Haskell for Grownups

Bill Harrison

May 14, 2024

Table of Contents

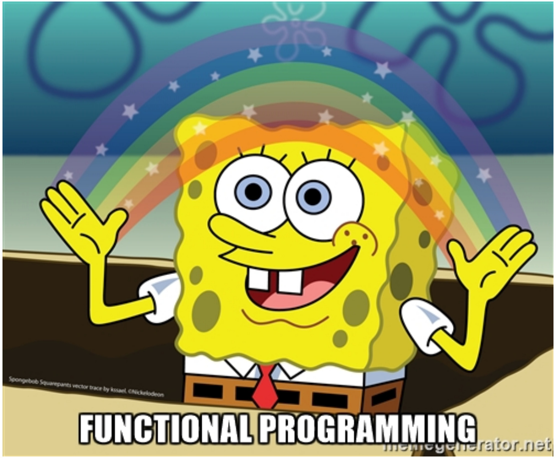
Haskell Basics

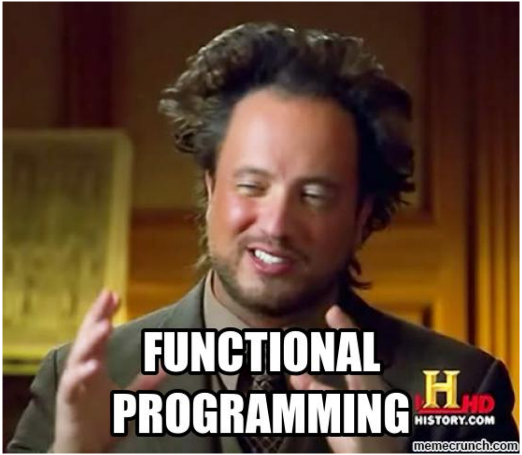
- Modern (pure) lazy functional language
- “Pure” means “takes types **really** seriously”
- Statically typed, supports type inference
- Compilers and interpreters:
  - http://www.haskell.org/implementations.html
  - GHC Compiler
  - GHCi interpreter
- A peculiar language feature: indentation & capitalization matter

Some Reference Texts

- *Programming in Haskell* by Graham Hutton.
  This is an excellent, step-by-step introduction to Haskell. Graham also has a lot of online resources (slides, videos, etc.) to go along with the book.
- *A Gentle Introduction to Haskell* by Hudak, Peterson, and Fasal.
  Available at `http://www.haskell.org/tutorial/`.
- *Learn You a Haskell for Good* by Miran Lipovaca.
  Highly amusing and informative; available online.
- *Real World Haskell* by Bryan O'Sullivan.
  Also available online (I believe). "Haskell for Working Programmers".
- Google.

Table of Contents


Spongebob Squarepants vector trace by kssael ©Nickelodeon
FUNCTIONAL PROGRAMMING
memegenerator.net


FUNCTIONAL
PROGRAMMING
HD
HISTORY.COM
memecrunch.com


I LOVE FUNCTIONAL PROGRAMMING
IT MAKES PEOPLE CRY

Table of Contents

Question: What does this program do?

```
n = i;
a = 1;
while (n > 0) {
    a = a * n;
    n = n - 1;
}
```

Functions in Mathematics

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

Functions in Mathematics

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

What does this have to do with that?

```
n = i;
a = 1;
while (n > 0) {
    a = a * n;
    n = n - 1;
}
```

First Haskell Function

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

First Haskell Function

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

It’s relationship to this Haskell function is apparent:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```


DID SOMEONE SAY
HASKELL


Your Main language is an 'actual' Programming Language
Your main language is Haskell

Hello World in C

```
#include <stdio.h>
int main() {
  printf("hello␣world\n");
}
```

Hello World in Haskell

```
module HelloWorld where
helloworld :: IO ()
helloworld = print "Hello␣World"
```


AND THIS IS HOW
WE WRITE HELLO WORLD
IN HASKELL

Factorial Revisited

```
#include <stdio.h>
int fac(int n) {
  if (n==0)
    { return 1; }
  else
    { return (n * fac (n-1)); }
}

int main() {
  printf("Factorial␣5␣=␣%d\n",fac(5));
  return 0;
}
```

Hello Factorial

```
#include <stdio.h>
int fac(int n) {
  printf("hello␣world");      // new
  if (n==0)
    { return 1; }
  else
    { return (n * fac (n-1)); }
}
    ...
```

Hello Factorial

```
#include <stdio.h>
int fac(int n) {
  printf("hello␣world");      // new
  if (n==0)
    { return 1; }
  else
    { return (n * fac (n-1)); }
}
    ...
```

(N.b., the type is the same)

```
int fac(int n) {...}
```

Hello Factorial in Haskell

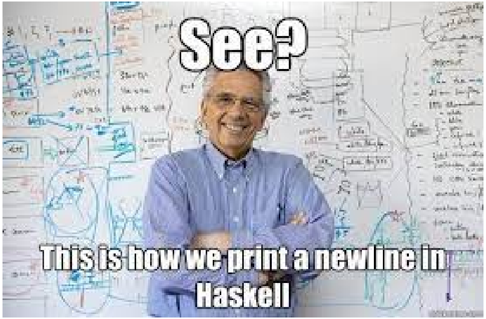
```
fac :: Int -> IO Int -- the type changed
fac 0 = do print "hello␣world"
           return 1
fac n = do print "hello␣world"
           i <- fac (n-1)
           return (n * i)
```

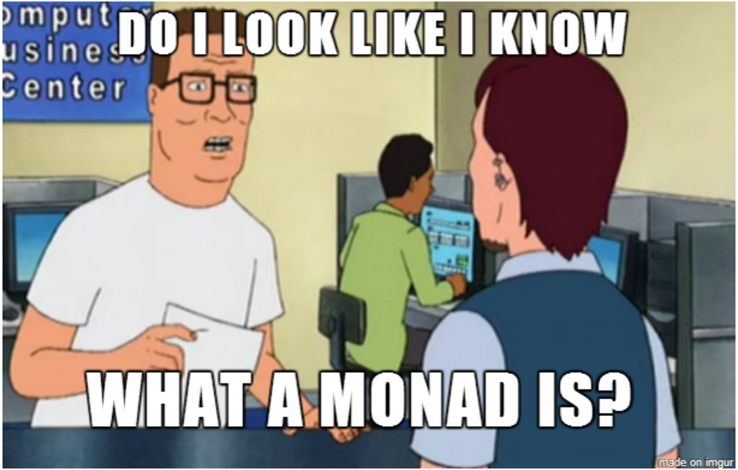
Hello Factorial in Haskell

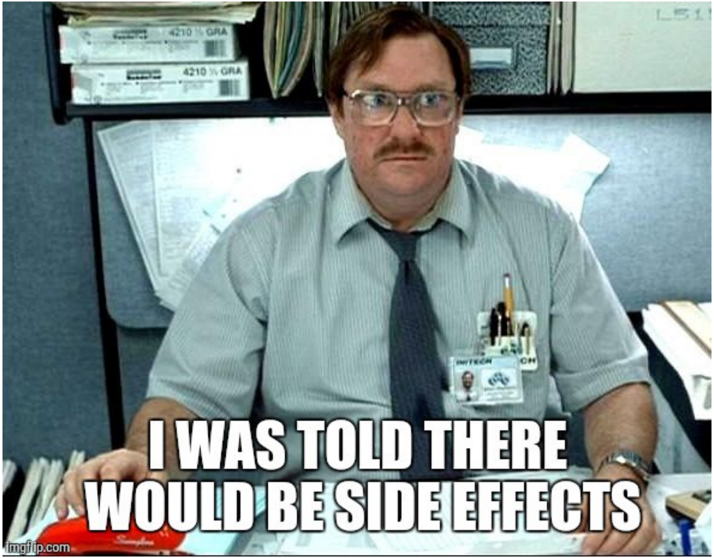
```
fac :: Int -> IO Int -- the type changed
fac 0 = do print "hello␣world"
           return 1
fac n = do print "hello␣world"
           i <- fac (n-1)
           return (n * i)
```

(Moral of the Story)

- *Haskell types are a contract telling you a lot about what the program can and can't do*
- *C types are documentation basically*


See?
This is how we print a newline in Haskell


DO I LOOK LIKE I KNOW
WHAT A MONAD IS?
made on imgur


I WAS TOLD THERE
WOULD BE SIDE EFFECTS
imgflip.com

Why Functional Languages?

Definition

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Why Functional Languages?

Definition

$$\texttt{length} :: [a] \rightarrow \texttt{Int}$$
$$\texttt{length}\,[\,] = 0$$
$$\texttt{length}\,(\texttt{x} : \texttt{xs}) = 1 + \texttt{length}\,\texttt{xs}$$

Theorem

$$\texttt{length}\,(\texttt{xs} +\!\!+ \texttt{ys}) = \texttt{length}\,\texttt{xs} + \texttt{length}\,\texttt{ys}$$

Why Functional Languages?

Definition

$$\texttt{length} :: [a] \rightarrow \texttt{Int}$$
$$\texttt{length}\,[\,] = 0$$
$$\texttt{length}\,(x : xs) = 1 + \texttt{length}\,xs$$

Theorem

$$\texttt{length}\,(xs +\!\!+\, ys) = \texttt{length}\,xs + \texttt{length}\,ys$$

Proof

$$
\begin{aligned}
&\texttt{length}((z : zs) +\!\!+\, ys) \\
&= \texttt{length}\,(z : (zs +\!\!+\, ys)) && +\!\!+ \text{ defn.} \\
&= 1 + \texttt{length}\,(zs +\!\!+\, ys) && \texttt{length} \text{ defn.} \\
&= 1 + \texttt{length}\,zs + \texttt{length}\,ys && \text{induction hyp.} \\
&= \texttt{length}\,(z : zs) + \texttt{length}\,ys && \texttt{length} \text{ defn.}
\end{aligned}
$$

Why Functional Languages?

Definition

$\texttt{length} :: [a] \rightarrow \texttt{Int}$
$\texttt{length}\,[\,] = 0$
$\texttt{length}\,(x : xs) = 1 + \texttt{length}\,xs$

Theorem

$\texttt{length}\,(xs +\!\!+\; ys) = \texttt{length}\,xs + \texttt{length}\,ys$

Proof

$\texttt{length}((z : zs) +\!\!+\; ys)$
$= \texttt{length}\,(z : (zs +\!\!+\; ys))$ — $+\!\!+$ defn.
$= 1 + \texttt{length}\,(zs +\!\!+\; ys)$ — length defn.
$= 1 + \texttt{length}\,zs + \texttt{length}\,ys$ — induction hyp.
$= \texttt{length}\,(z : zs) + \texttt{length}\,ys$ — length defn.

Mechanically-Checked Proof

```
length-++ : ∀ {A : Set} (xs ys : List A)
  → length (xs ++ ys) ≡ length xs + length ys
length-++ {A} [] ys    = …
length-++ (x ∷ xs) ys = …
```

Why Functional Languages?

Definition

$$\texttt{length} :: [a] \rightarrow \texttt{Int}$$
$$\texttt{length}\,[\,] = 0$$
$$\texttt{length}\,(x : xs) = 1 + \texttt{length}\,xs$$

Theorem

$$\texttt{length}\,(xs +\!\!+\, ys) = \texttt{length}\,xs + \texttt{length}\,ys$$

Proof

$$\begin{aligned}
&\texttt{length}((z : zs) +\!\!+\, ys) \\
&= \texttt{length}\,(z : (zs +\!\!+\, ys)) && +\!\!+\ \text{defn.} \\
&= 1 + \texttt{length}\,(zs +\!\!+\, ys) && \texttt{length}\ \text{defn.} \\
&= 1 + \texttt{length}\,zs + \texttt{length}\,ys && \text{induction hyp.} \\
&= \texttt{length}\,(z : zs) + \texttt{length}\,ys && \texttt{length}\ \text{defn.}
\end{aligned}$$

Mechanically-Checked Proof

```
length-++ : ∀ {A : Set} (xs ys : List A)
  → length (xs ++ ys) ≡ length xs + length ys
length-++ {A} [] ys    = …
length-++ (x ∷ xs) ys = …
```

- Supports scalable formal methods across the assurance spectrum
  - automated test generation (quickcheck)
  - security, safety, & privacy type systems
  - formal verification (Lean, Coq, Isabelle,...)

Data Types + Functions = Haskell Programs

Haskell programming is both data type and functional programming!

- Arithmetic interpreter
  - data type:

```
data Exp = Const Int | Neg Exp | Add Exp Exp
```

  - function:

```
interp :: Exp -> Int
interp (Const i)    = i
interp (Neg e)      = - (interp e)
interp (Add e1 e2) = interp e1 + interp e2
```

Data Types + Functions = Haskell Programs

Haskell programming is both data type and functional programming!

- Arithmetic interpreter
  - data type:
    ```
    data Exp = Const Int | Neg Exp | Add Exp Exp
    ```
  - function:
    ```
    interp :: Exp -> Int
    interp (Const i)    = i
    interp (Neg e)      = - (interp e)
    interp (Add e1 e2) = interp e1 + interp e2
    ```
- How do Haskell programs use data?
  - Patterns break data apart to access:
    "`interp (Neg e) =...`"
  - Functions recombine into new data:
    "`interp e1 + interp e2`"

Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

- `Bool` is a new type.
- `False` and `True` are called **constructors** for `Bool`.
- Type and constructor names begin with upper-case letters.
- Data declarations are similar to context free grammars.

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

`Nat` is a new type, with constructors

```
Zero :: Nat
Succ :: Nat -> Nat
```

Note:

- A value of type `Nat` is either `Zero`, or of the form `Succ n` where n :: Nat. That is, `Nat` contains the following infinite sequence of values:

```
Zero
Succ Zero
Succ (Succ Zero)
      ⋮
```

Note:

- We can think of values of type Nat as natural numbers, where `Zero` represents `0`, and `Succ` represents the successor function `1+`.
- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))
```

Recursive Data beget Recursive Functions

Recursive functions convert between values of type `Nat` and `Int`:

```
nat2int          :: Nat -> Int
nat2int Zero      = 0
nat2int (Succ n)  = 1 + nat2int n

int2nat          :: Int -> Nat
int2nat 0         = Zero
int2nat n         = Succ (int2nat (n - 1))
```

Data Types, cont'd

```
data Maybe a = Nothing | Just a
```

```
safediv    :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

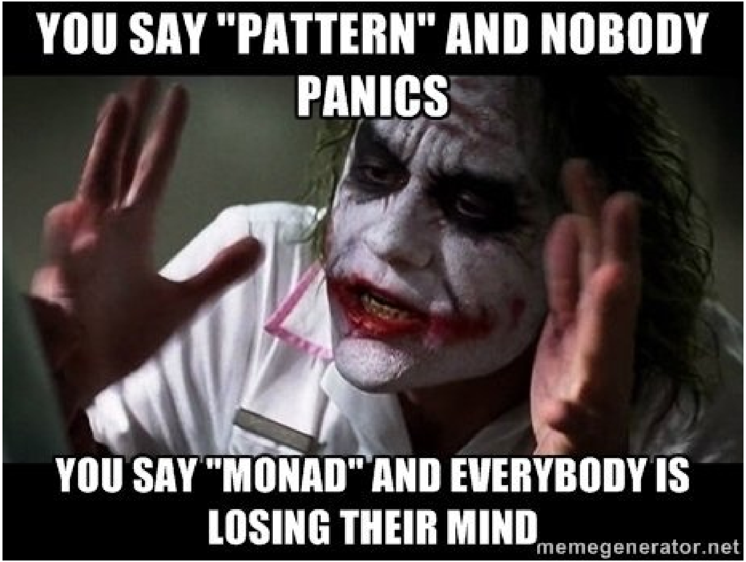

YOU SAY "PATTERN" AND NOBODY PANICS
YOU SAY "MONAD" AND EVERYBODY IS LOSING THEIR MIND
memegenerator.net

Table of Contents

Type-Driven Programming in Haskell

Types first, then programs

- Writing a function with type $A \to B$, then you have a lot of information to use for fleshing out the function.
- Why? Because the input type A — *whatever it happens to be* — has a particular form that determines a large part of the function itself.
- This is, in fact, the way that you should develop Haskell programs.

The edit-compile-test-until-done paradigm

I'm guessing that this is familar to you

When I was a student—the process of writing a C program tended to follow these steps:

1. Create/edit a version of the whole program using a text editor.
2. Compile. If there were compilation errors, develop a hypothesis about what the causes were and start again at 1.
3. Run the program on some tests. Do I get what I expect? If so, then declare victory and stop; otherwise, develop a hypothesis about what the causes were and start again at 1.

An Exercise

- Write a function that
  1. takes a list of items,
  2. takes a function that returns either `True` or `False` on those items,
  3. and returns a list of all the items on which the function is true.
- This is called *filter*, and it's a built-in function in Haskell, but let me show you how I'd write it from scratch.
  - I call the function I'm writing "`myfilter`" to avoid the name clash with the built-in version.

Step 1. Figure out the type of the thing you're writing

- Think about the type of `filter` and write it down as a type specification in a Haskell module (called `Sandbox` throughout).
- With what I've said about `filter`, it takes a list of items—i.e., something of type `[a]`.
- It also takes a function that takes an item—an `a` thing—and returns true or false—i.e., it returns a `Bool`. So, this function will have type `a` → `Bool`.
- ∴ the type should be:

```
myfilter :: [a] -> (a -> Bool) -> [a]
```

Step 2: Fill in the type template & load the module.

- In this case, we have a function with two arguments. The second argument of type `a->Bool` does not have a matchable form like the first argument.
- This leaves us with:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f     = undefined
myfilter (x:xs) f = undefined
```

Step 2: Fill in the type template & load the module.

- In this case, we have a function with two arguments. The second argument of type `a->Bool` does not have a matchable form like the first argument.
- This leaves us with:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f       = undefined
myfilter (x:xs) f = undefined
```

- A dumb mistake like:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f    = undefined
myfilter (x:xs) = undefined
```

  would be caught automatically by the type-checker.
- I.e., Debugging via Type-checking!

Step 3: Fill in the clauses one-by-one reloading as you go.

The [] case is obvious because there is nothing to filter out:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f      = []
myfilter (x:xs) f = undefined
```

No problems with this last bit:

```
> ghci Sandbox.hs
[1 of 1] Compiling Sandbox
Ok, modules loaded: Sandbox.
⋆Sandbox>
```

Step 3 (continued).

- The second clause should only include `x` if `f x` is `True`; one way to write that is with an `if−then−else`:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f     = []
myfilter (x:xs) f = if f x
                       then x : myfilter f xs
                       else myfilter f xs
```

- Loading this into GHC reveals a problem:

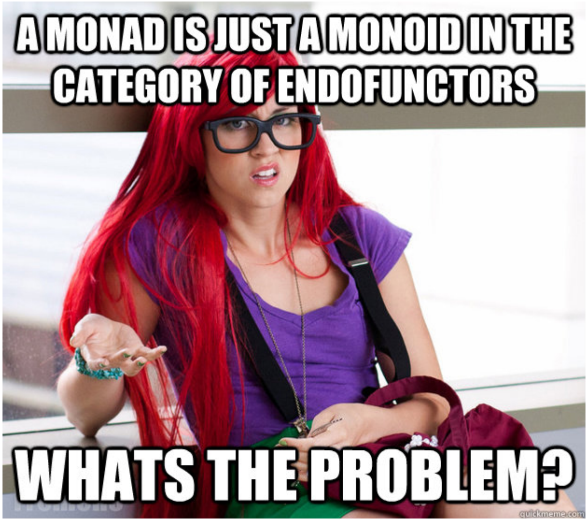
```
> ghci Sandbox.hs
[1 of 1] Compiling Sandbox          ( Sandbox.hs, interpreted )
Sandbox.hs:8:46:
    Couldn't match expected type '[a]' with actual type 'a -> Bool'
    In the first argument of 'myfilter', namely 'f'
    In the second argument of '(:)', namely 'myfilter f xs'
    In the expression: x : myfilter f xs
Failed, modules loaded: none.
Prelude>
```

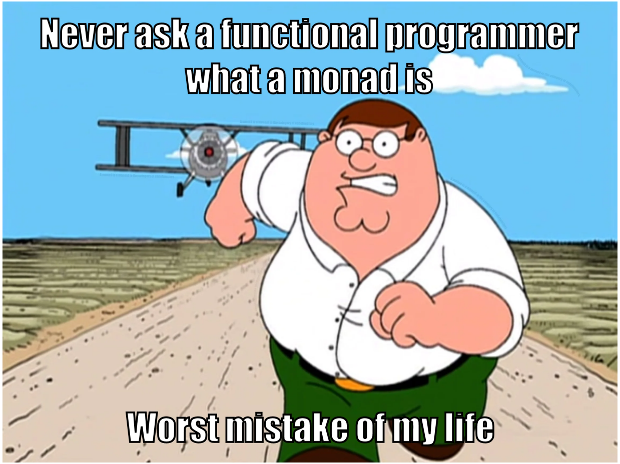
Step 3 (continued).

- This error occurs on line 8 of the module, which is the line "`then x : myfilter f xs`". GHCi is telling us that it expects that `f` would have type [`a`] but that it can see that `f` has type `a` → `Bool`. After a moment's pause, we can see that the order of the arguments is incorrect in both recursive calls. The corrected version works:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f     = []
myfilter (x:xs) f = if f x
                      then x : myfilter xs f
                      else myfilter xs f
```

Table of Contents


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS
WHATS THE PROBLEM?
quickmeme.com


Never ask a functional programmer
what a monad is
Worst mistake of my life

Programming Languages are Monads

- Periodic Table of Programming Languages

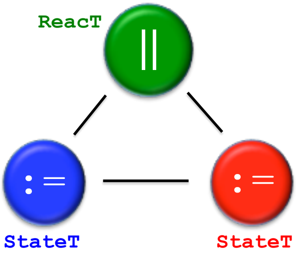
StateT *imperative* `:=`		BackT *backtracking* `cut`	ResT *threads* `step pause`
EnvT *binding* `λ @ v`	ErrorT *exceptions* `raise/catch`	ContT *continuations* `callcc`	NondetT *non-determ.* `choose`
	IoT *input/output* `printf`	DebugT *debugging* `rollback`	ReactT *reactivity* `send,recv,...`

- Moggi 1989: Languages are "molecules" composed of "elements" (aka, *monad transformers*)

- Haskell has
  - built-in monad syntax
  - formal semantics [JFP05,APLAS05]
- **Systems** are molecules
  - Compilers [ICCL98,MPC00]
  - Interrupts/asynchronous exceptions [MPC08]
  - Systems Biology [EMBC03]
  - POSIX-like kernels [AMAST06,CheapThreads]
  - Separation kernels [FCS03,CSF05,JCS09,ICFEM12]
  - Synchronous Hardware [FPT13/15,ARC15,ReCoSoC16, RSP16,TECS17,TECS19]

Monads are Programming Language Constructors

- Language “Molecule”

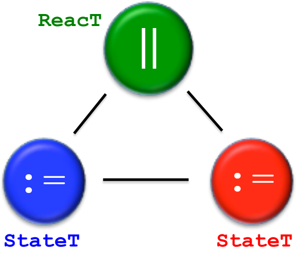


- ...constructs a language:

  :=, mask, :=, mask,
  ||, signal, ;

Monads are Programming Language Constructors

- Language "Molecule"



- ...constructs a language:

  :=, mask, :=, mask,
  ||, signal, ;

- With By-Construction Algebraic Properties [APLAS06,JCS09]:

$$a := x \; ; \; b := y = b := y \; ; \; a := x$$
$$a := x \; ; \; \texttt{mask} = \texttt{mask}$$
$$b := y \; ; \; \texttt{mask} = \texttt{mask}$$

- Each "element" adds new commands to the language "molecule"

Table of Contents

Achieving information flow security through monadic control of effects [HH09]

Classic Goguen-Meseguer Noninterference:

“changes in high-level inputs only change high-level outputs”

Monadic language approach [HH05, HH09, WHA12, PHG+15, PHG+17]**:**

“high-level operations must cancel”

“Bird-Wadler”’ Equational Reasoning

$x_1 := e_1$; $y_1 := f_1$; $x_2 := e_2$; maskHi

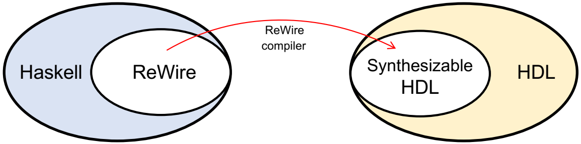
= $x_1 := e_1$; $y_1 := f_1$; maskHi

= $x_1 := e_1$; maskHi ; $y_1 := f_1$

= maskHi ; $y_1 := f_1$

= $y_1 := f_1$; maskHi

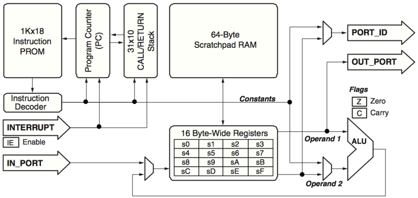
ReWire Language & Toolchain



- Inherits Haskell's good qualities
  - Pure functions, strong types, monads, equational reasoning, etc.
  - Denotational semantics [HK05, Har05, HSH02]
- Types & Operators for HW abstractions [HPG+16]
- ReWire Compiler (`rwc`) produces Verilog, VHDL, or FIRRTL
- Formalized Semantics in Coq [RPHA19] and Isabelle/Coq/Agda [HBB+23]
  - Embedding Tool translates ReWire into Isabelle

Semantics-directed Architecture in ReWire [FPT2013]

Xilinx PicoBlaze 8-bit Embedded Microcontroller



Data Layout

```
type RegFile  = Table W4 W8
type FlagFile = (Bit,Bit,Bit,Bit,Bit)
type Mem      = Table W6 W8
data Stack    = Stack { contents :: Table W5 W10,
                        pos :: W5 }
data Inputs   = Inputs { instruction_in :: W18,
                         in_port_in     :: W8,
                         interrupt_in   :: Bit,
                         reset_in       :: Bit }
data Outputs  = Outputs { address_out       :: W10,
                          port_id_out       :: W8,
                          write_strobe_out  :: Bit,
                          out_port_out      :: W8,
                          read_strobe_out   :: Bit,
                          interrupt_ack_out :: Bit }
```

Fetch-Decode-Execute

```
pico :: Dev Inputs PicoState Outputs
pico = do s <- getPicoState
          let i = inputs s
              instr = instruction_in i
          ie <- getFlagIE
          if reset_in i == 1
             then reset_event
             else if ie == 1 &&
                     interrupt_in i == 1
                     then interrupt_event
                     else decode instr
          pico
```

RV32i in ReWire

Undergraduate Capstone at Univ. of Missouri (2019)

Fetch-Decode-Execute

```
rv32i :: Monad m =>
         ReacT
            (InSig w (Instr))
            (OutSig W32 w e)
            (StateT RegFile (StateT (InSig w (Instr),OutSig W32 w e) m))
            ()
rv32i = do
  pc ← lift $ getReg PC
  iw ← async_fetch pc
  exec iw
  rv32i

exec :: Monad m => Instr → ReacT i o (StateT RegFile (StateT (i, o) m)) ()
exec c = case c of

     Add rd rs1 rs2   → do
       lift $ do
                rs1  ← getReg rs1
                rs2  ← getReg rs2
                putReg rd (rs1 + rs2)
       tick
         etc.
```

References I

- William Harrison.
  A simple semantics for polymorphic recursion.
  In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS05)*, pages 37–51, Tsukuba, Japan, November 2005.

- William L. Harrison, Ian Blumenfeld, Eric Bond, Chris Hathhorn, Paul Li, May Torrence, and Jared Ziegler.
  Formalized high level synthesis with applications to cryptographic hardware.
  In *NASA Formal Methods Symposium (NFM23)*, 2023.

- William Harrison and James Hook.
  Achieving information flow security through precise control of effects.
  In *18th IEEE Computer Security Foundations Workshop (CSFW05)*, pages 16–30, Aix-en-Provence, France, June 2005.

References II

W. Harrison and J. Hook.
Achieving information flow security through monadic control of effects.
JCS, 17:599–653, Oct 2009.

William L. Harrison and Richard B. Kieburtz.
The logic of demand in Haskell.
Journal of Functional Programming, 15(6):837–891, 2005.

W. Harrison, A. Procter, I. Graves, M. Becchi, and G. Allwein.
A programming model for reconfigurable computing based in functional concurrency.
In *11th Inter. Symp. on Reconfigurable Communication-centric Systems-on-Chip*, 2016.

References III

- William Harrison, Timothy Sheard, and James Hook.
  Fine control of demand in Haskell.
  In *6th International Conference on the Mathematics of Program Construction (MPC02), Dagstuhl, Germany*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer-Verlag, 2002.
- Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein.
  Semantics driven hardware design, implementation, and verification with ReWire.
  In *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, 2015.
- A. Procter, W. Harrison, I. Graves, M. Becchi, and G. Allwein.
  A principled approach to secure multi-core processor design with ReWire.
  ACM TECS, 16(2):33:1–33:25, February 2017.

References IV

Thomas N. Reynolds, Adam Procter, William L. Harrison, and Gerard Allwein.
The mechanized marriage of effects and monads with applications to high-assurance hardware.
ACM Transactions on Embedded Computing Systems, 18(1):6:1–6:26, January 2019.

A. Procter W. Harrison and G. Allwein.
The confinement problem in the presence of faults.
In *ICFEM*, pages 182–197, 2012.