Exam

Advanced Programming

Casper B. Hansen
University of Copenhagen
Department of Computer Science
`fvx507@alumni.ku.dk`

November 6, 2014

Abstract

In the exam I give a description of the implementation, design decision thereof, reflections on unresolved issues, as well as an assessment of the provided solution.

The topics covered are a parser for the fictional Fast language, an interpreter of the same language, and lastly an asyncronous backend for a spreadsheet.

Tasks

1 Parser

1.1 Preliminary

Before I started implementing any actual part of the parser, I defined some convenience functions that would aid the implementation of all the following parts of the parser. These include parsing whitespace, integers, strings, character- and string tokens, parens, brackets, and building upon these; things inside parens or brackets, which is a very modular and efficient use of the library.

1.2 Solution

In designing the parser, I drew upon the fact that the Haskell parser library used (ReadP) allows for building smaller parsers. I began the process by implementing a subset of the language; namely the expression grammar (excluding *Expr '.' Name '(' Args ')'*, which is an omission I have made because of lack of time). Each form an expression could take on was defined as a parser in its own right.

Once I was able to parse a selection of expressions, I decided to move on, as the expression parser was stable and adding new expression types was a breeze. I then moved on to implementing a parser for the *Params* and *Exprs* grammar production rules. These were fairly easy, as a result of having the convenience functions handy — all I had to do was to get the contents of the parens, which the `sepBy` parser was an obvious choice for.

With these done, I had the means to implement the remainder of the expression parsers, as some of them depended upon the formerly described functionality, and this concludes the expression parser.

Being able to parse expressions I could now focus on implementing the class declaration parser. Because I couldn't immediately see how to modularize the methods I decided to simply write a parser for each kind, so as to not spend too much time on figuring it out. Some erroneous parts occurs in this parser (see reflections and assessment).

Having all of the sub-parsers in place, I could add the main parser, which simply parses for a series of class declarations followed by an end-of-file.

1.3 Reflections

The arithmetic sub-parser has a flaw in it, that when evaluating just an expression like *2+3* we do not get the expected result `Plus ( IntConst 2, IntConst 3)`, the actual parsed result is `IntConst 2`. This would be a problem if we allowed such statements in our language, but this is *not* the case, as the language requires a semi-colon after each lone expression, and thus isn't a problem once used in the context of another parser requiring some termination symbol. This is also true arguments as each expression is then separated by a comma, and ended by a paren. I thought about enforcing it, to make it work without, but decided that it would make no difference and poses no problem as such expressions can never occur in Fast.

I realize that the implementation of parsing the different kinds of methods (constructor, receiver and class methods) can probably be modularized better, and this does pose some behavior expectancy anormaly — thus I cannot reason that they all work equally well. Furthermore, there is a problem with parsing the receiver, which is supposed to be at the very end of the class methods. This is a problem I realize does *not* comply with the grammar specification. The circumvention of the problem I had to do, because of time constraints was to rewrite the class declaration rule to *'*`class`*' Name '' ConstructorDecl RecvDecl NamedMethodDecls ''* — this is a definite error in the parser.

1.4 Assessment

A simple thing like the convience functions ensures a stable and well-defined expected behavior across any part of the parser making use of them. Also, having defined each form of the left-hand side of the grammar as a parser of its own allows me to reason that the precedence of the parser is correct.

The solution lacks the ability to parse the synonomous expression for sending messages (*Expr '.' Name '(' Args ')'*). Had I had more time I would probably look to parse it by way of using a chain parser, much like I did with the arithmetic operation parser. Furthermore, it does not live up to the grammar specification, because of the problem I had with the receiver method.

2 Interpreter

2.1 Preliminary

In order to start implementation of anything in the interpreter, I had to define the basic types (see appendix 8 and 9) used manipulated throughout the implementation. I reason that a simple `Map` synonym would do for the `Store` as they are basically synonomous in nature.

I reason that a global state should contain a store of references to objects (line 52). A store of methods (line 53) bound by class name and its own name for lookup at runtime, since these aren't bound by instantiations, and can be reused across many instances. An output (line 54), which is merely a string and a unique ID counter (line 55), for allocating of unique IDs.

2.2 Solution

With the types defined I then had the means to define the `FastM a` monad, which I defined as given a program `Prog` and a state `GlobalState` produces either an error or a state and a computed value `(GlobalState, a)`. The binding operator is defined as given the inputs of the monad type (a program and a state) compute the result, and apply the monadic function `f`, returning the transformed monad. The return operation merely wraps the given argument, and the fail operation is as simply defined as it can be, using `error`. This monad is responsible for handling the runtime of the program being executed.

Having the `FastM` monad, I proceeded to define the monadic manipulation functions (see appendix 11), which I believe are all very self-explanatory from their naming.

Defining the `FastMethodM a` as taking an object reference `ObjectReference`, and a method state `MethodState`, manipulates the runtime (in other words, producing a `FastM a`). The binding operator is defined as given the inputs of the monad type (a reference and a method state) compute the result, and apply the monadic function `f`, returning the transformed result. The return and fail operations are both feed to the lifting function (lines 199–200), which lifts the `FastMethodM` to perform the operation as if it were a `FastM`.

As the case with `FastM`, the definitions and workings of the monadic manipulation functions (see appendix 13) of `FastMethodM` are all very self-explanatory from their naming.

An interesting monadic function is the `evalMethodBody` which given an object reference, a list of arguments and a method body (a list of `Expr`) alters the runtime state. Firstly, it binds the given arguments, and it then proceeds to execute the method body. Also, the `evalArgs` is quite handy, as it allows for easy argument processing. It simply maps over the arguments using the monadic function `evalExpr`, producing a list of corresponding values.

The last, I believe, correctly implemented function is `findClassDecl`, which given a name looks up the corresponding class declaration. If it does not yield any results or if more than one such name exists, a run-time error is produced. Otherwise we can register the contained methods of the class and return the class definition for instantiation (see subsection Reflections on this matter).

2.3 Reflections

Although the interpreter is incomplete, and does not yield any meaningful output for testing, I believe I have a firm understanding of how to complete it, had I had more time to do so. The `FastM` monad handles the runtime and with it the global state, whilst `FastMethodM` handles the execution of function calls and the local scope thereof.

As I was out of time, and did not get to instantiate any objects, the `createObject` is incomplete, and does not do what it is supposed to. My thoughts of implementation thereof is that once allocated, the object should be pushed onto the state, allowing it to be referenced. After that, since the return value of the constructor is simply discarded, it would be a mere matter of running `evalMethodBody` on the constructor.

Most of the monadic expression evaluation functions aren't implemented because I was stalled at the method implementation discussed above. I would guess that it wouldn't take much time to implement them, once objects could be instantiated, including the `sendMessageTo` method, which also lacks an implementation.

2.4 Assessment

I find the solution to be quite good, in spite of the incompleteness. The code that was implemented does work, but cannot be tested against an actual program. It's too bad that I was out of time, as I found the overall design of the interpreter to be very well-thought out and I was so close to connecting the last pieces of the implementation. I do recognize however, that since it does not produce any observable results other than a debugging message in the output, where execution stalls because of lacking implementation, it is not in working order as an interpreter — yet, but it's close.

3 Spreadsheet

3.1 Solution

I began by defining the API (see appendix 18), notable design decisions were made for the `cell` function. The `cell` function requests a new cell from the spreadsheet, which handles the request and responds with a cell process ID (this may be an existing cell, and this is handled by the spreadsheet server, see appendix 21). A sidenote to this is that although `set_value` should be asyncronous and I call its handler using the syncronous `rpc` function, it does return from the handler immediately after an updater has received its job (see appendix 22).

The update server is implemented as a thread that runs in parallel with an associated cell. An important design decision was made to not keep an updater alive if the associated cell merely contains a value. This was done by the case in which the updater does not receive any dependencies (see appendix 23). If it does indeed have dependencies, the updater-thread will stay alive, and look for changes in its dependencies, and accumulate values until a result is acheived. Once a result is computed it will reset itself. If it is in the process of producing a result and there are missing data after a period of 500ms it will *ping* the cells for a value, and notify viewers of the associated cell that it is trying to update.

I circumvented the case in which a cell process id doesn't exist yet by simply creating it, should it be requested by the updater. New cells are given the value `undefined`, which I reason is appropriate in such cases — that is, if a cell doesn't exist it must evaluate as undefined for formulae depending on it.

3.2 Reflections

The provided solution does not handle circular dependencies, and that is a clear error in the solution. This is, however, only because I had to divert my attention to other parts of the exam. Had I had more time I would have either used the extra information field to accumulate which cells have are dependent on the current evaluation, and if the cell being evaluated at that time is a member of that set then a circular reference would have been detected, causing an error.

3.3 Assessment

Because of the design decision to only spawn update-threads when needed, and kill them when not need any longer, I find my solution to be very efficient in its used of threads — no thread alive is obsolete. A simpler solution would have an update thread for each cell kept alive, even though it has nothing but a singular value to return. My solution only keeps an update-thread alive if its associated cell is a formula — requiring it to stay informed of changes.

As a result of this design decision long-running formulae cannot block, any cell — not even itself. It allows formulae to be in the process of evaluation, but be stopped by issuing a `set_value` message to the cell, and this will simply discard the current update-thread, producing a new thread that will be associated with the cell in question, thus avoiding deadlocks as well.

A Code

A.1 Parser

```
module FastParser
       ( Error
       , parseString
       , parseFile
       )
       where

import FastAST
import Data.Char
import Text.ParserCombinators.ReadP

keywords :: [String]
keywords = ["class","match","new","receive","return","self","send","set"]

-- | You may change this type to whatever you want - just make sure it
-- is an instance of 'Show'.
type Error = String

-- CONVENIENCE
ws = munch (\c -> c `elem` " \t\n")
ws1 = munch1 (\c -> c `elem` " \t\n")

isKeyword :: String -> Bool
isKeyword s = s `elem` keywords

leftParen = chrToken '('
rightParen = chrToken ')'

parens :: ReadP p -> ReadP p
parens p = between leftParen rightParen p

leftBrace = chrToken '{'
rightBrace = chrToken '}'

braces :: ReadP p -> ReadP p
braces p = between leftBrace rightBrace p

-- parses char tokens
chrToken :: Char -> ReadP ()
chrToken c = do skipSpaces; char c; skipSpaces
```

Figure 1: Parser solution, part 1 of n (../src/fast/FastParser.hs)

```haskell
-- parses string tokens
strToken :: String -> ReadP ()
strToken s = do skipSpaces; string s; skipSpaces

-- parses digits
digits :: ReadP String
digits = munch1 isDigit

-- parses the sign
sign :: ReadP Integer
sign = (do
        chrToken '-'
        return $ -1)
    +++ return 1

-- parses a name
parseName :: ReadP Name
parseName = do
    f <- satisfy isLetter -- first must be a letter
    r <- munch (\c -> isAlphaNum c || c `elem` "_") -- rest
    return $ (f:r)

-- EXPRESSIONS
parseInt :: ReadP Integer
parseInt = do
    s <- sign
    d <- digits
    return $ (s * (read d))

parseStr :: ReadP String
parseStr = do
    char '"'
    s <- munch (\c -> c /= '"')
    char '"'
    return $ s

parseParams :: ReadP [Name]
parseParams = parens (sepBy parseName (chrToken ','))

parseArgs :: ReadP Exprs
parseArgs = parens (sepBy parseExpr (chrToken ','))
```

Figure 2: Parser solution, part 2 of n (../src/fast/FastParser.hs)

```haskell
parseExprs :: ReadP Exprs
parseExprs =
    do {
        leftBrace;
        es <- (sepBy parseExpr (chrToken ';'));
        chrToken ';';
        rightBrace;
        return es
    } +++
    do {
        between leftBrace rightBrace ws;
        return []
    }

parseExpr :: ReadP Expr
parseExpr = parseTerm -- TODO: allow nesting

parseTerm :: ReadP Expr
parseTerm =
    parens parseExpr +++
    parseNew +++
    parseSend +++
    parseMatch +++
    arithmetic +++
    parseSetVar +++
    parseSetField
    where arithmetic = chainl1 e0 op0
          e0 = chainl1 e1 op0
          e1 = chainl1 e2 op1
          e2 = chainl1 e3 op2
          e3 = chainl1 parseLiteral op3
          op0 = do { chrToken '+'; return Plus }
          op1 = do { chrToken '-'; return Minus }
          op2 = do { chrToken '*'; return Times }
          op3 = do { chrToken '/'; return DividedBy }
```

Figure 3: Parser solution, part 3 of n (../src/fast/FastParser.hs)

```
parseLiteral :: ReadP Expr
parseLiteral =
    do { strToken "self"; return $ Self } +++
    do { (n, es) <- parseCall; return $ TermLiteral n es } +++
    do { n <- parseName; return $ ReadVar n } +++
    do { s <- parseStr; return $ StringConst s } +++
    do { n <- parseInt; return $ IntConst n }

parseMatch :: ReadP Expr
parseMatch = do
    strToken "match"
    e <- parseExpr
    leftBrace
    cs <- many1 parseCase
    rightBrace
    return $ Match e cs

parseSend :: ReadP Expr
parseSend = do
    strToken "send"
    leftParen
    r <- parseExpr
    chrToken ','
    m <- parseExpr
    rightParen
    return $ SendMessage r m

parseNew :: ReadP Expr
parseNew = do
    strToken "new"
    (n, es) <- parseCall
    return $ New n es

parseSetVar :: ReadP Expr
parseSetVar = do
    strToken "set"
    n <- parseName
    chrToken '='
    e <- parseExpr
    return $ SetVar n e
```

Figure 4: Parser solution, part 4 of n (../src/fast/FastParser.hs)

```
parseSetField :: ReadP Expr
parseSetField = do
    strToken "set"
    strToken "self."
    n <- parseName
    chrToken '='
    e <- parseExpr
    return $ SetField n e

parseCall :: ReadP (Name, [Expr])
parseCall = do
    n <- parseName
    ws;
    args <- parseArgs
    ws;
    return (n, args)

parseReturn :: ReadP Expr
parseReturn = do
    strToken "return"
    e <- parseExpr
    return $ Return e

-- PATTERNS
parsePattern :: ReadP Pattern
parsePattern =
    do { n <- parseName; return $ AnyValue n } +++
    do { n <- parseName; ps <- parseParams; return $ TermPattern n ps } +++
    do { s <- parseStr; return $ ConstString s } +++
    do { n <- parseInt; return $ ConstInt n }

parseCase :: ReadP Case
parseCase = do
    p <- parsePattern
    strToken "->"
    es <- parseExprs
    return $ (p, es)

-- CLASSES
parseConstructor :: ReadP (Maybe ConstructorDecl)
parseConstructor = do
    string "new"
    ps <- between (char '(') (char ')') (sepBy parseName (char ','))
    ws1
    es <- between (char '{') (char '}') (sepBy parseExpr (char ';'))
    ws
    return $ Just ( MethodDecl { methodParameters=ps, methodBody=es} )
```

Figure 5: Parser solution, part 5 of n (../src/fast/FastParser.hs)

```haskell
parseMethod :: ReadP NamedMethodDecl
parseMethod = do
    n <- parseName
    ws1
    ps <- parseParams
    ws
    es <- parseExprs
    ws
    return $ NamedMethodDecl n (method ps es)
    where method a b = MethodDecl { methodParameters=a, methodBody=b}

parseReceive :: ReadP (Maybe ReceiveDecl)
parseReceive = do
    string "receive"
    ws1
    p <- parseName
    ws
    es <- between (char '{') (char '}') (sepBy parseExpr (char ';'))
    ws
    return $ Just (ReceiveDecl { receiveParam=p, receiveBody=es })

parseClass :: ReadP ClassDecl
parseClass = do
    string "class"
    ws1
    n <- parseName
    ws
    char '{'
    ws
    c <- option Nothing parseConstructor
    ws
    r <- option Nothing parseReceive -- TODO: fix order!
    ws
    m <- sepBy parseMethod ws
    ws
    char '}'
    ws
    return $ ClassDecl { className=n
                       , classConstructor=Nothing--c
                       , classMethods=m
                       , classReceive=Nothing--r
                       }
```

Figure 6: Parser solution, part 6 of n (../src/fast/FastParser.hs)

```
-- PROGRAM
parseProg :: ReadP Prog
parseProg = do
    decls <- many parseClass
    eof
    return decls

parseString :: String -> Either Error Prog
parseString str = case opt of
    [] -> Left "Parser error."
    (x:_) -> Right (fst x)
    where opt = readP_to_S parseProg str

parseFile :: FilePath -> IO (Either Error Prog)
parseFile filename = fmap parseString $ readFile filename
```

Figure 7: Parser solution, part 7 of n (../src/fast/FastParser.hs)

A.2 Interpreter

```haskell
module FastInterpreter
       ( runProg
       , Error (..)
       )
       where

import FastAST

import Control.Applicative
import Control.Monad
import Data.List
import Data.Maybe

import Data.Map (Map)
import qualified Data.Map as Map

-- | Give the printed representation of a value.
printed :: Value -> String
printed (IntValue x) = show x
printed (StringValue s) = s
printed (ReferenceValue ref) = "#<object " ++ show ref ++ ">"
printed (TermValue (Term sym vs)) =
  sym ++ "(" ++ intercalate ", " (map printed vs) ++ ")"

-- ^ Any runtime error.  You may add more constructors to this type
-- (or remove the existing ones) if you want.  Just make sure it is
-- still an instance of 'Show' and 'Eq'.
data Error = Error String
             deriving (Show, Eq)

type Output = String

-- | A key-value store where the keys are of type @k@, and the values
-- are of type @v@.  Used for mapping object references to objects and
-- variable names to values.
type Store k v = Map k v
```

Figure 8: Interpreter solution, types (../src/fast/FastInterpreter.hs)

```haskell
-- | A mapping from object references to objects.
type GlobalStore = Store ObjectReference ObjectState

-- | A mapping from field names to field values.
type ObjectFields = Store Name Value

-- | A mapping from variable names to variable values.
type MethodVariables = Store Name Value

-- | A mapping from class and method names to methods.
type MethodStore = Store (Name, Name) MethodDecl

-- | The global state of the program execution.
data GlobalState = GlobalState {
    refs :: GlobalStore,
    methods :: MethodStore,
    output :: Output,
    uuid :: ObjectReference
}

init_state :: GlobalState
init_state = GlobalState { refs=Map.empty, methods=Map.empty, output="", uuid=0 }

-- | The state of a single object.
data ObjectState = ObjectState {
    ref :: ObjectReference,
    name :: Name, -- class name
    fields :: ObjectFields
}

init_obj :: ObjectReference -> Name -> ObjectState
init_obj r n = ObjectState { ref=r, name=n, fields=Map.empty }

-- | The state of a method execution.
data MethodState = MethodState {
    vars :: MethodVariables,
    body :: [Expr]
}

init_method :: MethodState
init_method = MethodState { vars=Map.empty, body=[] }
```

Figure 9: Interpreter solution, types (../src/fast/FastInterpreter.hs)

```
-- | The basic monad in which execution of a Fast program takes place.
-- Maintains the global state, the running output, and whether or not
-- an error has occurred.

data FastM a = FastM {
    runFastM :: Prog -> GlobalState -> Either Error (GlobalState, a)
}

instance Functor FastM where
  fmap = liftM

instance Applicative FastM where
  pure = return
  (<*>) = ap

instance Monad FastM where

    -- (>>=) :: FastM a -> (a -> FastM b) -> FastM b
    -- a :: Prog -> GlobalState -> Either Error (GlobalState, a)
    (FastM a) >>= f = do
        FastM $ \p s -> case (a p s) of
                        Right (s', v) -> do (FastM b) <- return $ f v
                                            b p s'
                        Left e -> Left e

    -- return a :: FastM a
    return a = FastM (\_ s -> Right (s,a) )

    fail e = error e
```

Figure 10: Interpreter solution, the FastM Monad (../src/fast/FastInterpreter.hs)

```haskell
getGlobalState :: FastM GlobalState
getGlobalState = FastM (\_ s -> Right (s,s) )

putGlobalState :: GlobalState -> FastM ()
putGlobalState s = FastM (\_ s -> Right (s,()) )

modifyGlobalState :: (GlobalState -> GlobalState) -> FastM ()
modifyGlobalState f = do
    state <- getGlobalState
    FastM (\_ _ -> Right (f state, ()) )

modifyGlobalStore :: (GlobalStore -> GlobalStore) -> FastM ()
modifyGlobalStore f = modifyGlobalState (\s -> s { refs=f (refs s) } )

lookupMethod :: Name -> Name -> FastM MethodDecl
lookupMethod c m = do
    state <- getGlobalState
    case (Map.lookup (c, m) (methods state)) of
        Just method -> FastM (\_ s -> Right(s, method) )
        Nothing -> fail "No such method"

lookupObject :: ObjectReference -> FastM ObjectState
lookupObject ref = do
    state <- getGlobalState
    case (Map.lookup ref (refs state)) of
        Just obj -> FastM (\_ s -> Right (s, obj) )
        Nothing -> fail "No such reference"

setObject :: ObjectReference -> ObjectState -> FastM ()
setObject ref obj =
    modifyGlobalState (\s -> s { refs=Map.insert ref obj (refs s) } )

-- | Add the ’printed’ representation of the value to the output.
printValue :: Value -> FastM ()
printValue v = modifyGlobalState (\s -> s { output=out } )
    where out = printed v

-- | Debug messages
debug :: String -> FastM ()
debug s = printValue (StringValue $ "[DEBUG] " ++ s ++ "\n")

-- | Get the program being executed.
askProg :: FastM Prog
askProg = FastM (\p s -> Right (s,p) )

-- | Get a unique, fresh, never-before used object reference for use
-- to identify a new object.
allocUniqID :: FastM ObjectReference
allocUniqID = do
    state <- getGlobalState
    modifyGlobalState (\s -> s { uuid=(uuid state)+1 } )
    return (uuid state)
```

Figure 11: Interpreter solution, FastM monadic manipulation functions (../src/fast/FastInterpreter.hs)

```
-- | The monad in which methods (and constructors and receive actions)
-- execute.  Runs on top of 'FastM' - maintains the reference to self,
-- as well as the method variables.
--
-- Note that since FastMethodM runs on top of FastM, a FastMethodM
-- action has access to the global state (through liftFastM).
data FastMethodM a = FastMethodM {
    runFastMethodM :: ObjectReference -> MethodState -> FastM a
}

instance Functor FastMethodM where
  fmap = liftM

instance Applicative FastMethodM where
  pure = return
  (<*>) = ap

instance Monad FastMethodM where

    -- (>>=) :: FastMethodM a -> (a -> FastMethodM b) -> FastMethodM b
    (FastMethodM a) >>= f = do
        prog <- liftFastM $ askProg
        state <- liftFastM $ getGlobalState
        FastMethodM $ \r s -> do
            case (a r s) of
            -- v :: Prog -> GlobalState -> Either Error GlobalState
                FastM v -> do case (v prog state) of
                                   Right (x, v) -> do (FastMethodM b) <- return $ f v
                                                      b r s

    -- return a :: FastMethodM a
    return = liftFastM . return
    fail = liftFastM . fail

-- | Perform a 'FastM' operation inside a 'FastMethodM'.
liftFastM :: FastM a -> FastMethodM a
liftFastM = (\op -> FastMethodM (\_ _ -> op))
```

Figure 12: Interpreter solution, the FastMethodM Monad (../src/fast/FastInterpreter.hs)

```
-- | Who are we?
askSelf :: FastMethodM ObjectReference
askSelf = FastMethodM (\r _ -> return r)

-- | Add the given name-value associations to the variable store.
bindVars :: [(Name, Value)] -> FastMethodM a -> FastMethodM a
bindVars [] s = s
bindVars (v:vs) (FastMethodM g) = do
    FastMethodM $ \r s -> g r (bind v s)
    where bind (n, v) s = s { vars=Map.insert n v (vars s) }

getMethodState :: FastMethodM MethodState
getMethodState = FastMethodM (\_ s -> return s)

putMethodState :: MethodState -> FastMethodM ()
putMethodState s = FastMethodM (\_ s -> return ())

getsMethodState :: (MethodState -> a) -> FastMethodM a
getsMethodState f = do s <- getMethodState
                       return $ f s

modifyMethodState :: (MethodState -> MethodState) -> FastMethodM ()
modifyMethodState f = do s <- getMethodState
                         putMethodState $ f s

getObjectState :: FastMethodM ObjectState
getObjectState = do
    state <- liftFastM $ getGlobalState
    FastMethodM (\r _ -> check $ Map.lookup r (refs state) )
    where check v = case v of
                         Just v' -> return v'
                         Nothing -> fail "No such object"

putObjectState :: ObjectState -> FastMethodM ()
putObjectState obj = do
    state <- liftFastM $ getGlobalState
    liftFastM $ modifyGlobalStore (\s -> Map.insert (ref obj) obj s)

getsObjectState :: (ObjectState -> a) -> FastMethodM a
getsObjectState f = do s <- getObjectState
                       return $ f s

modifyObjectState :: (ObjectState -> ObjectState) -> FastMethodM ()
modifyObjectState f = do s <- getObjectState
                         putObjectState $ f s
```

Figure 13: Interpreter solution, FastMethodM monadic manipulation functions (../src/fast/-FastInterpreter.hs)

```
-- | Evaluate a method body - the passed arguments are the object in
-- which to run, the initial variable bindings (probably the
-- parameters of the method, constructor or receive action), and the
-- body.  Returns a value and the new state of the object.
evalMethodBody :: ObjectReference
               -> [(Name, Value)]
               -> Exprs
               -> FastM (Value, ObjectState)
evalMethodBody ref vars exprs
    = let (FastMethodM f) = bindVars vars $ eval exprs
      in do
        debug ("Evaluating method")
        f ref init_method
    where eval exps = do
                        val <- evalExprs exps
                        state <- getObjectState
                        return $ (val, state)

evalExprs :: [Expr] -> FastMethodM Value
evalExprs [] = return $ TermValue $ Term "nil" []
evalExprs [e] = evalExpr e
evalExprs (e:es) = evalExpr e >> evalExprs es

evalArgs :: [Expr] -> FastMethodM [Value]
evalArgs exprs = mapM evalExpr exprs
```

Figure 14: Interpreter solution, monadic evaluation functions (../src/fast/FastInterpreter.hs)

```haskell
evalExpr :: Expr -> FastMethodM Value
evalExpr (IntConst v)           = return $ IntValue v
evalExpr (StringConst v)        = return $ StringValue v

-- arithmetic
evalExpr (Plus (IntConst lhs) (IntConst rhs)) =
    return $ (IntValue (lhs + rhs))
evalExpr (Minus lhs rhs)        = undefined
evalExpr (Times lhs rhs)        = undefined
evalExpr (DividedBy lhs rhs)    = undefined


evalExpr (TermLiteral n exps)   = undefined
evalExpr (CallMethod e n exps)  = do
    s <- getObjectState
    method <- liftFastM $ lookupMethod (name s) n
    as <- evalArgs exps
    rec <- evalExpr e -- recipient (class)
    (msg, state) <- liftFastM $ evalMethodBody (ref s) (args method as) (body method)
    liftFastM $ sendMessageTo rec msg
    where args m vs = zip (params m) vs
          params m = methodParameters m
          body m = methodBody m

evalExpr (SendMessage obj e)    = do
    rec <- evalExpr obj
    msg <- evalExpr e
    liftFastM $ sendMessageTo rec msg

evalExpr (New name exprs)       = undefined --createObject name exprs
```

Figure 15: Interpreter solution, monadic expression functions (../src/fast/FastInterpreter.hs)

```haskell
-- | Find the declaration of the class with the given name, or cause
-- an error if that name is not a class.
findClassDecl :: Name -> FastM ClassDecl
findClassDecl n = do
    prog <- askProg
    case (filter condition prog) of
        [] -> fail $ "No class definition of class " ++ n
        [c] -> (do mapM (register n) (classMethods c); return c)
        _ -> fail $ "Duplicate definitions of class " ++ n
    where condition ClassDecl { className=name } = name == n
          register c m = let (NamedMethodDecl n mtd) = m
                         in modifyGlobalState (\s -> s { methods=Map.insert (c, n) mtd (methods s)} )

-- | Instantiate the class with the given name, passing the given
-- values to the constructor.
createObject :: Name -> [Value] -> FastM ObjectReference
createObject name args = do
    id <- allocUniqID

    debug ("Creating class " ++ name)

    -- FIX: add the object, putObjectState (init_obj id name)

    ClassDecl { classConstructor=constructor
              , classMethods=methods
              , classReceive=receive } <- findClassDecl name

    case constructor of
              Just MethodDecl { methodParameters=p
                              , methodBody=b } -> run id (zip p args) b
              Nothing -> run id [] []
    return id

    where run id args body = return id --evalMethodBody id args body
          reg id dec = let (NamedMethodDecl name method) = dec
                       in modifyObjectState (\s -> s { fields=Map.insert name (v method) (fields s) } )
                       where v m = StringValue ""
```

Figure 16: Interpreter solution, find and instantiate class monadic functions (../src/fast/-FastInterpreter.hs)

```haskell
sendMessageTo :: Value -> Value -> FastM Value
sendMessageTo = undefined

runProg :: Prog -> Either Error String
runProg p = let (FastM f) = createObject "Main" []
            in case fmap fst $ f p init_state of
                    Right state -> Right $ output state
                    Left e -> Left e
```

Figure 17: Interpreter solution, run function (../src/fast/FastInterpreter.hs)

A.3 Spreadsheet

```erlang
-module(sheet).

%% API
-export([ sheet/0
        , cell/2
        , add_viewer/2
        , remove_viewer/2
        , get_viewers/1
        , set_value/2
        ]).

%%%===================================================================
%%% API
%%%===================================================================

sheet() ->
    P = spawn(fun() -> sheet_server(dict:new()) end),
    io:format("Started sheet process ~p~n", [P]),
    {ok, P}.

cell(S, A) -> rpc(S, {cell, A}).
add_viewer(C, P) -> async(C, {self(), {add_viewer, P}}).
remove_viewer(C, P) -> async(C, {self(), {remove_viewer, P}}).
get_viewers(C) -> rpc(C, get_viewers).
set_value(C, V) -> rpc(C, {set, V}), ok.
```

Figure 18: Spreadsheet solution, API (../src/sheet/sheet.erl)

```
%%%===================================================================
%%% Internal functions
%%%===================================================================

notify_viewers([], _) -> ok;
notify_viewers([H|T], Msg) ->
    async(H, Msg),
    notify_viewers(T, Msg).

make_updater(Sheet, Cell, Expression) ->
    case Expression of
        {formula, Fun, Deps} ->
            Pid = spawn(fun() -> update_server(Sheet, Cell, Fun, Deps, [], Deps)
                end),
            lists:map(fun(D) -> {ok, C} = cell(Sheet, D), add_viewer(C, Pid) end,
                 Deps),
            Pid;

        Value ->
            Func = fun(X) -> X, Value end,
            spawn(fun() -> update_server(Sheet, Cell, Func, [], [], []) end)
    end.
```

Figure 19: Spreadsheet solution, internal functions (../src/sheet/sheet.erl)

```
%%%===================================================================
%%% Communication primitives
%%%===================================================================
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
      {Pid, Response} -> Response
    end.

async(Pid, Msg) -> Pid ! Msg.

reply(From, Msg) -> From ! {self(), Msg}.
reply_ok(From, Msg) -> From ! {self(), {ok, Msg}}.
```

Figure 20: Spreadsheet solution, communication functions (../src/sheet/sheet.erl)

```
sheet_server(Cells) ->
    receive
        {From, {cell, A}} ->
            case dict:is_key(A, Cells) of
                true ->
                    {ok, [Pid|[]]} = dict:find(A, Cells),
                    reply_ok(From, Pid),
                    sheet_server(Cells);
                false ->
                    Sheet = self(),
                    P = spawn(fun() ->
                        cell_server(undefined, Sheet, A, undefined, []) end),
                    NewCells = dict:append(A, P, Cells),
                    io:format("Created cell ~p with pid ~p~n", [A,P]),
                    reply_ok(From, P),
                    sheet_server(NewCells)
            end;

        stop -> ok;

        Unknown ->
        io:format("~p received an unknown message ~p~n", [self(), Unknown]),
        sheet_server(Cells)
    end.
```

Figure 21: Spreadsheet solution, spreadsheet server loop (../src/sheet/sheet.erl)

```
cell_server(Updater, Sheet, Name, Value, Viewers) ->
    receive
        {_, {add_viewer, P}} ->
            notify_viewers([P], {updated, Name, Value, []}),
            case lists:member(P, Viewers) of
                false ->
                    cell_server(Updater, Sheet, Name, Value, [P|Viewers]);
                true ->
                    cell_server(Updater, Sheet, Name, Value, Viewers)
            end;

        {_, {remove_viewer, P}} ->
            cell_server(Updater, Sheet, Name, Value, lists:delete(P, Viewers));

        {From, get_viewers} ->
            reply(From, Viewers),
            cell_server(Updater, Sheet, Name, Value, Viewers);

        {Updater, {updated, Result}} ->
            notify_viewers(Viewers, {updated, Name, Result, []}),
            cell_server(Updater, Sheet, Name, Result, Viewers);

        {From, {set, Expression}} ->
            case Updater of
                undefined -> ok;
                OldUpdater -> async(OldUpdater, stop)
            end,
            NewUpdater = make_updater(Sheet, self(), Expression),
            reply_ok(From, ok),
            cell_server(NewUpdater, Sheet, Name, Value, Viewers);

        stop ->
            case Updater of
                undefined -> ok;
                OldUpdater -> async(OldUpdater, stop)
            end;

        Unknown ->
        io:format("cell ~p received an unknown message ~p~n", [self(), Unknown]),
        cell_server(Updater, Sheet, Name, Value, Viewers)
    end.
```

Figure 22: Spreadsheet solution, cell server loop (../src/sheet/sheet.erl)

```
update_server(_, Cell, Fun, Acc, [], []) ->
    Result = Fun(Acc),
    case Result of
        undefined -> reply(Cell, {updated, Result});
        error -> reply(Cell, {updated, Result});
        Value -> reply(Cell, {updated, {def, Value}})
    end;

update_server(Sheet, Cell, Fun, Deps, Acc, []) ->
    Result = Fun(Acc),
    lists:map(fun(D) -> {ok, C} = cell(Sheet, D), remove_viewer(C, self()) end,
        Deps),
    reply(Cell, {updated, {def, Result}}),
    update_server(Sheet, Cell, Fun, Deps, [], Deps);
```

Figure 23: Spreadsheet solution, update server loop (../src/sheet/sheet.erl)

```
update_server(Sheet, Cell, Fun, Deps, Acc, Queue) ->
    receive
        {updated, Name, {def, Value}, _} ->
            case lists:member(Name, Queue) of
                true ->
                    NewQueue = lists:delete(Name, Queue),
                    {ok, X} = cell(Sheet, Name),
                    remove_viewer(X, self()),
                    lists:map(fun(D) -> {ok, C} = cell(Sheet, D), add_viewer(C,
                        self()) end, NewQueue),
                    update_server(Sheet, Cell, Fun, Deps, [Value|Acc], NewQueue);
                false ->
                    update_server(Sheet, Cell, Fun, Deps, Acc, Queue)
            end;

        {updated, _, undefined, _} ->
            reply(Cell, {updated, undefined}),
            update_server(Sheet, Cell, Fun, Deps, [], Deps);

        {updated, _, error, _} ->
            reply(Cell, {updated, error}),
            update_server(Sheet, Cell, Fun, Deps, [], Deps);

        stop -> ok;

        Unknown ->
        io:format("updater ~p received an unknown message ~p~n", [self(), Unknown
            ]),
        update_server(Sheet, Cell, Fun, Deps, Acc, Queue)

    after 500 ->

        case Queue of
            Deps ->
                update_server(Sheet, Cell, Fun, Deps, [], Queue);
            Q ->
                reply(Cell, {updated, updating}),
                lists:map(fun(D) -> {ok, C} = cell(Sheet, D), add_viewer(C, self
                    ()) end, Q),
                update_server(Sheet, Cell, Fun, Deps, Acc, Q)
        end

    end.
```

Figure 24: Spreadsheet solution, update server loop (../src/sheet/sheet.erl)