stack
>     new_heap = hUpdate heap root_of_redex (op arg1_node arg2_node)

> primIf (stack, dump, heap, globals, stats)
>   | length args < 3 = error "primIf: wrong number of args"
>   | not (isDataNode arg1_node) = ([arg1_addr], new_stack:dump, heap, globals, stats)
>   | otherwise = (new_stack, dump, new_heap, globals, stats)
>     where
>     args = getargs heap stack
>     (arg1_addr:arg2_addr:arg3_addr:rest_args) = args
>     arg1_node = hLookup heap arg1_addr
>     new_stack = drop 3 stack
>     root_of_redex = hd new_stack
>     NData tag [] = arg1_node
>     result_addr | tag == 2  = arg2_addr
>                 | otherwise = arg3_addr
>     new_heap = hUpdate heap root_of_redex (NInd result_addr)

> primCasePair (stack, dump, heap, globals, stats)
>   | length args < 2 = error "primCasePair: wrong number of args"
>   | not (isDataNode arg1_node) = ([arg1_addr], new_stack:dump, heap, globals, stats)
>   | otherwise = (new_stack, dump, new_heap, globals, stats)
>     where
>     args = getargs heap stack
>     (arg1_addr:arg2_addr:rest_args) = args
>     arg1_node = hLookup heap arg1_addr
>     new_stack = drop 2 stack
>     root_of_redex = hd new_stack
>     NData tag [fst,snd] = arg1_node
>     new_heap = hUpdate heap1 root_of_redex (NAp temp_addr snd)
>                where (heap1, temp_addr) = hAlloc heap (NAp arg2_addr fst)

> primCaseList (stack, dump, heap, globals, stats)
>   | length args < 3 = error "primCaseList: wrong number of args"
>   | not (isDataNode arg1_node) = ([arg1_addr], new_stack:dump, heap, globals, stats)
>   | otherwise = (new_stack, dump, new_heap, globals, stats)
>     where
>     args = getargs heap stack
```

```
>    (arg1_addr:arg2_addr:arg3_addr:rest_args) = args
>    arg1_node = hLookup heap arg1_addr
>    new_stack = drop 3 stack
>    root_of_redex = hd new_stack
>    NData tag compts = arg1_node
>    [head,tail] = compts
>    new_heap | tag == 1 = hUpdate heap root_of_redex (NInd arg2_addr)
>             | otherwise = hUpdate heap1 root_of_redex (NAp temp_addr tail)
>               where (heap1, temp_addr) = hAlloc heap (NAp arg3_addr head)


> primConstr (stack, dump, heap, globals, stats) tag arity
>  | length args < arity = error "primConstr: wrong number of args"
>  | otherwise = (new_stack, dump, new_heap, globals, stats)
>    where
>    args = getargs heap stack
>    new_stack = drop arity stack
>    root_of_redex = hd new_stack
>    new_heap = hUpdate heap root_of_redex (NData tag args)
```

Finally, `showNode` is extended to display `NData` nodes:

```
> showNode (NAp a1 a2) = iConcat [ iStr "NAp ", showAddr a1,
>                                  iStr " ",    showAddr a2
>                          ]
> showNode (NSupercomb name args body) = iStr ("NSupercomb " ++ name)
> showNode (NNum n) = (iStr "NNum ") `iAppend` (iNum n)
> showNode (NInd a) = (iStr "NInd ") `iAppend` (showAddr a)
> showNode (NPrim name prim) = iStr ("NPrim " ++  name)
> showNode (NData tag compts)
>      = iConcat [ iStr "NData ", iNum tag, iStr " [",
>                  iInterleave (iStr ",") (map showAddr compts),
>                  iStr "]"
>        ]
```

2.6 Solutions

2.6.1 Mark 1 solutions

2.6.2 Mark 2 solutions

Solution to Exercise 3.7.

```
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = (iStr "Pushglobal ") `iAppend` (iStr f)
> showInstruction (Push n)       = (iStr "Push ")       `iAppend` (iNum n)
> showInstruction (Pushint n)    = (iStr "Pushint ")    `iAppend` (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
> showInstruction (Update n)     = (iStr "Update ")     `iAppend` (iNum n)
> showInstruction (Pop n)        = (iStr "Pop ")        `iAppend` (iNum n)
```

Solution to Exercise 3.8.

```
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>                              where v = hd [n | (n,b) <- globals, a==b]
>                                    globals = getGlobals s
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ",  iStr (showaddr a1),
>                                       iStr " ",    iStr (showaddr a2)]
> showNode s a (NInd a1)     = iConcat [iStr "Ind ", iStr (showaddr a1)]
```

Solution to Exercise 3.9.

We need to introduce two new instructions: `Update` and `Pop`. These two instructions replace the `Slide` instruction in the Mark 1 machine.

```
> dispatch :: Instruction -> GmState -> GmState
> dispatch Unwind         = unwind
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Push n)       = push n
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
> dispatch (Update n)     = update n
> dispatch (Pop n)        = pop n
```

We implement the `Update` function so that it implements update's using indirection nodes.

```
> update :: Int -> GmState -> GmState
> update n state
>  = putHeap heap' (putStack as state)
>    where heap'  = hUpdate (getHeap state) (as !! n) (NInd a)
>          (a:as) = getStack state
```

The `Pop` instruction simply removes `n` items from the stack.

```
> pop :: Int -> GmState -> GmState
> pop n state
>  = putStack (drop n (getStack state)) state
```

The `Unwind` instruction needs changing to handle the case where it discovers an indirection node.

```
> unwind :: GmState -> GmState
> unwind state
>  = newState (hLookup heap a)
>    where
>    (a:as) = getStack state
>    heap   = getHeap state
>    newState (NNum n)      = state
>    newState (NAp a1 a2)   = putCode [Unwind] (putStack (a1:a:as) state)
>    newState (NGlobal n c)
>               | length as < n = error "Unwinding with too few arguments"
>               | otherwise     = putCode c state
>    newState (NInd a1)     = putCode [Unwind] (putStack (a1:as) state)
```

Solution to Exercise 3.10.

```
> compileR e args = compileC e args ++ [Update n, Pop n, Unwind]
>                   where n = length args
```

2.6.3 Mark 3 solutions

Solution to Exercise 3.14.

```
> data Instruction
>     = Slide Int
>     | Alloc Int
>     | Update Int
>     | Pop Int
>     | Unwind
>     | Pushglobal Name
>     | Pushint Int
>     | Push Int
>     | Mkap
```

```
> dispatch :: Instruction -> GmState -> GmState
> dispatch (Slide n)      = slide n
> dispatch Unwind         = unwind
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Push n)       = push n
```

```
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
> dispatch (Update n)     = update n
> dispatch (Pop n)        = pop n
> dispatch (Alloc n)      = alloc n
```

And of course we wish to be able to display these instructions, so we re-define `showInstruction`.

```
> showInstruction (Slide n)      = (iStr "Slide ")      `iAppend` (iNum n)
> showInstruction (Alloc n)      = (iStr "Alloc ")      `iAppend` (iNum n)
> showInstruction (Update n)     = (iStr "Update ")     `iAppend` (iNum n)
> showInstruction (Pop n)        = (iStr "Pop ")        `iAppend` (iNum n)
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = (iStr "Pushglobal ") `iAppend` (iStr f)
> showInstruction (Pushint n)    = (iStr "Pushint ")    `iAppend` (iNum n)
> showInstruction (Push n)       = (iStr "Push ")       `iAppend` (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
```

Solution to Exercise 3.16.

```
> compileLetrec comp defs e args
>  = [Alloc n]           ++
>    compiled defs (n-1) ++
>    comp e newArgs      ++
>    [Slide n]
>    where newArgs = compileArgs defs args
>          n       = length defs
>          compiled []     i = []
>          compiled (d:ds) i = compileC (second d) newArgs ++
>                              [Update i]                  ++
>                              compiled ds (i-1)
```

Solution to Exercise ??. It is an example of "almost circular programming".

You would allocate the memory first and then fill in its value. The difference lies in the fact that we can not return *both* the address *and* the node from the same function.

Solution to Exercise 3.15.

The major interest in the instructions lies in `Alloc`

```
> alloc :: Int -> GmState -> GmState
> alloc n state
>  = putHeap heap' (putStack (as'++getStack state) state)
>    where (heap', as') = allocNodes n (getHeap state)

> push :: Int -> GmState -> GmState
> push n state
>  = putStack ((as !! n): as) state
>    where as = getStack state
```

```
> unwind :: GmState -> GmState
> unwind state
>  = newState (hLookup heap a)
>    where
>    (a:as) = getStack state
>    heap   = getHeap state
>    newState (NNum n)      = state
>    newState (NAp a1 a2)   = putCode [Unwind] (putStack (a1:a:as) state)
>    newState (NGlobal n c)
>               | length as < n = error "Unwinding with too few arguments"
>               | otherwise     = putCode c (putStack as' state)
>                                 where as' = rearrange n heap (a:as)
>    newState (NInd a1)     = putCode [Unwind] (putStack (a1:as) state)
```

2.6.4 Mark 4 solutions

Solution to Exercise 3.21.

```
> getCode (i, stack, dump, heap, globals, stats) = i
> putCode i' (i, stack, dump, heap, globals, stats)
>  = (i', stack, dump, heap, globals, stats)


> getStack (i, stack, dump, heap, globals, stats) = stack
> putStack stack' (i, stack, dump, heap, globals, stats)
>  = (i, stack', dump, heap, globals, stats)


> getHeap (i, stack, dump, heap, globals, stats) = heap
> putHeap heap' (i, stack, dump, heap, globals, stats)
>  = (i, stack, dump, heap', globals, stats)


> getGlobals (i, stack, dump, heap, globals, stats) = globals


> getStats (i, stack, dump, heap, globals, stats) = stats
> putStats stats' (i, stack, dump, heap, globals, stats)
>  = (i, stack, dump, heap, globals, stats')
```

Solution to Exercise 3.22.

```
> showInstruction (Slide n)      = (iStr "Slide ")      `iAppend` (iNum n)
> showInstruction (Alloc n)      = (iStr "Alloc ")      `iAppend` (iNum n)
> showInstruction (Update n)     = (iStr "Update ")     `iAppend` (iNum n)
> showInstruction (Pop n)        = (iStr "Pop ")        `iAppend` (iNum n)
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = (iStr "Pushglobal ") `iAppend` (iStr f)
> showInstruction (Pushint n)    = (iStr "Pushint ")    `iAppend` (iNum n)
```

```
> showInstruction (Push n)       = (iStr "Push ")       `iAppend` (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
> showInstruction Eval           = iStr  "Eval"
> showInstruction Add            = iStr  "Add"
> showInstruction Sub            = iStr  "Sub"
> showInstruction Mul            = iStr  "Mul"
> showInstruction Div            = iStr  "Div"
> showInstruction Neg            = iStr  "Neg"
> showInstruction Eq             = iStr  "Eq"
> showInstruction Ne             = iStr  "Ne"
> showInstruction Lt             = iStr  "Lt"
> showInstruction Le             = iStr  "Le"
> showInstruction Gt             = iStr  "Gt"
> showInstruction Ge             = iStr  "Ge"
> showInstruction (Cond i1 i2)
>  = iConcat [iStr "Cond [2: ", shortShowInstructions 2 i1,
>             iStr ", 1: ",     shortShowInstructions 2 i2, iStr "]"]
```

Solution to Exercise 3.23.

```
> dispatch :: Instruction -> GmState -> GmState
> dispatch Unwind         = unwind
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Push n)       = push n
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
> dispatch (Update n)     = update n
> dispatch (Pop n)        = pop n
> dispatch (Alloc n)      = alloc n
> dispatch Add            = arithmetic2 (+)
> dispatch Sub            = arithmetic2 (-)
> dispatch Mul            = arithmetic2 (*)
> dispatch Div            = arithmetic2 (div)
```

We can also do the `Neg` instruction.

```
> dispatch Neg            = arithmetic1 negate
```

```
> dispatch Eq             = comparison (==)
> dispatch Ne             = comparison (/=)
> dispatch Lt             = comparison (<)
> dispatch Le             = comparison (<=)
> dispatch Gt             = comparison (>)
> dispatch Ge             = comparison (>=)
```

We add the `Eval` instruction, and being a *lazy* functional programmer, I've only used the general case.

```
> dispatch Eval           = evalop
> dispatch (Cond i1 i2)   = cond i1 i2


> cond :: GmCode -> GmCode -> GmState -> GmState
> cond i1 i2 state
>  = putCode (i'++i) (putStack s state)
>    where (a:s) = getStack state
>          i' | hLookup (getHeap state) a == NNum 1 = i1     -- 1 means true
>             | otherwise                           = i2     -- 0 means false
>          i     = getCode state


> evalop :: GmState -> GmState
> evalop state
>  = putCode [Unwind] (putStack [a] (putDump d' state))
>    where (a:s) = getStack state
>          d'    = (getCode state, s): getDump state
```

We must modify the `Unwind` instruction, so that it performs the occasional popping operation.

```
> unwind :: GmState -> GmState
> unwind state
>  = newState (hLookup heap a)
>    where
>    (a:as)       = getStack state
>    heap         = getHeap state
>    ((i',s'):d') = getDump state
>    newState (NNum n)
>     = putCode i' (putStack (a:s') (putDump d' state))
>    newState (NAp a1 a2)
>     = putCode [Unwind] (putStack (a1:a:as) state)
>    newState (NGlobal n c)
>     | length as >= n = putCode c  (putStack rs state)
>     | otherwise  =  putCode i' (putStack (last (a:as):s') (putDump d' state))
>     where rs = rearrange n heap (a:as)
>    newState (NInd a1)
>     = putCode [Unwind] (putStack (a1:as) state)
```

2.6.5 Mark 5 solutions

```
> compileR :: GmCompiler
> compileR (EAp (EAp (EVar op) e1) e2) args
>  | op `elem` binaryOps = compileE (EAp (EAp (EVar op) e1) e2) args ++
>                            [Update n, Pop n, Unwind]
>    where binaryOps = map first builtInDyadic
>          n =  length args
> compileR (EAp (EVar "negate") e) args
```

```
>  = compileE (EAp (EVar "negate") e) args ++ [Update n, Pop n, Unwind]
>    where n = length args
> compileR (EAp (EAp (EAp (EVar "if") e1) e2) e3) args
>   = compileE e1 args ++ [Cond (compileR e2 args) (compileR e3 args)]
> compileR e args = compileC e args ++ [Update n, Pop n, Unwind]
>                   where n = length args


> compileE :: GmCompiler
> compileE (EAp (EAp (EVar op) e1) e2) args
>  | op `elem` binaryOps = compileE e2 args ++ compileE e1 args' ++ [inst]
>    where binaryOps = map first builtInDyadic
>          inst = aLookup builtInDyadic op (error "This can't happen")
>          args' = argOffset 1 args
> compileE (EAp (EVar "negate") e) args
>  = compileE e args ++ [Neg]
> compileE (EAp (EAp (EAp (EVar "if") e1) e2) e3) args
>  = compileE e1 args ++ [Cond (compileE e2 args) (compileE e3 args)]
> compileE (ENum n)      args = [Pushint n]
> compileE (ELet recursive defs e)
>                        args | recursive = compileLetrec compileE defs e args
>                             | otherwise = compileLet    compileE defs e args
> compileE e             args = compileC e args ++ [Eval]
```

2.6.6 Mark 6 solutions

Solution to Exercise 3.32.

```
> showInstruction (Slide n)      = (iStr "Slide ")      `iAppend` (iNum n)
> showInstruction (Alloc n)      = (iStr "Alloc ")      `iAppend` (iNum n)
> showInstruction (Update n)     = (iStr "Update ")     `iAppend` (iNum n)
> showInstruction (Pop n)        = (iStr "Pop ")        `iAppend` (iNum n)
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = (iStr "Pushglobal ") `iAppend` (iStr f)
> showInstruction (Pushint n)    = (iStr "Pushint ")    `iAppend` (iNum n)
> showInstruction (Push n)       = (iStr "Push ")       `iAppend` (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
> showInstruction Eval           = iStr  "Eval"
> showInstruction Add            = iStr  "Add"
> showInstruction Sub            = iStr  "Sub"
> showInstruction Mul            = iStr  "Mul"
> showInstruction Div            = iStr  "Div"
> showInstruction Neg            = iStr  "Neg"
> showInstruction Eq             = iStr  "Eq"
> showInstruction Ne             = iStr  "Ne"
> showInstruction Le             = iStr  "Le"
> showInstruction Lt             = iStr  "Lt"
> showInstruction Ge             = iStr  "Ge"
```

```
> showInstruction Gt             = iStr  "Gt"
> showInstruction (Pack t a)     = ((iStr "Pack ")   `iAppend` (iNum t))
>                                  `iAppend` (iNum a)
> showInstruction (Casejump nis) = (iStr "Casejump ")   `iAppend`
>                                  (showAlternatives nis)
> showInstruction (Split n)      = (iStr "Split ")      `iAppend` (iNum n)
> showInstruction Print          = iStr "Print"

> showInstruction (Cond i1 i2)
>  = iConcat [iStr "Cond [2: ", shortShowInstructions 2 i1,
>             iStr ", 1: ",     shortShowInstructions 2 i2, iStr "]"]

> showAlternatives nis
>  = iConcat [iStr "[",
>             iInterleave (iStr ", ") (map showLabelInstructions nis),
>             iStr "]"]
>    where showLabelInstructions (tag, code)
>           = iConcat [iNum tag, iStr ": ", shortShowInstructions 2 code]
```

Solution to Exercise 3.33.

```
> dispatch :: Instruction -> GmState -> GmState
> dispatch Unwind         = unwind
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Push n)       = push n
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
> dispatch (Update n)     = update n
> dispatch (Pop n)        = pop n
> dispatch (Alloc n)      = alloc n
> dispatch Add            = arithmetic2 (+)
> dispatch Sub            = arithmetic2 (-)
> dispatch Mul            = arithmetic2 (*)
> dispatch Div            = arithmetic2 (div)
> dispatch Neg            = arithmetic1 negate
> dispatch Eq             = comparison (==)
> dispatch Ne             = comparison (/=)
> dispatch Lt             = comparison (<)
> dispatch Le             = comparison (<=)
> dispatch Gt             = comparison (>)
> dispatch Ge             = comparison (>=)
> dispatch Eval           = evalop
> dispatch (Cond i1 i2)   = cond i1 i2
> dispatch (Casejump alts)= casejump alts
> dispatch (Split n)      = split n
> dispatch Print          = gmprint
```

The new instructions, beginning with `Casejump`:

```
> casejump alts state
>  = putCode (i ++ getCode state) state
>    where (NConstr t as) = hLookup (getHeap state) (hd (getStack state))
>          i = aLookup alts t (error ("No case for constructor" ++ show t))
```

Next consider `Split`

```
> split :: Int -> GmState -> GmState
> split j state
>  = putStack (as++s) state
>    where (NConstr t as) = hLookup (getHeap state) a
>          (a:s)          = getStack state
```

Next `Print`

```
> gmprint :: GmState -> GmState  -- gmprint not print, to avoid name clash
> gmprint state
>  = newState (hLookup (getHeap state) a) state
>    where
>    newState (NConstr t as) = putOutput ("Pack{"++show t++","++ show(length as)++"}"). -- P
>                              putCode (printcode (length as) ++ getCode state) .
>                               putStack (as++s)
>    newState (NNum n)       = putOutput (show n) . putStack s
>    newState n              = error "Print of non data structure"
>    (a:s) = getStack state

> printcode 0     = []
> printcode (n+1) = Eval: Print: printcode n
```

Finally, `Pack`

```
> pack :: Int -> Int -> GmState -> GmState
> pack t a state
>  = putHeap heap' (putStack (addr: drop a s) state)
>    where s = getStack state
>          (heap', addr) = hAlloc (getHeap state) (NConstr t (take a s))
```

We also need to modify `Unwind` (again).

```
> unwind :: GmState -> GmState
> unwind state
>  = newState (hLookup heap a)
>    where
>    (a:as)       = getStack state
>    heap         = getHeap state
>    ((i',s'):d') = getDump state
```

```
>    newState (NNum n)
>     = putCode i' (putStack (a:s') (putDump d' state))
>    newState (NAp a1 a2)
>     = putCode [Unwind] (putStack (a1:a:as) state)
>    newState (NGlobal n c)
>     | length as >= n = putCode c  (putStack rs state)
>     | otherwise      = putCode i' (putStack (last (a:as):s') (putDump d' state))
>       where rs = rearrange n heap (a:as)
>    newState (NInd a1)
>     = putCode [Unwind] (putStack (a1:as) state)
>    newState (NConstr t as)
>     = putCode i' (putStack (a:s') (putDump d' state))
```

Solution to Exercise ??.

```
> getCode (o, i, stack, dump, heap, globals, stats) = i
> putCode i' (o, i, stack, dump, heap, globals, stats)
>  = (o, i', stack, dump, heap, globals, stats)

> getStack (o, i, stack, dump, heap, globals, stats) = stack
> putStack stack' (o, i, stack, dump, heap, globals, stats)
>  = (o, i, stack', dump, heap, globals, stats)

> getDump (o, i, stack, dump, heap, globals, stats) = dump
> putDump dump' (o, i, stack, dump, heap, globals, stats)
>  = (o, i, stack, dump', heap, globals, stats)

> getHeap (o, i, stack, dump, heap, globals, stats) = heap
> putHeap heap' (o, i, stack, dump, heap, globals, stats)
>  = (o, i, stack, dump, heap', globals, stats)

> getGlobals (o, i, stack, dump, heap, globals, stats) = globals
> putGlobals globals' (o, i, stack, dump, heap, globals, stats) -- KH for exercise
>  = (o, i, stack, dump, heap, globals', stats)

> getStats (o, i, stack, dump, heap, globals, stats) = stats
> putStats stats' (o, i, stack, dump, heap, globals, stats)
>  = (o, i, stack, dump, heap, globals, stats')
```

Solution to Exercise ??.

```
> compile :: CoreProgram -> GmState
> compile program
>  = ([], initialCode, [], [], heap, globals, statInitial)
>    where (heap, globals) = buildInitialHeap program
```

```
> initialCode :: GmCode
> initialCode = [Pushglobal "main", Eval, Print]

> compileC :: GmCompiler
> compileC (EConstr t 0) args = [Pack t 0]
> compileC (EVar v)      args | v `elem` (aDomain args) = [Push n]
>                             | otherwise = [Pushglobal v]
>                               where n = aLookup args v (error "")
> compileC (ENum n)      args = [Pushint n]
> compileC (ELet recursive defs e)
>                        args | recursive = compileLetrec compileC defs e args
>                             | otherwise = compileLet    compileC defs e args
> compileC (EAp e1 e2)   args
>  | saturatedCons spine = compileCS (reverse spine) args
>  | otherwise = compileC e2 args ++ compileC e1 (argOffset 1 args) ++ [Mkap]
>    where spine = makeSpine (EAp e1 e2)
>          saturatedCons (EConstr t a:es) = a == length es
>          saturatedCons (e:es)           = False

> makeSpine (EAp e1 e2) = makeSpine e1 ++ [e2]
> makeSpine e           = [e]

> compileCS [EConstr t a] args = [Pack t a]
> compileCS (e:es)        args = compileC  e args ++
>                                compileCS es (argOffset 1 args)

> compileE :: GmCompiler
> compileE (EAp (EAp (EVar op) e1) e2) args
>  | op `elem` binaryOps = compileE e2 args ++ compileE e1 args' ++ [inst]
>    where binaryOps = map first builtInDyadic
>          inst = aLookup builtInDyadic op (error "This can't happen")
>          args' = argOffset 1 args
> compileE (EAp (EVar "negate") e) args
>  = compileE e args ++ [Neg]
> compileE (EAp (EAp (EAp (EVar "if") e1) e2) e3) args
>  = compileE e1 args ++ [Cond (compileE e2 args) (compileE e3 args)]
> compileE (ENum n)      args = [Pushint n]
> compileE (ELet recursive defs e)
>                        args | recursive = compileLetrec compileE defs e args
>                             | otherwise = compileLet    compileE defs e args
> compileE (ECase e as)  args = compileE e args ++
>                               [Casejump (compileAlts compileE' as args)]
> compileE e             args = compileC e args ++ [Eval]
```

2.6.7 Mark 6: solutions

Solution to Exercise ??.

```
> getOutput (o, i, stack, dump, vstack, heap, globals, stats) = o
> putOutput o' (o, i, stack, dump, vstack, heap, globals, stats)
>  = (o', i, stack, dump, vstack, heap, globals, stats)

> getCode (o, i, stack, dump, vstack, heap, globals, stats) = i
> putCode i' (o, i, stack, dump, vstack, heap, globals, stats)
>  = (o, i', stack, dump, vstack, heap, globals, stats)

> getStack (o, i, stack, dump, vstack, heap, globals, stats) = stack
> putStack stack' (o, i, stack, dump, vstack, heap, globals, stats)
>  = (o, i, stack', dump, vstack, heap, globals, stats)

> getDump (o, i, stack, dump, vstack, heap, globals, stats) = dump
> putDump dump' (o, i, stack, dump, vstack, heap, globals, stats)
>  = (o, i, stack, dump', vstack, heap, globals, stats)

> getHeap (o, i, stack, dump, vstack, heap, globals, stats) = heap
> putHeap heap' (o, i, stack, dump, vstack, heap, globals, stats)
>  = (o, i, stack, dump, vstack, heap', globals, stats)

> getGlobals (o, i, stack, dump, vstack, heap, globals, stats) = globals
> putGlobals globals' (o, i, stack, dump, heap, globals, stats) -- KH for exercise
>  = (o, i, stack, dump, heap, globals', stats)

> getStats (o, i, stack, dump, vstack, heap, globals, stats) = stats
> putStats stats' (o, i, stack, dump, vstack, heap, globals, stats)
>  = (o, i, stack, dump, vstack, heap, globals, stats')

> primitive1 :: (Int -> Int)             -- operator
>               -> (GmState -> GmState) -- state transition
>
> primitive1 op state
>  = putVStack (op n: ns) state
>    where (n:ns) = getVStack state

> primitive2 :: (Int -> Int -> Int)        -- operator
>               -> (GmState -> GmState)   -- state transition
>
> primitive2 op state
>  = putVStack (op n0 n1: ns) state
>    where (n0:n1:ns) = getVStack state

> arithmetic1 = primitive1

> arithmetic2 = primitive2
```

```
> comparison op = primitive2 op'
>                 where op' x y | op x y = 2    -- 2 is the tag of True
>                               | otherwise = 1 -- 1 is the tag of False
```

Solution to Exercise ??.

```
> showInstruction (Slide n)      = (iStr "Slide ")      `iAppend` (iNum n)
> showInstruction (Alloc n)      = (iStr "Alloc ")      `iAppend` (iNum n)
> showInstruction (Update n)     = (iStr "Update ")     `iAppend` (iNum n)
> showInstruction (Pop n)        = (iStr "Pop ")        `iAppend` (iNum n)
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = (iStr "Pushglobal ") `iAppend` (iStr f)
> showInstruction (Pushint n)    = (iStr "Pushint ")    `iAppend` (iNum n)
> showInstruction (Push n)       = (iStr "Push ")       `iAppend` (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
> showInstruction Eval           = iStr  "Eval"
> showInstruction Add            = iStr  "Add"
> showInstruction Sub            = iStr  "Sub"
> showInstruction Mul            = iStr  "Mul"
> showInstruction Div            = iStr  "Div"
> showInstruction Neg            = iStr  "Neg"
> showInstruction Eq             = iStr  "Eq"
> showInstruction Ne             = iStr  "Ne"
> showInstruction Le             = iStr  "Le"
> showInstruction Lt             = iStr  "Lt"
> showInstruction Ge             = iStr  "Ge"
> showInstruction Gt             = iStr  "Gt"
> showInstruction (Pack t a)     = ((iStr "Pack ")   `iAppend` (iNum t))
>                                                    `iAppend` (iNum a)
> showInstruction (Casejump nis) = (iStr "Casejump ")   `iAppend`
>                                  (showAlternatives nis)
> showInstruction (Split n)      = (iStr "Split ")      `iAppend` (iNum n)
> showInstruction Print          = iStr "Print"
> showInstruction Mkbool         = iStr "Mkbool"
> showInstruction Mkint          = iStr "Mkint"
> showInstruction Get            = iStr "Get"
> showInstruction (Pushbasic n)  = iStr "Pushbasic" `iAppend` (iNum n)
> showInstruction Return         = iStr "Return"
> showInstruction (Cond t f)     = (iStr "Cond ")   `iAppend`
>                                  (showAlternatives [(2,t),(1,f)])
```

```
> dispatch :: Instruction -> GmState -> GmState
> dispatch Unwind         = unwind
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Push n)       = push n
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
```

```
> dispatch (Update n)     = update n
> dispatch (Pop n)        = pop n
> dispatch (Alloc n)      = alloc n
> dispatch Add            = arithmetic2 (+)
> dispatch Sub            = arithmetic2 (-)
> dispatch Mul            = arithmetic2 (*)
> dispatch Div            = arithmetic2 (div)
> dispatch Neg            = arithmetic1 negate
> dispatch Eq             = comparison (==)
> dispatch Ne             = comparison (/=)
> dispatch Lt             = comparison (<)
> dispatch Le             = comparison (<=)
> dispatch Gt             = comparison (>)
> dispatch Ge             = comparison (>=)
> dispatch Eval           = evalop
> dispatch (Cond i1 i2)   = cond i1 i2
> dispatch (Casejump alts)= casejump alts
> dispatch (Split n)      = split n
> dispatch Print          = gmprint
> dispatch Mkbool         = mkbool
> dispatch Mkint          = mkint
> dispatch Get            = get
> dispatch (Pushbasic n)  = pushbasic n
> dispatch Return         = gmreturn
> dispatch (Pack t a)     = pack t a -- KH

> gmreturn :: GmState -> GmState
> gmreturn state
>  = putDump d (putStack (a:s') (putCode i' state))
>    where (a:s)       = getStack state
>          ((i',s'):d) = getDump state
```

We also change the `Eval` instruction.

```
> step (o, Eval:i, a:s, d, v, heap, globals, stats)
>  = newState (hLookup heap a)
>    where
>    newState (NNum n)       = (o, i, a:s, d, v, heap, globals, stats)
>    newState (NConstr t as) = (o, i, a:s, d, v, heap, globals, stats)
>    newState n              = (o, [Unwind], [a], (i,s):d, v, heap, globals, stats)
> step state = dispatch i (putCode is state)
>              where (i:is) = getCode state

> mkbool :: GmState -> GmState
> mkbool state
>  = putStack (a:getStack state) (putVStack v (putHeap heap' state))
>    where (heap',a) = hAlloc (getHeap state) (NConstr x [])
>          (x:v)     = getVStack state
```

```
> mkint :: GmState -> GmState
> mkint state
>  = putStack (a:getStack state) (putVStack v (putHeap heap' state))
>    where (heap',a) = hAlloc (getHeap state) (NNum x)
>          (x:v)     = getVStack state

> get :: GmState -> GmState
> get state
>  = newState (hLookup (getHeap state) a) (putStack s state)
>    where newState (NConstr n []) = putVStack (n:v)
>          newState (NNum n)       = putVStack (n:v)
>          newState (NInd a')      = newState (hLookup (getHeap state) a')
>          newState n              = error "Get of a non-number or bool"
>          v = getVStack state
>          (a:s) = getStack state

> cond :: GmCode -> GmCode -> GmState -> GmState
> cond t f state
> {- KH -- this looks bogus -- we should have NConstr 1 0 on the A Stack!
>  = putCode i (putVStack v state)
>    where (x:v) = getVStack state
>          i | x == 1 = f ++ getCode state   -- 1 is the tag of False
>            | otherwise = t ++ getCode state        -- 2 is the tag of True
> KH -}
>  = casejump [(1,f),(2,t)] state
> {- KH Section ends here! -}

> pushbasic :: Int -> GmState -> GmState
> pushbasic n state
>  = putVStack (n:getVStack state) state
```

We also need to adjust comparison, so that it now returns a "boolean" on the value stack.

```
> data Instruction = Slide Int
>                  | Alloc Int
>                  | Update Int
>                  | Pop Int
>                  | Unwind
>                  | Pushglobal Name
>                  | Pushint Int
>                  | Push Int
>                  | Mkap
>                  | Eval
>                  | Add
>                  | Sub
>                  | Mul
```

```
>                 | Div
>                 | Neg
>                 | Eq
>                 | Ne
>                 | Le
>                 | Lt
>                 | Ge
>                 | Gt
>                 | Negate
>                 | Pack Int Int
>                 | Casejump [(Int,GmCode)]
>                 | Split Int
>                 | Print
>                 | Pushbasic Int
>                 | Mkbool
>                 | Mkint
>                 | Get
>                 | Return
>                 | Cond GmCode GmCode
```

The cases for local definitions and casejumps remain the same as they were in the Mark 5 machine.

```
> compileR :: GmCompiler
> compileR (EAp (EAp (EVar op) e1) e2) args
>  | op `elem` binaryOps = compileE (EAp (EAp (EVar op) e1) e2) args ++
>          [Update n, Pop n, Unwind]
>    where binaryOps = map first builtInDyadic
>          n = length args
> compileR (EAp (EVar "negate") e) args
>  = compileE (EAp (EVar "negate") e) args ++ [Update n, Pop n, Unwind]
>    where n = length args
> compileR (EAp (EAp (EAp (EVar "if") e1) e2) e3) args
>  = compileE e1 args ++ [Cond (compileR e2 args) (compileR e3 args)]
> compileR (ELet recursive defs e)
>                         env | recursive = compileLetrec compileR defs e env
>                             | otherwise = compileLet    compileR defs e env
> compileR (ECase e as)  env = compileE e env ++
>                              [Casejump (compileAlts compileR' as env)]
> compileR e args = compileC e args ++ [Update n, Pop n, Unwind]
>                   where n = length args
```

We will also need `compileR'` to compile code for alternatives that are compiled using the $\mathcal{R}$-scheme.

```
> compileR' n expr env = [Split n] ++ compileR expr env
```

```
> compileE :: GmCompiler
> compileE (EAp (EAp (EVar op) e1) e2) args
>  | op `elem` ["+", "-", "*", "/"]
>    = compileB (EAp (EAp (EVar op) e1) e2) args ++ [Mkint]
>  | op `elem` ["==", "~+", "<", "<=", ">", ">="]
>    = compileB (EAp (EAp (EVar op) e1) e2) args ++ [Mkbool]
> compileE (EAp (EVar "negate") e) args
>  = compileB (EAp (EVar "negate") e) args ++ [Mkint]
> compileE (EAp (EAp (EAp (EVar "if") e1) e2) e3) args
>  = compileB e1 args ++ [Cond (compileE e2 args) (compileE e3 args)]
> compileE (ENum n)      args = [Pushint n]
> compileE (ELet recursive defs e)
>                        args | recursive = compileLetrec compileE defs e args
>                             | otherwise = compileLet    compileE defs e args
> compileE (ECase e as)  args = compileE e args ++
>                               [Casejump (compileAlts compileE' as args)]
> compileE e             args = compileC e args ++ [Eval]
```

In the Mark 5 machine, we will only work witha restricted set of operators. We record the salient information in `strictOperators`.

```
> strictOperators :: ASSOC Name (Instruction, Int)
> strictOperators
>  = [("+", (Add, 2)), ("-", (Sub, 2)), ("*", (Mul, 2)), ("/", (Div, 2)),
>     ("negate", (Neg, 1)),
>     ("==", (Eq, 2)), ("~=", (Ne, 2)), (">=", (Ge, 2)),
>     (">",  (Gt, 2)), ("<=", (Le, 2)), ("<",  (Lt, 2))]


> compileB (EAp (EAp (EAp (EVar "if") e1) e2) e3) env
>  = compileB e1 env ++
>    [Cond (compileB e2 env) (compileB e3 env)]


> compileB (ENum n)    env = [Pushbasic n]


> compileB (EAp (EAp (EVar "+")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Add]
> compileB (EAp (EAp (EVar "-")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Sub]
> compileB (EAp (EAp (EVar "*")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Mul]
> compileB (EAp (EAp (EVar "/")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Div]


> compileB (EAp (EAp (EVar "==")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Eq]
> compileB (EAp (EAp (EVar "~=")  e1) e2) env
```

```
>  = compileB e2 env ++ compileB e1 env ++ [Ne]
> compileB (EAp (EAp (EVar ">")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Gt]
> compileB (EAp (EAp (EVar ">=")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Ge]
> compileB (EAp (EAp (EVar "<")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Lt]
> compileB (EAp (EAp (EVar "<=")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Le]


> compileB (EAp (EVar "negate")  e) env
>  = compileB e env ++ [Neg]
```

It is possible to compile code to propagate the $\mathcal{B}$-scheme through local definitions.

```
> compileB (ELet recursive defs e)
>                       env | recursive = compileLetrec compileB defs e env
>                           | otherwise = compileLet    compileB defs e env
```

Finally we give a default case, which is used if none of the above special cases arises.

```
> compileB e             env = compileE e env ++ [Get]
```

This concludes the presentation of the Mark 7 machine.

Chapter 3

TIM: the three-instruction machine

3.1 Mark 2: Adding arithmetic

We present the extra code required for adding arithmetic, incorporating everything up to Exercise 4.8.

Compilation rules

Here are the compilation schemes corresponding to Figure 4.2. The only additions are extra rules for conditionals which are discussed in Exercises 4.7 and 4.8.

```
> compileR e env | isBasicOp e = compileB e env [Return]
> compileR e env | isCondOp e = compileB kCond env [Cond il1 il2]
>                  where
>                  (kCond, kThen, kElse) = unpackCondOp e
>                  il1 = compileR kThen env
>                  il2 = compileR kElse env
>
> compileR (EAp e1 e2) env = Push (compileA e2 env) : compileR e1 env
> compileR (EVar v)    env = [Enter (compileA (EVar v) env)]
> compileR (ENum n)    env = [PushV (IntVConst n), Return]
> compileR e           env = error "compileR: can't do this yet"
```

The $\mathcal{B}[\![]\!]$ compilation scheme has a case for each form of specially-recognised operation.

```
> compileB e env cont
>  | isBinOp e = compileB e2 env (compileB e1 env (Op op : cont))
>    where
>    (e1,op,e2) = unpackBinOp e
>
> compileB e env cont
>  | isUnOp e = compileB e1 env (Op op : cont)
>    where
```

```
>    (op,e1) = unpackUnOp e
>
> compileB (ENum n) env cont = PushV (IntVConst n) : cont
> compileB e        env cont = Push  (Code cont) : compileR e env
```

New state transitions

New state transitions need to be provided for the new instructions:

```
> step ([Return], fptr, (instr',fptr'):stack, vstack,
>    dump, heap, cstore, stats)
>  = (instr', fptr', stack, vstack, dump, heap, cstore, stats)

> step ((PushV FramePtr:instr), (FrameInt n), stack, vstack,
>    dump, heap, cstore, stats)
>  = (instr, FrameInt n, stack, n:vstack, dump, heap, cstore, stats)

> step ((PushV (IntVConst n):instr), fptr, stack, vstack,
>    dump, heap, cstore, stats)
>  = (instr, fptr, stack, n:vstack, dump, heap, cstore, stats)

> step ([Cond il1 il2], fptr, stack, b:vstack, dump, heap, cstore, stats)
>  = (instr', fptr, stack, vstack, dump, heap, cstore, stats)
>    where
>    instr' | numToBool b = il1     -- True
>           | otherwise = il2       -- False

> step ((Op op:instr), fptr, stack, vstack, dump, heap, cstore, stats)
>  = (instr, fptr, stack, performOp op vstack, dump, heap, cstore, stats)
```

All the old transitions are exactly as before, except that `IntConst` is now a possible `timAMode` for `Push` and `Enter`:

```
> step ((Take n:instr), fptr, stack, vstack, dump, heap, cstore,stats)
>  | length stack >= n = (instr, fptr', drop n stack, vstack, dump, heap', cstore, stats)
>  | otherwise = error "Too few args for Take instruction"
>    where (heap', fptr') = fAlloc heap (take n stack)

> step ([Enter am], fptr, stack, vstack, dump, heap, cstore, stats)
>  = (instr', fptr', stack, vstack, dump, heap, cstore, stats)
>    where (instr',fptr') = amToClosure am fptr heap cstore

> step ((Push am:instr), fptr, stack, vstack, dump, heap, cstore, stats)
>  = (instr, fptr, amToClosure am fptr heap cstore : stack,
>     vstack, dump, heap, cstore, stats)
```

We define a stub function, `step3` (for the Mark 3 machine), to fall through to, so that subsequent increments to the `step` function can be done without copying all the code again.

```
> step state = step3 state
> step3 state = error "Unknown instruction"
```

Built-in functions

Auxilliary functions deal with recognising, unpacking, executing and showing primitives:

```
> isBasicOp :: CoreExpr -> Bool
> isBasicOp e = isBinOp e || isUnOp e
>
> isBinOp :: CoreExpr -> Bool
> isBinOp (EAp (EAp (EVar op) e1) e2) = isOp op
> isBinOp e = False
>
> unpackBinOp (EAp (EAp (EVar op) e1) e2) = (e1, mkOp op, e2)
>
> isUnOp :: CoreExpr -> Bool
> isUnOp (EAp (EVar op) e1) = isOp op
> isUnOp e = False
> unpackUnOp (EAp (EVar op) e1) = (mkOp op, e1)
>
> isCondOp :: CoreExpr -> Bool
> isCondOp (EAp (EAp (EAp (EVar "if") kCond) kThen) kElse) = True
> isCondOp e = False
> unpackCondOp (EAp (EAp (EAp (EVar "if") kCond) kThen) kElse) = (kCond,kThen,kElse)


> isOp :: Name -> Bool
> isOp op_name = elem op_name (map fst builtIns)
>
> mkOp :: Name -> Op
> mkOp op_name = aLookup builtIns op_name (error ("Unknown operator "
>                                                 ++ op_name))
>
> showOp :: Op -> Iseq
> showOp op = iStr (hd [name | (name, op') <- builtIns, op == op']) -- KH


> builtIns = [ ("+", Add), ("-", Sub), ("*", Mult), ("/", Div),
>              ("negate", Neg),
>              (">", Gr), (">=", GrEq), ("<", Lt), ("<=", LtEq),
>              ("==", Eq), ("~=", NotEq)
>            ]


> performOp :: Op -> TimValueStack -> TimValueStack
```

```
> performOp Add   (a:b:vs)    = (a+b:vs)
> performOp Sub   (a:b:vs)    = (a-b:vs)
> performOp Mult  (a:b:vs)    = (a*b:vs)
> performOp Div   (a:b:vs)  = (a `div` b:vs)
> performOp Neg   (a:vs)      = (-a:vs)
> performOp Gr    (a:b:vs)    = (boolToNum (a>b) : vs)
> performOp GrEq  (a:b:vs)    = (boolToNum (a>=b) : vs)
> performOp Lt    (a:b:vs)    = (boolToNum (a<b) : vs)
> performOp LtEq  (a:b:vs)    = (boolToNum (a<=b) : vs)
> performOp Eq    (a:b:vs)  = (boolToNum (a==b) : vs)
> performOp NotEq (a:b:vs)  = (boolToNum (a/=b) : vs)
```

`boolToNum` and `numToBool` give the encoding of booleans as numbers.

```
> boolToNum True = 2
> boolToNum False = 1
>
> numToBool 1 = False
> numToBool 2 = True
> numToBool n = error ("numToBool: unexpected number " ++ shownum n)
```

New printing facilites

Extra functionality needs to be provided in `showInstruction` for displaying the new instructions and addressing modes.

```
> showInstruction d (Take m)  = (iStr "Take ")  `iAppend` (iNum m)
> showInstruction d (Enter x) = (iStr "Enter ") `iAppend` (showArg d x)
> showInstruction d (Push x)  = (iStr "Push ")  `iAppend` (showArg d x)
> showInstruction d (PushV x) = (iStr "PushV ")  `iAppend` (showVArg d x)
> showInstruction d Return    = (iStr "Return")
> showInstruction d (Op op)   = (iStr "Op ") `iAppend` (showOp op)
> showInstruction d (Cond il1 il2)
>  = iConcat [iStr "Cond ",
>             iIndent (iConcat [showInstructions d il1, iNewline,
>                               showInstructions d il2])
>    ]
```

Just as with `step` we fall through the Mark 3 machine if there is no match. For now, `showInstructions3` just fails.

```
> showInstruction d i = showInstruction3 d i
> showInstruction3 d i = error "Unknown instruction"
```

```
> showVArg d FramePtr = iStr "FramePtr"
> showVArg d (IntVConst n) = iStr "IntVConst " `iAppend` (iNum n)
```

showValueStack is redefined to print the value stack.

```
> showValueStack vstack
>  = iConcat [iStr "Val stack: [",
>             iIndent (iInterleave (iStr ",") (map iNum vstack)),
>             iStr "]", iNewline
>    ]
```

3.2 Mark 3: Letrec expressions

We need a new instruction data type, adding an extra argument to Take, and adding the new Move instruction.

```
> data Instruction = Take Int Int
>                  | Push TimAMode
>                  | PushV ValueAMode
>                  | Enter TimAMode
>                  | Return
>                  | Op Op
>                  | Cond [Instruction] [Instruction]
>                  | Move Int TimAMode
```

3.2.1 Compilation

The modifications to the compiler are mostly straightforward. compileSC generates a modified Take instruction.

```
> compileSC env (name, args, body)
>  = (name, Take d no_of_args : instructions)
>    where
>    (d, instructions) = compileR body (arg_env ++ env) no_of_args
>    arg_env = zip2 args (map Arg [1..])
>    no_of_args = length args
```

The interesting bit of compileR is the case for let(rec) expressions.

```
> compileR (ELet isrec defs body) env d
>  = (d', move_instrs ++ il)
>    where
>    (dn, move_instrs) = mapAccuml
>                              make_move_instr (d+no_of_defs)
>                              (zip2 defs frame_slots)
>    (d', il) = compileR body new_env dn
>
>    new_env = zip2 names (map mkIndMode frame_slots) ++ env
```

```
>    no_of_defs = length defs
>    names = [name | (name,rhs) <- defs]
>    frame_slots = [d+1..d+no_of_defs]
>
>    make_move_instr d ((name, rhs), frame_slot)
>      = (d', Move frame_slot am) where (d', am) = compileA rhs rhs_env d
>
>    rhs_env | not isrec = env
>            | isrec     = new_env
```

The other cases are as before, extended with additional plumbing. The main other interesting point is the special case for conditionals. Since only one branch can execute, we can overlap the frame slots used in each branch.

```
> compileR e env d | isBasicOp e = compileB e env (d, [Return])
> compileR e env d | isCondOp e = compileB kCond env (dmax, [Cond il1 il2])
>                     where
>                     (kCond, kThen, kElse) = unpackCondOp e
>                     (d1, il1) = compileR kThen env d
>                     (d2, il2) = compileR kElse env d
>                     dmax = maximum [d1,d2]
>
> compileR (EAp e1 e2) env d
>  = (d2, Push am : il)
>    where
>    (d1, am) = compileA e2 env d
>    (d2, il) = compileR e1 env d1
>
> compileR (EVar v)    env d = (d1, [Enter am])
>                              where
>                              (d1,am) = compileA (EVar v) env d
>
> compileR (ENum n)    env d = (d, [PushV (IntVConst n), Return])
> compileR e           env d = error "compileR: can't do this yet"
```

The modifications to `compileA` add the extra plumbing.

```
> compileA (EVar v) env d = (d, aLookup env v (error ("Unknown variable "
>                                                     ++ v)))
>
> compileA (ENum n) env d = (d, IntConst n)
> compileA e env d = (d1, Code il) where (d1, il) = compileR e env d
```

The $\mathcal{B}[\![]\!]$ compilation scheme has a case for each form of specially-recognised operation.

```
> compileB e env (d,cont)
>  | isBinOp e = compileB e2 env (compileB e1 env (d, Op op:cont))
```

```
>    where
>    (e1,op,e2) = unpackBinOp e
>
> compileB e env (d,cont)
>  | isUnOp e = compileB e1 env (d, Op op:cont)
>    where
>    (op,e1) = unpackUnOp e
>
> compileB (ENum n) env (d,cont) = (d, PushV (IntVConst n) : cont)
> compileB e env (d, cont) = (d1, Push (Code cont) : il)
>                            where
>                            (d1, il) = compileR e env d
```

3.2.2 New state transitions

We express the new state transitions by redefining `step3` which was left as a fall-through from the Mark 2 machine. We need to add transitions for `Take` and `Move`.

The `Take` instruction initialises the as-yet-unused frame slots with `dummy_closure`. In a real implementation, of course, these slots could be left uninitialised.

```
> step3 ((Take tot n : instr), fptr, stack, vstack, dump, heap, cstore,stats)
>  | length stack >= n = (instr, fptr', drop n stack, vstack, dump, heap', cstore, stats)
>  | otherwise = error "Too few args for Take instruction"
>    where (heap', fptr') = fAlloc heap frame_cts
>                           where
>                           frame_cts = take n stack ++
>                                       take (tot-n) (repeat dummy_closure)
>                           dummy_closure = ([], FrameNull)
```

The `Move` instruction is straightforward. It uses `amToClosure` again.

```
> step3 ((Move n am : instr), fptr, stack, vstack, dump, heap, cstore,stats)
>  = (instr, fptr, stack, vstack, dump, heap', cstore, stats)
>    where heap' = fUpdate heap fptr n (amToClosure am fptr heap cstore)
```

Finally, we add a fall-through case for the Mark 4 machine:

```
> step3 state = step4 state


> step4 state = error "Can't do this instruction yet"
```

3.2.3 Printing

We need extra cases for `showInstruction3`, with the usual fall-through to `showInstruction4`.

```
> showInstruction3 d (Take tot n)
>  = iConcat [iStr "Take ", iNum tot, iStr " ", iNum n]
> showInstruction3 d (Move n am)
>  = iConcat [iStr "Move ", iNum n, iStr " ", showArg d am]
> showInstruction3 d i = showInstruction4 d i

> showInstruction4 d i = error "Unknown instruction"
```

3.3 Mark 4: Updating

We present only the completed code for the developments of this section.

We need to extend the `instruction` type yet again, to add the instructions `PushMarker` and `UpdateMarkers`.

```
> data Instruction = Take Int Int
>                  | Push TimAMode
>                  | PushV ValueAMode
>                  | Enter TimAMode
>                  | Return
>                  | Op Op
>                  | Cond [Instruction] [Instruction]
>                  | Move Int TimAMode
>                  | PushMarker Int
>                  | UpdateMarkers Int
```

3.3.1 Compilation

The only change to `compileSC` is to add the `UpdateMarkers` instruction before the `Take`.

```
> compileSC env (name, args, body)
>  = (name, UpdateMarkers no_of_args : Take d no_of_args : instructions)
>    where
>    (d, instructions) = compileR body (arg_env ++ env) no_of_args
>    arg_env = zip2 args (map Arg [1..])
>    no_of_args = length args
```

The differences to `compileR` are:

- it builds self-updating closures for `let(rec)`-bound variables, and uses `compileAL` to generate the right-hand side.
- it generates non-updating indirection addressing modes for of `let(rec)`-bound variables.
- it makes non-trivial arguments of function calls behave just as if they were `let`-bound to a new variable, and that variable passed in the call, using `compileAL`.

```
> compileR (ELet isrec defs body) env d
>  = (d', move_instrs ++ il)
>    where
>    (dn, move_instrs) = mapAccuml
>                              make_move_instr (d+no_of_defs)
>                              (zip2 defs frame_slots)
>    (d', il) = compileR body new_env dn
>
>    new_env = zip2 names (map mkIndMode frame_slots) ++ env
>    no_of_defs = length defs
>    names = [name | (name,rhs) <- defs]
>    frame_slots = [d+1..d+no_of_defs]
>
>    make_move_instr d ((name, rhs), frame_slot)
>      = (d', Move frame_slot am)
>        where (d', am) = compileAL rhs frame_slot rhs_env d
>
>    rhs_env | not isrec = env
>            | isrec     = new_env
>
> compileR e env d | isBasicOp e = compileB e env (d, [Return])
> compileR e env d | isCondOp e = compileB kCond env (dmax, [Cond il1 il2])
>                     where
>                     (kCond, kThen, kElse) = unpackCondOp e
>                     (d1, il1) = compileR kThen env d
>                     (d2, il2) = compileR kElse env d
>                     dmax = maximum [d1,d2]
>
> compileR (EAp e1 e2) env d
>  | isAtomicExpr e2 = (d1, Push (compileA e2 env) : il)
>    where
>    (d1, il) = compileR e1 env d
>
> compileR (EAp e1 e2) env d -- Non-atomic argument e2
>  = (d2, Move (d+1) am : Push (Code [Enter (Arg (d+1))]) : il)
>    where
>    (d1, am) = compileAL e2 (d+1) env (d+1)
>    (d2, il) = compileR e1 env d1
>
> compileR (EVar v)    env d = (d, [Enter (compileA (EVar v) env)])
>
> compileR (ENum n)    env d = (d, [PushV (IntVConst n), Return])
> compileR e           env d = error "compileR: can't compile this!"
```

The auxiliary function `idAtomicExpr` picks out variables and constants. It is called from the `EAp` case of `compileR`.

`compileB` is unchanged. `compileA` no longer needs a case for anything other than variables and numbers, nor does it need a frame-usage argument or result.

```
> compileA (EVar v) env = aLookup env v (error ("Unknown variable "
>                                               ++ v))
> compileA (ENum n) env = IntConst n
```

`compileAL` is new. It follows the compilation scheme given in Figure 4.4.

```
> compileAL (ENum n) upd_slot env d = (d, IntConst n)
> compileAL e        upd_slot env d = (d1, Code (PushMarker upd_slot: il))
>                                     where (d1, il) = compileR e env d
```

3.3.2 New state transitions

```
> step4 ((PushMarker x : instr), fptr, stack, vstack,
>        dump, heap, cstore, stats)
>  = (instr, fptr, [], vstack, (fptr, x, stack):dump, heap, cstore, stats)
```

The `Return` instruction has an extra case to cope with the empty stack, in which case an update should be performed:

```
> step4 ([Return], fptr, [], n:vstack, (f_upd, x, stack) : dump,
>         heap, cstore, stats)
>  = ([Return], fptr, stack, n:vstack, dump, heap', cstore, stats)
>    where
>    heap' = fUpdate heap f_upd x (intCode, FrameInt n)
```

The `UpdateMarkers` instruction updates closures with a partial application. The first rule deals with the case where enough arguments are on the stack, so no update need be performed:

```
> step4 ((UpdateMarkers n : instr), fptr, stack, vstack,
>        dump, heap, cstore, stats)
>  | n <=  length stack = (instr, fptr, stack, vstack, dump, heap, cstore, stats)
```

If this does not work, the second rule performs an update:

```
> step4 ((UpdateMarkers n : instr), fptr, stack, vstack,
>        (f_upd, x, stack') : dump, heap, cstore, stats)
>  = (UpdateMarkers n : instr, fptr, stack ++ stack', vstack,
>     dump, heap2, cstore, stats)
>    where
>    (heap1, pa_fptr) = fAlloc heap stack
>    heap2 = fUpdate heap1 f_upd x (pa_code, pa_fptr)
>    pa_code = (map (Push . Arg) (reverse [1..m])) ++
>                (UpdateMarkers n : instr)
>    m = length stack

> step4 state = error "Can't do this instruction yet"
```

3.3.3 Printing

We add code to print the new instructions.

```
> showInstruction4 d (UpdateMarkers n) = iStr "UpdateMarkers " `iAppend` iNum n
> showInstruction4 d (PushMarker n) = iStr "PushMarker " `iAppend` iNum n
> showInstruction4 d i = showInstruction5 d i

> showInstruction5 d i = error "Unknown instruction"
```

We print the dump showing only the size of the saved stack for brevity.

```
> showDump dump
>  = iConcat [iStr "Dump:   [",
>             iIndent (iInterleave (iStr ", ") (map showDumpItem dump)),
>             iStr "]", iNewline
>    ]
>    where
>    showDumpItem (fptr, slot, stack)
>     = iConcat [ iStr "(", showFramePtr fptr, iStr ",",
>                 iNum slot, iStr ",",
>                iStr "<stk size ", iNum (length stack), iStr ">)"
>       ]
```

3.4 Mark 5: Data structures

The changes needed to accomodate data structures are a little tiresome, because we need to add a new components to the machine state, for the data frame pointer and the machine output.

```
> type TimState = ([Instruction],    -- The current instruction stream
>                  FramePtr,         -- Address of current frame
>                  FramePtr,         -- Data frame pointer
>                  TimStack,         -- Stack of arguments
>                  TimValueStack,    -- Value stack
>                  TimDump,          -- Dump
>                  TimHeap,          -- Heap of frames
>                  CodeStore,        -- Labelled blocks of code
>                   [Int],           -- Output
>                  TimStats)         -- Statistics
```

This entails a new `timFinal` and `applyToStats` functions.

```
> timFinal ([], fptr, fdptr, stack, vstack,
>           dump, heap, cstore, output, stats) = True
> timFinal state                               = False
```

```
> applyToStats stats_fun (instr, fptr, fdptr, stack, vstack,
>                         dump, heap, cstore, output, stats)
>  = (instr, fptr, fdptr, stack, vstack,
>     dump, heap, cstore, output, stats_fun stats)
```

We need to extend the `instruction` type, to add the instructions `Switch` and `ReturnConstr`:

```
> data Instruction = Take Int Int
>                  | Push TimAMode
>                  | PushV ValueAMode
>                  | Enter TimAMode
>                  | Return
>                  | Op Op
>                  | Cond [Instruction] [Instruction]
>                  | Move Int TimAMode
>                  | MoveD Int Int
>                  | PushMarker Int
>                  | UpdateMarkers Int
>                  | Switch [(Int, [Instruction])]
>                  | ReturnConstr Int
>                  | Print
```

There is a new addressing mode `Data` to add to `timAMode`.

```
> data TimAMode = Arg Int
>               | Data Int
>               | Label [Char]
>               | Code [Instruction]
>               | IntConst Int
```

3.4.1 New compilation rules

The `compile` function is altered to initialise the new state components.

```
> compile program
>     = (  [Enter (Label "main")],   -- Initial instructions
>        FrameNull,                  -- Null frame pointer
>        FrameNull,                  -- Null data frame pointer
>        [(topCont, two_slot_frame)],-- Argument stack
>        initialValueStack,          -- Value stack
>        initialDump,                -- Dump
>        initial_heap,               -- Heap
>        compiled_code,              -- Compiled code for supercombinators
>        [],                         -- Output
>        statInitial)                -- Initial statistics
>        where
>          sc_defs          = preludeDefs ++ extraPreludeDefs ++ program
```

```
>          compiled_sc_defs = map (compileSC initial_env) sc_defs
>          compiled_code    = compiled_sc_defs ++ compiledPrimitives
>          (initial_heap, two_slot_frame)
>              = fAlloc hInitial [([],FrameNull), ([],FrameNull)]
>          initial_env = [(name, Label name) | (name, args, body) <- sc_defs]
>                        ++ [(name, Label name) | (name, code) <- compiledPrimitives]
```

`extraPreludeDefs` is defined below.

The changes to `compileR` amount to adding cases for `Cons` and `case`, and removing the special case for conditionals (which is now done via `case`):

```
> compileR (EConstr tag arity) env d
>  = (d, [UpdateMarkers arity, Take arity arity, ReturnConstr tag])
>
> compileR (ECase e alts) env d
>  = (d2, Push (Code [Switch branches]) : il)
>    where
>    compiled_alts = map (compileE d env) alts
>    d1 = maximum [d | (d,branch) <- compiled_alts]
>    branches = [branch | (d,branch) <- compiled_alts]
>    (d2, il) = compileR e env d1


> compileR (ELet isrec defs body) env d
>  = (d', move_instrs ++ il)
>    where
>    (dn, move_instrs) = mapAccuml
>                              make_move_instr (d+no_of_defs)
>                              (zip2 defs frame_slots)
>    (d', il) = compileR body new_env dn
>
>    new_env = zip2 names (map mkIndMode frame_slots) ++ env
>    no_of_defs = length defs
>    names = [name | (name,rhs) <- defs]
>    frame_slots = [d+1..d+no_of_defs]
>
>    make_move_instr d ((name, rhs), frame_slot)
>      = (d', Move frame_slot am)
>        where (d', am) = compileAL rhs frame_slot rhs_env d
>
>    rhs_env | not isrec = env
>            | isrec     = new_env
>
> compileR e env d | isBasicOp e = compileB e env (d, [Return])
>
> compileR (EAp e1 e2) env d
>  | isAtomicExpr e2 = (d1, Push (compileA e2 env) : il)
>    where
```

```
>    (d1, il) = compileR e1 env d
>
> compileR (EAp e1 e2) env d -- Non-atomic argument e2
>  = (d2, Move (d+1) am : Push (Code [Enter (Arg (d+1))]) : il)
>    where
>    (d1, am) = compileAL e2 (d+1) env (d+1)
>    (d2, il) = compileR e1 env d1
>
> compileR (EVar v)    env d = (d, [Enter (compileA (EVar v) env)])
>
> compileR (ENum n)    env d = (d, [PushV (IntVConst n), Return])
> compileR e           env d = error "compileR: can't compile this!"
```

`compileE` implements the $\mathcal{E}[\![]\!]$ compilation scheme. Notice that the variables bound by the `case` alternative are freely copyable, because they were originally passed as argument to `Cons`; hence we generate `Arg` addressing modes for them.

```
> compileE d env (tag, args, rhs)
>  = (d', (tag, map make_move_instr component_slots ++ rhs_instrs))
>    where
>    component_slots = [d+1..d+no_of_args]
>    make_move_instr slot = Move slot (Data (slot-d))
>    (d', rhs_instrs) = compileR rhs new_env (d+no_of_args)
>    new_env = zip2 args (map Arg component_slots) ++ env
>    no_of_args = length args
```

3.4.2 New state transitions

The `Switch` instruction just looks up the tag to find the appropriate branch.

```
> step ([Switch branches], fptr, fdptr, stack, (tag:vstack),
>        dump, heap, cstore, output, stats)
>  = (new_code, fptr, fdptr, stack, vstack,
>     dump, heap, cstore, output, stats)
>    where
>    new_code = aLookup branches tag (error ("Switch: bad tag "
>                                            ++ show tag))
```

The `ReturnConstr` instruction has two cases. The first is the usual case, where no update is to be performed:

```
> step ([ReturnConstr tag], fptr, fdptr, (instr',fptr'):stack, vstack,
>        dump, heap, cstore, output, stats)
>  = (instr', fptr', fptr, stack, tag:vstack,
>     dump, heap, cstore, output, stats)
```

The second deals with the update case:

```
> step ([ReturnConstr tag], fptr, fdptr, [], vstack,
>        (f_upd,upd_slot,stack'):dump, heap, cstore, output, stats)
>  = ([ReturnConstr tag], fptr, fdptr, stack', vstack,
>     dump, heap', cstore, output, stats)
>    where
>    heap' = fUpdate heap f_upd upd_slot ([ReturnConstr tag], fptr)
```

The `MoveD` instruction moves a closure from the data structure to the current frame:

```
> step ((MoveD n d : instr), fptr, fdptr, stack, vstack,
>       dump, heap, cstore,output, stats)
>  = (instr, fptr, fdptr, stack, vstack, dump, heap', cstore, output, stats)
>    where heap' = fUpdate heap fptr n (fGet heap fdptr d)
```

Finally, the `Print` instruction performs output:

```
> step (Print:instr, fptr, fdptr, stack, n:vstack,
>       dump, heap, cstore, output, stats)
>  = (instr, fptr, fdptr, stack, vstack,
>     dump, heap, cstore, output ++ [n], stats)
```

3.4.3 Old state transitions

Here are all the old state transitions, with the extra state components added.

```
> step ([Enter am], fptr, fdptr, stack, vstack,
>       dump, heap, cstore, output, stats)
>  = (instr', fptr', FrameNull, stack, vstack,
>     dump, heap, cstore, output, stats)
>    where (instr',fptr') = amToClosure am fptr fdptr heap cstore


> step ((Push am:instr), fptr, fdptr, stack, vstack,
>       dump, heap, cstore, output, stats)
>  = (instr, fptr, fdptr, amToClosure am fptr fdptr heap cstore : stack,
>     vstack, dump, heap, cstore, output, stats)


> step ((PushV FramePtr:instr), (FrameInt n), fdptr, stack, vstack,
>      dump, heap, cstore, output, stats)
>  = (instr, FrameInt n, fdptr, stack, n:vstack,
>     dump, heap, cstore, output, stats)


> step ((PushV (IntVConst n):instr), fptr, fdptr, stack, vstack,
>      dump, heap, cstore, output, stats)
>  = (instr, fptr, fdptr, stack, n:vstack, dump, heap, cstore, output, stats)
```

```
> step ((Op op:instr), fptr, fdptr, stack, vstack,
>       dump, heap, cstore, output, stats)
>  = (instr, fptr, fdptr, stack, performOp op vstack,
>     dump, heap, cstore, output, stats)

> step ((Take tot n : instr), fptr, fdptr, stack, vstack,
>       dump, heap, cstore,output, stats)
>  | length stack >= n = (instr, fptr', fdptr, drop n stack, vstack,
>     dump, heap', cstore, output, stats)
>  | otherwise = error "Too few args for Take instruction"
>    where (heap', fptr') = fAlloc heap frame_cts
>                           where
>                           frame_cts = take n stack ++
>                                       take (tot-n) (repeat dummy_closure)
>                           dummy_closure = ([], FrameNull)

> step ((Move n am : instr), fptr, fdptr, stack, vstack,
>       dump, heap, cstore,output, stats)
>  = (instr, fptr, fdptr, stack, vstack, dump, heap', cstore, output, stats)
>    where heap' = fUpdate heap fptr n (amToClosure am fptr fdptr heap cstore)

> step ((PushMarker x : instr), fptr, fdptr, stack, vstack,
>        dump, heap, cstore, output, stats)
>  = (instr, fptr, fdptr, [], vstack,
>     (fptr, x, stack):dump, heap, cstore, output, stats)

> step ([Return], fptr, fdptr, (instr',fptr'):stack, vstack,
>      dump, heap, cstore, output, stats)
>  = (instr', fptr', FrameNull, stack, vstack,
>     dump, heap, cstore, output, stats)

> step ([Return], fptr, fdptr, [], n:vstack,
>       (f_upd, x, stack) : dump, heap, cstore, output, stats)
>  = ([Return], fptr, fdptr, stack, n:vstack,
>     dump, heap', cstore, output, stats)
>    where
>    heap' = fUpdate heap f_upd x (intCode, FrameInt n)

> step ((UpdateMarkers n : instr), fptr, fdptr, stack, vstack,
>        dump, heap, cstore, output, stats)
>  | n <= length stack = (instr, fptr, fdptr, stack, vstack,
>     dump, heap, cstore, output, stats)

> step ((UpdateMarkers n : instr), fptr, fdptr, stack, vstack,
>        (f_upd, x, stack') : dump, heap, cstore, output, stats)
>  = (UpdateMarkers n : instr, fptr, fdptr, stack ++ stack', vstack,
```

```
>     dump, heap2, cstore, output, stats)
>    where
>    (heap1, pa_fptr) = fAlloc heap stack
>    heap2 = fUpdate heap1 f_upd x (pa_code, pa_fptr)
>    pa_code = (map (Push . Arg) (reverse [1..m])) ++
>                (UpdateMarkers n : instr)
>    m = length stack

> amToClosure (Arg n)      fptr fdptr heap cstore = fGet heap fptr n
> amToClosure (Data n)     fptr fdptr heap cstore = fGet heap fdptr n
> amToClosure (Code il)    fptr fdptr heap cstore = (il, fptr)
> amToClosure (IntConst n) fptr fdptr heap cstore = (intCode, FrameInt n)
> amToClosure (Label l)    fptr fdptr heap cstore
>                              = (codeLookup cstore l, fptr)
```

3.4.4 The printing mechanism

Here is the code for `topCont` and `headCont`, transcribed from Section 4.6.4.

```
> topCont = [Switch [(1, []),
>                    (2, [ Move 1 (Data 1), Move 2 (Data 2),
>                          Push (Code headCont),
>                          Enter (Arg 1)
>                   ])]]
>
> headCont = [Print, Push (Code topCont), Enter (Arg 2)]
```

3.4.5 The new prelude

We add to the standard prelude the definitions given in Section 4.6.3

```
> extraPreludeDefs
>  = [ ("cons",        [], EConstr 2 2),
>      ("nil",         [], EConstr 1 0),
>      ("true",        [], EConstr 2 0),
>      ("false",       [], EConstr 1 0),
>
>      ("if", ["c","t","f"], ECase (EVar "c") [(1, [], EVar "f"),
>                                              (2, [], EVar "t")])
>    ]
```

3.4.6 Printing the new instructions

We add code to print the new instructions.

```
> showInstruction5 d Print = iStr "Print"
> showInstruction5 d (ReturnConstr t) = iStr "ReturnConstr " `iAppend` iNum t
> showInstruction5 d (Switch branches)
>  = iConcat [
>     iStr "Switch {", iNewline, iStr "  ",
>     iIndent (iInterleave iNewline (map show_branch branches)),
>     iStr " }"
>   ]
>   where
>   show_branch (tag, il)
>     = iConcat [ iNum tag, iStr " -> ", showInstructions d il]
>
> showInstruction5 d i = error "Unknown instruction"

> showArg d (Arg m)   = (iStr "Arg ")   `iAppend` (iNum m)
> showArg d (Data m)  = (iStr "Data ")  `iAppend` (iNum m)
> showArg d (Code il) = (iStr "Code ")  `iAppend` (showInstructions d il)
> showArg d (Label s) = (iStr "Label ") `iAppend` (iStr s)
> showArg d (IntConst n) = (iStr "IntConst ") `iAppend` (iNum n)

\indexDTT{showArg}%
\indexDTT{showArg}%
\indexDTT{showArg}%
\indexDTT{showArg}%
\indexDTT{showArg}%

> showStats (instr, fptr, fdptr, stack, vstack,
>            dump, heap, code, output, stats)
>  = iConcat [showOutput output,
>             iStr "Steps taken = ", iNum (statGetSteps stats), iNewline,
>           iStr "No of frames allocated = ", iNum (length (hAddresses heap)),
>             iNewline, iNewline]

> showState (instr, fptr, fdptr, stack, vstack,
>            dump, heap, cstore, output, stats)
>  = iConcat [
>      iStr "Code:  ", showInstructions Terse instr, iNewline,
>      showFrame heap fptr,
>      showStack stack,
>      showValueStack vstack,
>      showDump dump,
>      showOutput output,
>      iNewline
>    ]

> showSCDefns (instr, fptr, fdptr, stack, vstack,
>              dump, heap, cstore, output, stats)
>  = iInterleave iNewline (map showSC cstore)
```

```
> showOutput output = iConcat [
>                      iStr "Output: [",
>                      iInterleave (iStr ",") (map iNum output),
>                      iStr "]", iNewline
>                     ]
```

The new `showResults` function prints output incrementally:

```
> showResults states
>  = iDisplay (showR 0 states)
>    where
>    showR n [state] = iConcat [iNewline, showStats state]
>    showR n (state:states)
>     | length output == n = (iStr ".") `iAppend` (showR n states)
>     | otherwise = iConcat [ iNewline, iNum (last output),
>                             showR (n+1) states ]
>      where
>      (instr, fptr, fdptr, stack, vstack,
>       dump, heap, cstore, output, stats) = state
```

Chapter 4

A Parallel G-Machine

4.1 Mark 1

```
> runProg :: [Char] -> [Char]
> runProg = showResults . eval . compile . parse

> pushglobal :: Name -> GmState -> GmState
> pushglobal f state
>  = putStack (a: getStack state) state
>    where a = aLookup (getGlobals state) f (error ("Undeclared global " ++ f))

> pushint :: Int -> GmState -> GmState
> pushint n state
>  = putHeap heap' (putStack (a: getStack state) state)
>    where (heap', a) = hAlloc (getHeap state) (NNum n)

> mkap :: GmState -> GmState
> mkap state
>  = putHeap heap' (putStack (a:as') state)
>    where (heap', a)  = hAlloc (getHeap state) (NAp a1 a2)
>          (a1:a2:as') = getStack state

> getArg :: Node -> Addr
> getArg (NAp a1 a2) = a2

> slide :: Int -> GmState -> GmState
> slide n state
>  = putStack (a: drop n as) state
>    where (a:as) = getStack state

> data Node = NNum Int          -- Numbers
>           | NAp  Addr Addr    -- Applications
```

```
>          | NGlobal Int GmCode -- Globals
>          | NInd Addr          -- Indirections
>          | NConstr Int [Addr] -- Constructors

> type GmCompiledSC = (Name, Int, GmCode)

> allocateSc :: GmHeap -> GmCompiledSC -> (GmHeap, (Name, Addr))
> allocateSc heap (name, nargs, instns)
>  = (heap', (name, addr))
>    where (heap', addr) = hAlloc heap (NGlobal nargs instns)

> makeTask a = ([Eval], [a], [], [], 0)

> initialCode = [Eval, Print]

> compileSc :: (Name, [Name], CoreExpr) -> GmCompiledSC
> compileSc (name, env, body)
>  = (name, length env, compileR body (zip env [0..]))

> type GmCompiler = CoreExpr -> GmEnvironment -> GmCode

> type GmEnvironment = ASSOC Name Int

> argOffset :: Int -> GmEnvironment -> GmEnvironment
> argOffset n env = [(v, n+m) | (v,m) <- env]

> rearrange :: Int -> GmHeap -> GmStack -> GmStack
> rearrange n heap as
>  = take n as' ++ drop n as
>    where as' = map (getArg . hLookup heap) (tl as)

> allocNodes :: Int -> GmHeap -> (GmHeap, [Addr])
> allocNodes 0     heap = (heap,  [])
> allocNodes (n+1) heap = (heap2, a:as)
>                         where (heap1, as) = allocNodes n heap
>                               (heap2, a)  = hAlloc heap1 (NInd hNull)

> compileLet :: GmCompiler -> [(Name, CoreExpr)] -> GmCompiler
> compileLet comp defs expr env
>  = compileLet' defs env ++ comp expr env' ++ [Slide (length defs)]
>    where env' = compileArgs defs env

> compileLet' :: [(Name, CoreExpr)] -> GmEnvironment -> GmCode
> compileLet' []                  env = []
> compileLet' ((name, expr):defs) env
>  = compileC expr env ++ compileLet' defs (argOffset 1 env)
```

```
> compileArgs :: [(Name, CoreExpr)] -> GmEnvironment -> GmEnvironment
> compileArgs defs env
>  = zip (map first defs) [n-1, n-2 .. 0] ++ argOffset n env
>    where n = length defs

> showResults states
>  = iDisplay (iConcat [
>        iStr "Supercombinator definitions", iNewline,
>        iInterleave iNewline (map (showSC s) (pgmGetGlobals s)),
>        iNewline, iNewline, iStr "State transitions", iNewline, iNewline,
>        iLayn (map showState states),
>        iNewline, iNewline,
>        showStats (last states)])
>        where (s:ss) = states

> showSC s (name, addr)
>  = iConcat [ iStr "Code for ", iStr name, iNewline,
>              showInstructions code, iNewline, iNewline]
>    where (NGlobal arity code) = (hLookup (pgmGetHeap s) addr)

> showStats s
>  = iLayn (map showStat (pgmGetStats s))

> showStat :: Int -> Iseq
> showStat n = iConcat [ iStr "Steps taken = ", iNum n, iNewline]

> showState s
>  = iConcat [showOutput (pgmGetOutput s),         iNewline,
>             showSparks (pgmGetSparks s),         iNewline,
>             iLayn [showProcessor (g,p) | p <- l]]
>    where (g,l) = s

> showProcessor :: GmState -> Iseq
> showProcessor s
>  = iConcat [iStr "<",  shortShowInstructions 2 (getCode s),
>             iStr ", ", shortShowStack (getStack s),
>             iStr ", ", showDump s,
>             iStr ", ", showVStack s,
>             iStr ">"]

> showInstructions :: GmCode -> Iseq
> showInstructions is
>  = iConcat [iStr "  Code:{",
>             iIndent (iInterleave iNewline (map showInstruction is)),
>             iStr "}", iNewline]
```

```
> showStack :: GmState -> Iseq
> showStack s
>  = iConcat [iStr " Stack:[",
>             iIndent (iInterleave iNewline
>                         (map (showStackItem s) (reverse (getStack s)))),
>             iStr "]"]

> showStackItem :: GmState -> Addr -> Iseq
> showStackItem s a
>  = iConcat [iStr (showaddr a), iStr ": ",
>             showNode s a (hLookup (getHeap s) a)]

> showDump :: GmState -> Iseq
> showDump s
>  = iConcat [iStr "  Dump:[",
>             iIndent (iInterleave iNewline
>                         (map showDumpItem (reverse (getDump s)))),
>             iStr "]"]

> showDumpItem :: GmDumpItem -> Iseq
> showDumpItem (code, stack)
>  = iConcat [iStr "<",
>             shortShowInstructions 2 code, iStr ", ",
>             shortShowStack stack,         iStr ">"]

> shortShowInstructions :: Int -> GmCode -> Iseq
> shortShowInstructions number code
>  = iConcat [iStr "{", iInterleave (iStr "; ") dotcodes, iStr "}"]
>    where codes = map showInstruction (take number code)
>          dotcodes | length code > number = codes ++ [iStr "..."]
>                   | otherwise = codes

> shortShowStack :: GmStack -> Iseq
> shortShowStack stack
>  = iConcat [iStr "[",
>             iInterleave (iStr ", ") (map (iStr . showaddr) stack),
>             iStr "]"]

> showVStack :: GmState -> Iseq
> showVStack s
>  = iConcat [iStr "Vstack:[",
>             iInterleave (iStr ", ") (map iNum (getVStack s)),
>             iStr "]"]

> showOutput out = iConcat [iStr "Output:\"", iStr out, iStr "\""]
```

```
> showSparks s
>  = iConcat [iStr "Sparks:[",
>             iInterleave (iStr ", ") (map (iStr . showaddr) s),
>             iStr "]"]

> showNode :: GmState -> Addr -> Node -> Iseq
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>                              where v = hd [n | (n,b) <- globals, a==b]
>                                    globals = getGlobals s
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ",  iStr (showaddr a1),
>                                       iStr " ",    iStr (showaddr a2)]
> showNode s a (NInd a1)     = iConcat [iStr "Ind ", iStr (showaddr a1)]
> showNode s a (NConstr t as)
>  = iConcat [iStr "Cons ", iNum t, iStr " [",
>             iInterleave (iStr ", ") (map (iStr.showaddr) as), iStr "]"]

> compileAlts :: (Int -> GmCompiler) -- compiler for alternative bodies
>                -> [CoreAlt]        -- the list of alternatives
>                -> GmEnvironment    -- the current environment
>                -> [(Int, GmCode)]  -- list of alternative code sequences
> compileAlts comp alts env
>  = [(tag, comp (length names) body (zip names [0..] ++ argOffset (length names) env))
>           | (tag, names, body) <- alts]

> compileE' :: Int -> GmCompiler
> compileE' offset expr env
>  = [Split offset] ++ compileE expr env ++ [Slide offset]

> boxInteger :: Int -> GmState -> GmState
> boxInteger n state
>  = putStack (a: getStack state) (putHeap h' state)
>    where (h', a) = hAlloc (getHeap state) (NNum n)

> unboxInteger :: Addr -> GmState -> Int
> unboxInteger a state
>  = ub (hLookup (getHeap state) a)
>    where ub (NNum i) = i
>          ub n        = error "Unboxing a non-integer"

> builtInDyadic :: ASSOC Name Instruction
> builtInDyadic
>  = [("+", Add), ("-", Sub), ("*", Mul), ("div", Div),
>     ("==", Eq), ("~=", Ne), (">=", Ge),
>     (">",  Gt), ("<=", Le), ("<",  Lt)]
```

```
> boxBoolean :: Bool -> GmState -> GmState
> boxBoolean b state
>  = putStack (a: getStack state) (putHeap h' state)
>    where (h',a) = hAlloc (getHeap state) (NConstr b' [])
>          b' | b = 2
>             | otherwise = 1


> buildInitialHeap :: CoreProgram -> (GmHeap, GmGlobals)
> buildInitialHeap program
>  = mapAccuml allocateSc hInitial compiled
>    where compiled = map compileSc (preludeDefs ++ program ++ primitives)


> primitives :: [(Name,[Name],CoreExpr)]
> primitives
>  = [("+", ["x","y"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "y"))),
>     ("-", ["x","y"], (EAp (EAp (EVar "-") (EVar "x")) (EVar "y"))),
>     ("*", ["x","y"], (EAp (EAp (EVar "*") (EVar "x")) (EVar "y"))),
>     ("/", ["x","y"], (EAp (EAp (EVar "/") (EVar "x")) (EVar "y"))),

>     ("negate", ["x"], (EAp (EVar "negate") (EVar "x"))),

>     ("==", ["x","y"], (EAp (EAp (EVar "==") (EVar "x")) (EVar "y"))),
>     ("~=", ["x","y"], (EAp (EAp (EVar "~=") (EVar "x")) (EVar "y"))),
>     (">=", ["x","y"], (EAp (EAp (EVar ">=") (EVar "x")) (EVar "y"))),
>     (">",  ["x","y"], (EAp (EAp (EVar ">")  (EVar "x")) (EVar "y"))),
>     ("<=", ["x","y"], (EAp (EAp (EVar "<=") (EVar "x")) (EVar "y"))),
>     ("<",  ["x","y"], (EAp (EAp (EVar "<")  (EVar "x")) (EVar "y"))),

>     ("if",  ["c","t","f"],
>        (EAp (EAp (EAp (EVar "if") (EVar "c")) (EVar "t")) (EVar "f"))),
>     ("True",  [], (EConstr 2 0)),
>     ("False", [], (EConstr 1 0)),

>     ("par", ["x","y"], (EAp (EAp (EVar "par") (EVar "x")) (EVar "y")))]


> update :: Int -> GmState -> GmState
> update n state
>  = putHeap heap' (putStack as state)
>    where heap'  = hUpdate (getHeap state) (as!!n) (NInd a)
>          (a:as) = getStack state


> pop :: Int -> GmState -> GmState
> pop n state
>  = putStack (drop n (getStack state)) state
```

```
> compileLetrec comp defs e args
>  = [Alloc n]           ++
>    compiled defs (n-1) ++
>    comp e newArgs      ++
>    [Slide n]
>    where newArgs = compileArgs defs args
>          n       = length defs
>          compiled []     i = []
>          compiled (d:ds) i = compileC (second d) newArgs ++
>                              [Update i]                  ++
>                              compiled ds (i-1)

> alloc :: Int -> GmState -> GmState
> alloc n state
>  = putHeap heap' (putStack (as'++getStack state) state)
>    where (heap', as') = allocNodes n (getHeap state)

> push :: Int -> GmState -> GmState
> push n state
>  = putStack ((as!!n): as) state
>    where as = getStack state

> showAlternatives nis
>  = iConcat [iStr "[",
>             iInterleave (iStr ", ") (map showLabelInstructions nis),
>             iStr "]"]
>    where showLabelInstructions (tag, code)
>           = iConcat [iNum tag, iStr ": ", shortShowInstructions 2 code]

> casejump alts state
>  = putCode (i ++ getCode state) state
>    where (NConstr t as) = hLookup (getHeap state) (hd (getStack state))
>          i = aLookup alts t (error ("No case for constructor" ++ show t))

> split :: Int -> GmState -> GmState
> split j state
>  = putStack (as++s) state
>    where (NConstr t as) = hLookup (getHeap state) a
>          (a:s)          = getStack state

> gmprint :: GmState -> GmState
> gmprint state
>  = newState (hLookup (getHeap state) a) state
>    where
>    newState (NConstr t as) = putCode (printcode (length as) ++ getCode state) .
>                              putStack (as++s)
```

```
>    newState (NNum n)       = putOutput (show n) . putStack s
>    newState n              = error "Print of non data structure"
>    (a:s) = getStack state

> printcode 0     = []
> printcode (n+1) = Eval: Print: printcode n

> pack :: Int -> Int -> GmState -> GmState
> pack t a state
>  = putHeap heap' (putStack (addr: drop a s) state)
>    where s = getStack state
>          (heap', addr) = hAlloc (getHeap state) (NConstr t (take a s))

> compileC :: GmCompiler
> compileC (EConstr t 0) args = [Pack t 0]
> compileC (EVar v)      args | v `elem` aDomain args  = [Push n]
>                             | otherwise = [Pushglobal v]
>                               where n = aLookup args v (error "")
> compileC (ENum n)      args = [Pushint n]
> compileC (ELet recursive defs e)
>                        args | recursive = compileLetrec compileC defs e args
>                             | otherwise = compileLet    compileC defs e args
> compileC (EAp e1 e2)   args
>  | saturatedCons spine = compileCS (reverse spine) args
>  | otherwise = compileC e2 args ++ compileC e1 (argOffset 1 args) ++ [Mkap]
>   where spine = makeSpine (EAp e1 e2)
>         saturatedCons (EConstr t a:es) = a == length es
>         saturatedCons (e:es)           = False

> makeSpine (EAp e1 e2) = makeSpine e1 ++ [e2]
> makeSpine e           = [e]

> compileCS [EConstr t a] args = [Pack t a]
> compileCS (e:es)        args = compileC  e args ++
>                                compileCS es (argOffset 1 args)

> getOutput ((o, heap, globals, sparks, stats), locals) = o
> putOutput o' ((o, heap, globals, sparks, stats), locals)
>  = ((o', heap, globals, sparks, stats), locals)

> getCode (globals, (i, stack, dump, vstack, clock)) = i
> putCode i' (globals, (i, stack, dump, vstack, clock))
>  = (globals, (i', stack, dump, vstack, clock))

> getStack (globals, (i, stack, dump, vstack, clock)) = stack
> putStack stack' (globals, (i, stack, dump, vstack, clock))
>  = (globals, (i, stack', dump, vstack, clock))
```

```
> getVStack (globals, (i, stack, dump, vstack, clock)) = vstack
> putVStack vstack' (globals, (i, stack, dump, vstack, clock))
>  = (globals, (i, stack, dump, vstack', clock))

> getClock (globals, (i, stack, dump, vstack, clock)) = clock
> putClock clock' (globals, (i, stack, dump, vstack, clock))
>  = (globals, (i, stack, dump, vstack, clock'))

> getDump (globals, (i, stack, dump, vstack, clock)) = dump
> putDump dump' (globals, (i, stack, dump, vstack, clock))
>  = (globals, (i, stack, dump', vstack, clock))

> getHeap ((o, heap, globals, sparks, stats), locals) = heap
> putHeap heap' ((o, heap, globals, sparks, stats), locals)
>  = ((o, heap', globals, sparks, stats), locals)

> getGlobals ((o, heap, globals, sparks, stats), locals) = globals

> getSparks ((o, heap, globals, sparks, stats), locals) = sparks
> putSparks sparks' ((o, heap, globals, sparks, stats), locals)
>  = ((o, heap, globals, sparks', stats), locals)

> getStats ((o, heap, globals, sparks, stats), locals) = stats
> putStats stats' ((o, heap, globals, sparks, stats), locals)
>  = ((o, heap, globals, sparks, stats'), locals)

> primitive1 :: (Int -> Int)             -- operator
>               -> (GmState -> GmState) -- state transition
>
> primitive1 op state
>  = putVStack (op n: ns) state
>    where (n:ns) = getVStack state

> primitive2 :: (Int -> Int -> Int)      -- operator
>               -> (GmState -> GmState) -- state transition
>
> primitive2 op state
>  = putVStack (op n0 n1: ns) state
>    where (n0:n1:ns) = getVStack state

> arithmetic1 = primitive1

> arithmetic2 = primitive2
```

```
> comparison op = primitive2 op'
>                 where op' x y = if op x y then 2 else 1

> showInstruction (Slide n)      = iAppend (iStr "Slide ")       (iNum n)
> showInstruction (Alloc n)      = iAppend (iStr "Alloc ")       (iNum n)
> showInstruction (Update n)     = iAppend (iStr "Update ")      (iNum n)
> showInstruction (Pop n)        = iAppend (iStr "Pop ")         (iNum n)
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = iAppend (iStr "Pushglobal ")  (iStr f)
> showInstruction (Pushint n)    = iAppend (iStr "Pushint ")     (iNum n)
> showInstruction (Push n)       = iAppend (iStr "Push ")        (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
> showInstruction Eval           = iStr  "Eval"
> showInstruction Add            = iStr  "Add"
> showInstruction Sub            = iStr  "Sub"
> showInstruction Mul            = iStr  "Mul"
> showInstruction Div            = iStr  "Div"
> showInstruction Neg            = iStr  "Neg"
> showInstruction Eq             = iStr  "Eq"
> showInstruction Ne             = iStr  "Ne"
> showInstruction Le             = iStr  "Le"
> showInstruction Lt             = iStr  "Lt"
> showInstruction Ge             = iStr  "Ge"
> showInstruction Gt             = iStr  "Gt"
> showInstruction (Pack t a)     = iAppend (iAppend (iStr "Pack ")   (iNum t)) (iNum a)
> showInstruction (Casejump nis) = iAppend (iStr "Casejump ")
>                                  (showAlternatives nis)
> showInstruction (Split n)      = iAppend (iStr "Split ")       (iNum n)
> showInstruction Print          = iStr "Print"
> showInstruction Mkbool         = iStr "Mkbool"
> showInstruction Mkint          = iStr "Mkint"
> showInstruction Get            = iStr "Get"
> showInstruction (Pushbasic n)  = iAppend (iStr "Pushbasic")  (iNum n)
> showInstruction Return         = iStr "Return"
> showInstruction (Cond t f)     = iAppend (iStr "Cond ")
>                                  (showAlternatives [(2,t),(1,f)])
> showInstruction Par            = iStr "Par"

> data Instruction = Slide Int            |
>                  Alloc Int              |
>                  Update Int             |
>                  Pop Int                |
>                  Unwind                 |
>                  Pushglobal Name        |
>                  Pushint Int            |
>                  Push Int               |
>                  Mkap                   |
>                  Eval                   |
```

```
>                 Add                     |
>                 Sub                     |
>                 Mul                     |
>                 Div                     |
>                 Neg                     |
>                 Eq                      |
>                 Ne                      |
>                 Le                      |
>                 Lt                      |
>                 Ge                      |
>                 Gt                      |
>                 Negate                  |
>                 Pack Int Int            |
>                 Casejump [(Int,GmCode)] |
>                 Split Int               |
>                 Print                   |
>                 Pushbasic Int           |
>                 Mkbool                  |
>                 Mkint                   |
>                 Get                     |
>                 Return                  |
>                 Cond GmCode GmCode      |
>                 Par
>         deriving (Eq,Text)

> compileR :: GmCompiler
> compileR (EAp (EAp (EVar "par") e1) e2) args
>  = compileC e2 args ++ [Push 0, Par] ++
>    compileC e1 (argOffset 1 args) ++ [Mkap, Update n, Pop n, Unwind]
>    where n = length args
> compileR (EAp (EAp (EVar op) e1) e2) args
>  | op `elem` binaryOps = compileE (EAp (EAp (EVar op) e1) e2) args ++
>         [Return]
>    where binaryOps = map first builtInDyadic
>          n = length args
> compileR (EAp (EVar "negate") e) args
>  = compileE (EAp (EVar "negate") e) args ++ [Return]
>    where n = length args
> compileR (EAp (EAp (EAp (EVar "if") e1) e2) e3) args
>  = compileE e1 args ++ [Cond (compileR e2 args) (compileR e3 args)]
> compileR (ECase e as) args = compileE e args
> compileR (ELet recursive defs e)
>                        env | recursive = compileLetrec compileR defs e env
>                            | otherwise = compileLet    compileR defs e env
> compileR (ECase e as)  env = compileE e env ++
>                              [Casejump (compileAlts compileR' as env)]
> compileR e args = compileC e args ++ [Update n, Pop n, Unwind]
>                   where n = length args
```

```
> compileR' :: Int -> GmCompiler
> compileR' n expr env = [Split n] ++ compileR expr env

> compileE :: GmCompiler
> compileE (EAp (EAp (EVar "par") e1) e2) args
>  = compileC e2 args ++ [Push 0, Par] ++
>    compileC e1 (argOffset 1 args) ++ [Mkap, Eval]
> compileE (EAp (EAp (EVar op) e1) e2) args
>  | op `elem` ["+", "-", "*", "/"]
>   = compileB (EAp (EAp (EVar op) e1) e2) args ++ [Mkint]
>  | op `elem` ["==", "~+", "<", "<=", ">", ">="]
>   = compileB (EAp (EAp (EVar op) e1) e2) args ++ [Mkbool]
> compileE (EAp (EVar "negate") e) args
>  = compileB (EAp (EVar "negate") e) args ++ [Mkint]
> compileE (EAp (EAp (EAp (EVar "if") e1) e2) e3) args
>  = compileB e1 args ++ [Cond (compileE e2 args) (compileE e3 args)]
> compileE (ENum n)      args = [Pushint n]
> compileE (ELet recursive defs e)
>                        args | recursive = compileLetrec compileE defs e args
>                             | otherwise = compileLet    compileE defs e args
> compileE (ECase e as)  args = compileE e args ++
>                               [Casejump (compileAlts compileE' as args)]
> compileE e             args = compileC e args ++ [Eval]

> strictOperators :: ASSOC Name (Instruction, Int)
> strictOperators
>  = [("+", (Add, 2)), ("-", (Sub, 2)), ("*", (Mul, 2)), ("/", (Div, 2)),
>     ("negate", (Neg, 1)),
>     ("==", (Eq, 2)), ("~=", (Ne, 2)), (">=", (Ge, 2)),
>     (">",  (Gt, 2)), ("<=", (Le, 2)), ("<",  (Lt, 2))]

> compileB :: GmCompiler
> compileB (EAp (EAp (EAp (EVar "if") e1) e2) e3) env
>  = compileB e1 env ++ [Cond (compileB e2 env) (compileB e3 env)]
> compileB (ENum n) env = [Pushbasic n]
> compileB (EAp (EAp (EVar "+")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Add]
> compileB (EAp (EAp (EVar "-")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Sub]
> compileB (EAp (EAp (EVar "*")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Mul]
> compileB (EAp (EAp (EVar "/")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Div]
> compileB (EAp (EAp (EVar "==")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Eq]
> compileB (EAp (EAp (EVar "~=")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Ne]
```

```
> compileB (EAp (EAp (EVar ">")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Gt]
> compileB (EAp (EAp (EVar ">=")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Ge]
> compileB (EAp (EAp (EVar "<")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Lt]
> compileB (EAp (EAp (EVar "<=")  e1) e2) env
>  = compileB e2 env ++ compileB e1 env ++ [Le]
> compileB (EAp (EVar "negate")  e) env
>  = compileB e env ++ [Neg]
> compileB (ELet recursive defs e)
>                      env | recursive = compileLetrec compileB defs e env
>                          | otherwise = compileLet    compileB defs e env
> compileB e            env = compileE e env ++ [Get]

> dispatch :: Instruction -> GmState -> GmState
> dispatch Unwind         = unwind
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Push n)       = push n
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
> dispatch (Update n)     = update n
> dispatch (Pop n)        = pop n
> dispatch (Alloc n)      = alloc n
> dispatch Add            = arithmetic2 (+)
> dispatch Sub            = arithmetic2 (-)
> dispatch Mul            = arithmetic2 (*)
> dispatch Div            = arithmetic2 (div)
> dispatch Neg            = arithmetic1 (negate)
> dispatch Eq             = comparison (==)
> dispatch Ne             = comparison (/=)
> dispatch Lt             = comparison (<)
> dispatch Le             = comparison (<=)
> dispatch Gt             = comparison (>)
> dispatch Ge             = comparison (>=)
> dispatch Eval           = evalop
> dispatch (Cond i1 i2)   = cond i1 i2
> dispatch (Casejump alts)= casejump alts
> dispatch (Split n)      = split n
> dispatch Print          = gmprint
> dispatch Mkbool         = mkbool
> dispatch Mkint          = mkint
> dispatch Get            = get
> dispatch (Pushbasic n)  = pushbasic n
> dispatch Return         = gmreturn
> dispatch Par            = par

> gmreturn :: GmState -> GmState
```

```
> gmreturn state
>  | d == [] = putCode [] state
>  | otherwise = putDump d' (putStack (a:s') (putCode i' state))
>    where (a:s)        = getStack state
>          ((i',s'):d') = d
>          d            = getDump state

> mkbool :: GmState -> GmState
> mkbool state
>  = putStack (a:getStack state) (putVStack v (putHeap heap' state))
>    where (heap',a) = hAlloc (getHeap state) (NConstr x [])
>          (x:v)     = getVStack state

> mkint :: GmState -> GmState
> mkint state
>  = putStack (a:getStack state) (putVStack v (putHeap heap' state))
>    where (heap',a) = hAlloc (getHeap state) (NNum x)
>          (x:v)     = getVStack state

> get :: GmState -> GmState
> get state
>  = newState (hLookup (getHeap state) a) (putStack s state)
>    where newState (NConstr n []) = putVStack (n:v)
>          newState (NNum n)       = putVStack (n:v)
>          newState (NInd a')      = newState (hLookup (getHeap state) a')
>          newState n              = error "Get of a non-number or bool"
>          v = getVStack state
>          (a:s) = getStack state

> cond :: GmCode -> GmCode -> GmState -> GmState
> cond t f state
>  = putCode i (putVStack v state)
>    where (x:v) = getVStack state
>          i = if x==2 then f else t ++ getCode state

> pushbasic :: Int -> GmState -> GmState
> pushbasic n state
>  = putVStack (n:getVStack state) state

> evalop :: GmState -> GmState
> evalop state
>  = putCode [Unwind] (putStack [a] (putDump d' state))
>    where (a:s) = getStack state
>          d'    = (getCode state, s): getDump state

> unwind :: GmState -> GmState
```

```
> unwind state
>  = newState (hLookup heap a)
>    where
>    (a:as)       = getStack state
>    heap         = getHeap state
>    d            = getDump state
>    ((i',s'):d') = d
>    newState (NNum n)
>     = if d == [] then putCode [] state
>       else putCode i' (putStack (a:s') (putDump d' state))
>    newState (NAp a1 a2)
>     = putCode [Unwind] (putStack (a1:a:as) state)
>    newState (NGlobal n c)
>     | length as >= n = putCode c  (putStack rs state)
>     | d == [] = putCode [] state
>     | otherwise = putCode i' (putStack (last (a:as):s') (putDump d' state))
>       where rs = rearrange n heap (a:as)
>    newState (NInd a1)
>     = putCode [Unwind] (putStack (a1:as) state)
>    newState (NConstr t as)
>     | d == [] = putCode [] state
>     | otherwise = putCode i' (putStack (a:s') (putDump d' state))
```

4.2 Mark 2

```
> showNode :: GmState -> Addr -> Node -> Iseq
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>                              where v = hd [n | (n,b) <- globals, a==b]
>                                    globals = getGlobals s
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ",  iStr (showaddr a1),
>                                       iStr " ",    iStr (showaddr a2)]
> showNode s a (NInd a1)     = iConcat [iStr "Ind ", iStr (showaddr a1)]
> showNode s a (NConstr t as)
>  = iConcat [iStr "Cons ", iNum t, iStr " [",
>             iInterleave (iStr ", ") (map (iStr.showaddr) as), iStr "]"]
> showNode s a (NLGlobal n g) = iConcat [iStr "*Global ", iStr v]
>                               where v = hd [n | (n,b) <- globals, a==b]
>                                     globals = getGlobals s
> showNode s a (NLAp a1 a2)   = iConcat [iStr "*Ap ",  iStr (showaddr a1),
>                                        iStr " ",     iStr (showaddr a2)]

> unwind :: GmState -> GmState
> unwind state
>  = newState (hLookup heap a)
>    where
```

```
>    (a:as)       = getStack state
>    heap         = getHeap state
>    d            = getDump state
>    ((i',s'):d') = d
>    newState (NNum n)
>     | d == [] = putCode [] state
>     | otherwise = putCode i' (putStack (a:s') (putDump d' state))
>    newState (NAp a1 a2)
>     = putCode [Unwind] (putStack (a1:a:as) (lock a state))
>    newState (NGlobal n c)
>     | length as >= n = putCode c  (putStack rs state')
>     | d == [] = putCode [] state'
>     | otherwise = putCode i' (putStack (last (a:as):s') (putDump d' state'))
>       where rs = rearrange n heap (a:as)
>             state' = if n==0 then lock a state else state
>    newState (NInd a1)
>     = putCode [Unwind] (putStack (a1:as) state)
>    newState (NConstr t as)
>     | d == [] = putCode [] state
>     | otherwise = putCode i' (putStack (a:s') (putDump d' state))
>    newState (NLGlobal n c)
>     = putCode [Unwind] state
>    newState (NLAp a1 a2)
>     = putCode [Unwind] state

> update :: Int -> GmState -> GmState
> update n state
>  = putHeap heap' (putStack as state)
>    where heap'  = hUpdate (getHeap state') root (NInd a)
>          (a:as) = getStack state
>          state' = unlock a state
>          root   = as!!n
```

4.2.1 Mark 4

```
> showNode :: GmState -> Addr -> Node -> Iseq
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>                              where v = hd [n | (n,b) <- globals, a==b]
>                                    globals = getGlobals s
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ",  iStr (showaddr a1),
>                                       iStr " ",    iStr (showaddr a2)]
> showNode s a (NInd a1)     = iConcat [iStr "Ind ", iStr (showaddr a1)]
> showNode s a (NConstr t as)
>  = iConcat [iStr "Cons ", iNum t, iStr " [",
>             iInterleave (iStr ", ") (map (iStr.showaddr) as), iStr "]"]
> showNode s a (NLGlobal n g pl) = iConcat [iStr "*Global ", iStr v]
```

```
>                                  where v = hd [n | (n,b) <- globals, a==b]
>                                        globals = getGlobals s
> showNode s a (NLAp a1 a2 pl) = iConcat [iStr "*Ap ",  iStr (showaddr a1),
>                                         iStr " ",     iStr (showaddr a2)]

> lock addr state
>  = putHeap (newHeap (hLookup heap addr)) state
>    where
>    heap = getHeap state
>    newHeap (NAp a1 a2)   = hUpdate heap addr (NLAp a1 a2 [])
>    newHeap (NGlobal n c) = if n== 0 then hUpdate heap addr (NLGlobal n c [])
>                            else heap

> unlock addr state
>  = newState (hLookup heap addr) state
>    where
>    heap = getHeap state
>    newState (NLAp a1 a2 pl)
>      = (unlock a1) .
>        (putHeap (hUpdate heap addr (NAp a1 a2))) .
>        (emptyPendingList pl)
>    newState (NLGlobal n c pl)
>      = (putHeap (hUpdate heap addr (NGlobal n c))) .
>        (emptyPendingList pl)
>    newState n = id

> steps state
>  = scheduler global' local'
>    where ((out, heap, globals, sparks, stats), local) = state
>          newtasks = sparks
>          global'  = (out, heap, globals, [], stats)
>          local'   = local ++ newtasks

> showSparks s
>  = iConcat [iStr "Tasks:", iNum (length s)]

> scheduler (out, heap, globals, sparks, stats) tasks
>  = mapAccuml step global' running
>    where running    = map tick (take machineSize tasks)
>          nonRunning = drop machineSize tasks
>          global'    = (out, heap, globals, nonRunning, stats)

> par state
>  = putSparks (makeTask a:sparks) (putStack s state)
>    where (a:s)  = getStack  state
>          sparks = getSparks state
```

```
> unwind :: GmState -> GmState
> unwind state
>  = newState (hLookup heap a)
>    where
>    (a:as)       = getStack state
>    heap         = getHeap state
>    d            = getDump state
>    ((i',s'):d') = d
>    newState (NNum n)
>     | d == [] = putCode [] state
>     | otherwise = putCode i' (putStack (a:s') (putDump d' state))
>    newState (NAp a1 a2)
>     = putCode [Unwind] (putStack (a1:a:as) (lock a state))
>    newState (NGlobal n c)
>     | length as >= n = putCode c  (putStack rs state')
>     | d == [] = putCode [] state'
>     | otherwise = putCode i' (putStack (last (a:as):s') (putDump d' state'))
>      where rs = rearrange n heap (a:as)
>            state' = if n==0 then lock a state else state
>    newState (NInd a1)
>     = putCode [Unwind] (putStack (a1:as) state)
>    newState (NConstr t as)
>     | d == [] = putCode [] state
>     | otherwise = putCode i' (putStack (a:s') (putDump d' state))
>    newState (NLGlobal n c pl)
>     = putHeap (hUpdate heap a (NLGlobal n c (local:pl))) (global, emptyTask)
>       where (global,local) = putCode [Unwind] state
>    newState (NLAp a1 a2 pl)
>     = putHeap (hUpdate heap a (NLAp a1 a2 (local:pl))) (global, emptyTask)
>       where (global,local) = putCode [Unwind] state

> doAdmin ((out, heap, globals, sparks, stats), local)
>  = ((out, heap, globals, sparks, stats'), local')
>    where (local', stats') = foldr filter ([], stats) local
>          filter (i, stack, dump, vstack, clock) (local, stats)
>           = if i == [] then if clock == 0 then (local, stats)
>                             else (local, clock:stats)
>             else ((i, stack, dump, vstack, clock): local, stats)
```

Chapter 5

Lambda lifting

5.1 Introducion

5.2 Improving the expr data type

5.3 Mark 1: A Simple Lambda Lifter

5.4 Mark 2: Improving the simple lambda lifter

```
> mkELet is_rec [] e    = e
> mkELet is_rec defns e = ELet is_rec defns e
```

Here is the missing code for case expressions.

```
> freeVars_case lv e alts
>  = (setUnion (freeVarsOf e') free, ACase e' alts')
>    where
>    e'     = freeVars_e lv e
>    alts'  = [ (tag, args, freeVars_e (setUnion lv (setFromList args)) e)
>             | (tag, args, e) <- alts]
>    free   = setUnionList (map freeVarsOf_alt alts')


> abstract_case free e alts
>  = ECase (abstract_e e) [(tag, args, abstract_e e) | (tag, args, e) <- alts]


> rename_case env ns e alts
>  = (ns2, ECase e' alts')
>    where
>    (ns1, e') = rename_e env ns e
>    (ns2, alts') = mapAccuml rename_alt ns alts
>    rename_alt ns (tag, args, rhs)
```

```
>     = (ns2, (tag, args', rhs'))
>       where
>       (ns1, args', env') = newNames ns args
>       (ns2, rhs')  = rename_e (env' ++ env) ns1 rhs
```

5.5 Mark 3: Johnsson style lambda-lifting

5.6 Mark 4: A separate full laziness pass

5.7 Mark 5: Improvements to fully laziness

```
> freeToLevel_case level env free e alts
>  = (freeSetToLevel env free, ACase e' alts')
>    where
>    e' = freeToLevel_e level env e
>    alts' = map freeToLevel_alt alts
>    freeToLevel_alt (tag, args, rhs)
>     = (tag, args', freeToLevel_e (level+1) env' rhs)
>       where env'  = args' ++ env
>             args' = [(arg, level+1) | arg <- args]
```

Here is the code for `identifyMFEs_e1`.

```
> identifyMFEs_case1 level body alts
>  = ECase (identifyMFEs_e level body) (map identifyMFEs_alt alts)
>    where
>    identifyMFEs_alt (tag, [], e) = (tag, [], identifyMFEs_e level e)
>    identifyMFEs_alt (tag, args, e)
>     = (tag, args, identifyMFEs_e arg_level e)
>       where
>       (name, arg_level) = hd args
```

The code for `renameGen_e` is very straightforward:

```
> renameGen_e new_b env ns (EVar v)      = (ns, EVar (aLookup env v v))
```

Constructors, numbers and applications are easy.

```
> renameGen_e new_b env ns (EConstr t a) = (ns, EConstr t a)
> renameGen_e new_b env ns (ENum n)      = (ns, ENum n)
> renameGen_e new_b env ns (EAp e1 e2)
>  = (ns2, EAp e1' e2')
>    where
>    (ns1, e1') = renameGen_e new_b env ns e1
>    (ns2, e2') = renameGen_e new_b env ns1 e2
```

When we meet an `ELam` we need to invent new names for the arguments, using `newNames`, and augment the enironment with the mapping returned by `newNames`.

```
> renameGen_e new_b env ns (ELam args body)
>  = (ns1, ELam args' body')
>    where
>    (ns1, args', env') = new_b ns args
>    (ns2, body') = renameGen_e new_b (env' ++ env) ns1 body
```

`let(rec)` expressions work similarly:

```
> renameGen_e new_b env ns (ELet is_rec defns body)
>  = (ns3, ELet is_rec (zip2 binders' values') body')
>    where
>    (ns1, body') = renameGen_e new_b body_env ns body
>    binders = bindersOf defns
>    (ns2, binders', env') = new_b ns1 binders
>    body_env = env' ++ env
>    (ns3, values') = mapAccuml (renameGen_e new_b rhsEnv) ns2 (rhssOf defns)
>    rhsEnv | is_rec    = body_env
>           | otherwise = env
```

```
> renameGen_e new_b env ns (ECase e alts)
>  = (ns2, ECase e' alts')
>    where
>    (ns1, e') = renameGen_e new_b env ns e
>    (ns2, alts') = mapAccuml rename_alt ns alts
>    rename_alt ns (tag, args, rhs)
>     = (ns2, (tag, args', rhs'))
>       where
>       (ns1, args', env') = new_b ns args
>       (ns2, rhs')  = renameGen_e new_b (env' ++ env) ns1 rhs
```

Here is the `float_e` case for `case` expressions. They are done in a similar way to lambda abstractions, except that if an alternative binds no arguments then no definitions are deposited.

```
> float_case e alts
>  = (fd2, ECase e' alts')
>    where
>    (fd1, e') = float_e e
>    (fd2, alts') = mapAccuml float_alt fd1 alts
>
>    float_alt fd (tag, [], rhs) = (fd ++ fd_rhs, (tag, [], rhs'))
>                                  where
>                                  (fd_rhs, rhs') = float_e rhs
>    float_alt fd (tag, args, rhs)
>     = (fd ++ fd_outer, (tag, args', install fd_this_level rhs'))
```

```
>       where
>       (fd_rhs, rhs') = float_e rhs
>       args' = [arg | (arg,level) <- args]
>       (first_arg,this_level) = hd args
>       (fd_outer, fd_this_level) = partitionFloats this_level fd_rhs
```