

Refactoring for Verification: Utilizing Type Theory Programming to Verify Functional and Imperative Counterparts

Gregory Morse
gregory.morse@live.com

Eötvös Loránd Tudományegyetem/University (ELTE), Budapest, Hungary

Type Theory Seminar
November 26, 2020

Introduction

- Professional level terminal game in Haskell
- Proper non-blocking IO
- Random number generation
- Console control handler/signal handlers added and removed properly
- Windows and Linux compatible
- Conditional Compilation in Agda - one block as not reliable with `#if/#ifdef/#else/#endif` - which it will put in reverse order while moving imports elsewhere
- Everything can be just a Set in Agda and a “type” in Haskell, but “data” in Haskell can be used if mirroring the data structure in Agda and each element is spelled out in a special syntax for the compiler
- Type classes can be either ignored and specific types applied, or they could be postulated also and equated with a compile definition
- Runtime OS checking and combining routine for each routine jacketed out

Introduction (cntd.)

- Identification of useless code - harder to translate to Agda
- Identification of monad and let assignments which were premature
- Identifying bugs such as division by 0 due to proof strictness requirements
- Restructuring IO - nesting and order
- Pitfalls:
 - overuse of non-termination marking when not needed - should only be used for infinite loops or recursion, otherwise correction should be applied
 - overuse of return when no IO occurred which adds the infinite corecursion onto code which does not need it
 - not using let and letting the code get too messy

Keywords: Verification of Functional Programs, Agda, Haskell, JavaScript, C, Python, Refactoring for Verification


Snake game and terminal emulator

- Classic game example chosen due to its relative simplicity but complexity when dealing with the system level issues
- Meant to be an investigation of how type theory proof checking could be used to reason about an interactive game
- Many implementations in C but few if any do it portably or professionally
- Only a few implementations in Haskell exist but like with C not done portably or professionally
- Was first implemented in C, with full handling of early termination cases, Linux/Windows portable
- A small header and its corresponding C library wrapping the conditional compilation and Linux/Windows details such as non-blocking issues, or specific character sequences for the arrow keys
- Next implemented in Python, also with a replica Python library to wrap the same portability issues
- Next implemented in Haskell
- Finally implemented in Agda

Snake game and terminal emulator (cntd.)

- Implemented in JavaScript by emulating the terminal with a Canvas object
- Terminal emulator implemented then in: C, Haskell and Agda (compiles to JavaScript but not yet tested or working)
- Terminal emulator takes command line input for the width and height and then an input stream of VT100 compatible data and emits the final screen or it can emit the final screen as a series of plain text screens representing character data, colors and flags

Agda and its Standard Library on Linux/Windows

- Agda: <https://agda.readthedocs.io/>
- <https://wiki.portal.chalmers.se/agda/Main/Download>
- Latest Version 2.6.1.1 as of September 8, 2020
- Agda Standard Library: <https://github.com/agda/agda-stdlib>
- Latest Version 1.4 released on September 18, 2020
- Disadvantages: Many reorganizations, changes, not backward compatible
- Advantages: the changes made are steadily improving, making more consistent and stabilizing the library
- Alternative: Ulf Norell's Agda Prelude library:
<https://github.com/UlfNorell/agda-prelude>
- GNU Emacs: <https://www.gnu.org/software/emacs/download.html>
- Latest Version 27.1 released August 25, 2020
- Glasgow Haskell Compiler (GHC):
<https://www.haskell.org/platform/windows.html>
- Latest Version 8.10.2 released August 8, 2020
- Installed on Windows via Chocolatey: <https://chocolatey.org/install> 

Github Contribution in Agda

- Reported: No option to prevent Erasure makes use of IORef and FunPtr from Haskell not possible #3408
- <https://github.com/agda/agda/issues/3408>
- Ulf Norell took the bug portion to a new issue: No error if mapping the empty type to non-empty Haskell type #3409
- <https://github.com/agda/agda/issues/3409>
- Closed with commit:
<https://github.com/agda/agda/commit/841e6b1a00df2de37207a09d8a4>
- Reported: Bad JavaScript generated #5002
- <https://github.com/agda/agda/issues/5002>
- Moved to issue: JS backend: bugs involving “null” #4962
- <https://github.com/agda/agda/issues/4962>

Conditional Compilation

- Must use CPP mode which supports `#if/#ifdef/#else/#endif` preprocessor directives
- `agda --ghc --ghc-flag=-cpp`
- `{-# LANGUAGE CPP #-}` does not work
- It causes the `.hs` generated code to put the module after the `FOREIGN` directives unfortunately
- Would require some sort of special customization in emacs to pass this on a per `.agda` file basis e.g. defining a new Agda menu command

Conditional Compilation Type Checking Example

```
{-# FOREIGN GHC
#if defined(mingw32_HOST_OS)
import Foreign.C.Types (CInt (CInt))
import Data.Char (chr)
foreign import ccall unsafe "conio.h getch"
    c_getch :: IO CInt
foreign import ccall unsafe "conio.h kbhit"
    c_kbhit :: IO CInt
--dummy definitions from the Linux side
type CLong = CInt
type Fd = Integer
stdInput = 0
select'' _ _ _ _ = do return 0
#else
import Foreign.C.Types (CLong, CInt)
import System.Posix.Types (Fd)
import System.Posix.IO (stdInput)
import System.Posix.IO.Select (select'')
import System.Posix.IO.Select.Types (finite, Timeout (Never, Time),
    CTimeval)
--dummy definitions from the Windows side
chr :: Int -> Char
chr _ = '\x0'
c_getch :: IO CInt
c_getch = do return 0
c_kbhit :: IO CInt
c_kbhit = do return 0
#endif
```

Conditional Compilation Runtime

```
{-# FOREIGN GHC
toColist :: [a] -> MAlonzo.Code.Codata.Musical.Colist.AgdaColist a
toColist [] = MAlonzo.Code.Codata.Musical.Colist.Nil
toColist (x : xs) =
  MAlonzo.Code.Codata.Musical.Colist.Cons x (MAlonzo.RTE.Sharp (
    toColist xs))
postulate
  os : Costring
{-# COMPILER GHC os = toColist System.Info.os #-}
data _⊥ {a} (A : Set a) : Set a where
  now   : (x : A) → A ⊥
  later : (x : ∞ (A ⊥)) → A ⊥
rev : Costring → String ⊥
rev = go []
  where
    go : List Char → Costring → String ⊥
    go acc [] = now (Data.String.fromList acc)
    go acc (x :: xs) = later (λ go (x :: acc) (b xs))
{-# NON_TERMINATING #-}
getStr : String ⊥ → String
getStr (now s) = s
getStr (later s) = getStr (b s)
isWin : Bool
isWin = (Data.String.fromList (reverse (Data.String.toList (getStr
  (rev os))))) == "mingw32"
```

Conditional Compilation Runtime (cntd.)

```
getCh : _
getCh = if isWin then ((# (lift (hIsTerminalDevice stdin))) >>= λ
  isTerm → # (if isTerm then (# (lift c_getch)) >>= (λ ch → #
    return (chr (fromEnum ch))) else (# return Data.Unit.tt >>= λ _
      → # lift (hGetChar stdin)))) else (# return Data.Unit.tt) >>=
  (λ _ → # lift (hGetChar stdin))

chkKB : IO.IO Bool
chkKB = if isWin then (# lift c_kbhit >>= λ is → # return (not [ (
  toInteger (fromEnum is)) Data.Nat.? 0 ])) else (# lift (select'
  , (stdInput :: []) [] [] (finite (fromInteger 0) (fromInteger 0)
  ))) >>= λ x → # return [ (toInteger (fromEnum x)) Data.Nat.? 1
  ]
#-}
```

Early Termination Handling

- Requires Signal Handlers in Linux for signals generated like SIGINT on Ctrl+C
- Requires Console Control Handler in Windows for Ctrl+C or Ctrl+Break
- Haskell library requires threading, this detail is not equivalent in expressability to raw C code
- `-ghc-flag=-threaded`
- Not specifiable via pragmas
- On Linux, `installHandler` with `sigINT` and `sigTERM` which requires saving the main thread ID, and takes a function directly
- On Linux, `throwTo` with a thread ID will allow an `exitFailure` to be sent to the main thread
- On Windows, must obtain function pointers which use the `IORef` wrapper via `newIORef`, `writelIORef` (`,` `readIORef`) and use `SetConsoleCtrlHandler`
- On Windows, the boolean return value from the handler will decide termination or continuation

Pseudo Random Number Generator (PRNG)

- Pseudo Random Number Generator (PRNG) in Agda
- What to prove?
- The standard mathematical proofs of a perfect RNG that in a given discrete interval, there is an equal probability of each value occurring
- Obviously a perfect RNG does not exist
- For a PRNG instead of equal probability, the probability would be less than some error parameter, still very difficult to prove
- How about for a periodic algorithm like the classic Mersenne Twister MT19937
- https://en.wikipedia.org/wiki/Mersenne_Twister
- Prove that the period is $2^{19937} - 1$?
- Prove that it is k-distributed to v-bit accuracy?
- Probability Monads in Cubical Agda
- <https://doisinkidney.com/posts/2019-04-17-cubical-probability.html>
- Standard solution is to use one from Haskell
- <https://hackage.haskell.org/package/random>
- `cabal install -lib random`

PRNG (cntd.)

- `System.Random`: pure pseudo-random number interface
- vs. `System.Random.Stateful`: monadic pseudo-random number interface
- Underlying algorithm is `splitmix`:
<https://hackage.haskell.org/package/splitmix>
- Splittable pseudorandom number generator that is quite fast: 9 64 bit arithmetic/logical operations per 64 bits generated.
- Guy L. Steele, Jr., Doug Lea, and Christine H. Flood. 2014. Fast splittable pseudorandom number generators. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14). ACM, New York, NY, USA, 453-472.
- Not suitable for cryptographic or security uses

PRNG (cntd.)

- IO wrapper must be used to allow for delay and repeated invocation
- `Random.randomIO` exists for this purpose
- Alternatively IO `return` can explicitly wrap into the IO monadic context
- Wrapping PRNG result with `let` or as a function argument, etc will not work as substitution will cause repeated generation

PRNG Example adapting to IO

```
{-# FOREIGN GHC
data RandomDict a = R.Random a => RandomDict
data RandomGenDict g = R.RandomGen g => RandomGenDict
#-}
RStdGen : Set
RmkStdGen : Int → RStdGen
RandomGen : Set → Set
instance RandomGenStdGen : RandomGen RStdGen
Random : Set → Set
instance RandomNat : Random ℕ
RRandomR : {a g : Set} {{_ : Random a}} → {{_ : RandomGen g}}
          → (Pair a a) → g → Pair a g

{-# COMPILE GHC RStdGen = type R.StdGen #-}
{-# COMPILE GHC RmkStdGen = R.mkStdGen #-}
{-# COMPILE GHC RandomGen = type RandomGenDict #-}
{-# COMPILE GHC RandomGenStdGen = RandomGenDict #-}
{-# COMPILE GHC Random = type RandomDict #-}
{-# COMPILE GHC RandomNat = RandomDict #-}
{-# COMPILE GHC RRandomR = \ _ _ RandomDict RandomGenDict -> R.
  randomR #-}
{-# FOREIGN GHC
import qualified System.Random as R (Random, RandomGen, randomR,
  mkStdGen, StdGen)
#-}
open import IO public using (putStrLn)
open import Agda.Builtin.String public using (primShowNat)
rngExample : -
rngExample = run (♯ (♯ lift getPOSIXTime >= λ tm → ♯ (return (
  RandomR (0 , 100) (RmkStdGen (round tm)))))) >= λ p → ♯
  putStrLn (primShowNat (Pair.fst ‘ ‘ p)))
```


Another PRNG example which uses randomIO

<https://stackoverflow.com/questions/55631108/how-to-generate-random-numbers-in-agda>

```
open import Agda.Builtin.Float
import IO.Primitive as Prim
open import IO
random : IO Float
random = lift primRandom where
  postulate primRandom : Prim.IO Float
  {-# FOREIGN GHC import qualified System.Random as Random #-}
  {-# COMPILER GHC primRandom = Random.randomIO #-}
open import Codata.Musical.Notation
open import Function
main : Prim.IO _
main = run $
  # random >>= λ f → # putStrLn (primShowFloat f)
```

Formal Verification of input/output (IO)

- Beauty in the Beast: A Functional Semantics for the Awkward Squad by Wouter Swierstra, Thorsten Altenkirch
- <https://webpace.science.uu.nl/swier004/publications/2007-haskell.pdf>
- Agda interactive application/GUI/widget library: https://github.com/divipp/frp_agda by Péter Diviánszky
- VT100 terminal IO uses stdin for indicating certain things such as the current cursor position
- Dynamic querying for terminal size involves setting cursor position to maximum 999, 999 and querying cursor position
- Terminal can be represented as a rows * columns list or list of rows containing lists of columns
- Terminal emulator can take a stream of input and convert it into this row by column representation
- However not just character data appears at each position, but color information, state information
- Non-blocking IO has a sleep timeout call which represents no IO

Agda JavaScript (JS) generation

- `agda --js [--compile-dir=<output directory>]`
- Compiles the entire standard library so using an output directory is desirable
- Compiles to node.js form which uses `require` and `export` method
- A plugin called `Browserify` exists to take node.js code and allow it to run in a client browser context: <http://browserify.org/>
- However for the sake of having maximum control over the details of the JS code without learning how to do certain things in a more complicated or at least indirect node.js/Browserify context, a different idea of a custom loader
- Must load code asynchronously but the recursive `require` directives must be synchronous
- Option of adding `script` objects into the Document Object Model (DOM) of the Hyper Text Markup Language (HTML) tree which is effectively similar to a static load

Agda JS generation (cntd.)

- Option of using `XMLHttpRequest` to do dynamic loading but this runs into security issues if done locally and can only be allowed if changing special global options or command line options to browsers like Google Chrome or Microsoft Edge
- Option of loading the .js files as strings and using `eval` but this runs into even more stringent security issues for both opening local files and evaluating foreign code
- Need to do a dependency sort so a short Python script does just this by using topological ordering via reverse post-order of depth first search (DFS)
- A lot of primitive functions are left undefined, partly because no maintainer for the JS compiler in Agda
- Many of these primitive functions such as `primNatToChar` or `primCharToNat` are trivial to implement
- ECMAScript 6 drastically changed JS and introduced features such as proxy objects
- Strict evaluation vs. lazy evaluation in GHC generated code

Agda JS generation (cntd.)

- Leads to issues in musical notation delay (\sharp) which is represented as delayed (∞) and flattening (b) specifically
- Primitive IO only requires two wrappers: continuation operator \gg and return
- The async/await model of javascript perfectly aligns with the primitive IO wrappers which is necessary for non-blocking IO e.g. keyboard input and updating the terminal (which is an HTML Canvas object)
- The proxy object allows for the get operation on the JS object dictionaries to be overridden and customized with custom provided code
 - Two bugs were fixed in the generated Agda Standard Library code in the proxy
 - All the primitive functions necessary were filled
 - The postulated functions in the main module were filled

- Saving terminal output from a C program and sending it through Agda terminal emulator
- Snake game in Agda in console
- Snake game in Agda in browser
- More in depth study of Snake game Agda source code and brief overview of terminal emulator
- In depth study of Snake game Agda JS code wrapper

Some lines of inquiry

- Is there a way to take the immutability concepts from functional languages and import them to imperative languages without losing efficiency e.g. via macros?
- Is there a way to take the recursion concepts from functional languages and import them to imperative languages without using the stack e.g. dynamic translation to loops via a series of complicated macros?
- Can a C to Haskell generator be made for at least a subset of C programs?
- Can a Haskell to Agda generator be made for at least a subset of Haskell programs?
- Can unit tests or automated testing occur via Agda generation to GHC and JavaScript and execution?

Conclusion and Future Research

- Easier to translate to Agda by first translating to Haskell
- JavaScript testing can reveal subtle lazy evaluation order bugs and enhances the Agda code
- Agda Standard Library
- Conditional compilation should get some sort of proper support in Agda at some point