Executable, Synthesizable, Human Readable: Pick Three
Pong on an FPGA with Clash

Gergő Érdi
https://unsafePerform.IO/

Haskell Love, 10\textsuperscript{th} September 2021.
Solitaire Pong
1. Software implementation
Game state

- data St
- initState :: St
- updateState :: Inputs -> St -> St
Game state

- data St
- initState :: St
- updateState :: Inputs -> St -> St

type Coord = Signed 10

data St = MkSt
    { _ballH, _ballV :: (Coord, Coord)
    , _paddleY :: Coord
    , _gameOver :: Bool
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''St
Game state

- data St
- initState :: St
- updateState :: Inputs -> St -> St

initState :: St
initState = MkSt
  { _ballH = (10, 2)
    , _ballV = (100, 3)
    , _paddleY = 100
    , _gameOver = False
  }
Game state

- data St
- initState :: St
- updateState :: Inputs -> St -> St

data Inputs = MkInputs
  { paddleUp, paddleDown :: Bool
  }

updateState inp = execState $ do
  updateBall     -- Bounce on walls and paddle
  updatePaddle inp -- Move paddle based on user input
  checkBounds   -- Ball out of bounds?
Decomposed in vertical (walls only) and horizontal (walls or paddle) direction:

```haskell
updateBall :: State St ()
updateBall = do
    updateVert
    updateHoriz
```
Decomposed in vertical (walls only) and horizontal (walls or paddle) direction:

```haskell
updateBall :: Inputs -> State St ()
updateBall MkInputs{..} = do
    updateVert
    hitPaddle <- updateHoriz
    when hitPaddle $ ballV._2 += nudge
where
    nudge | paddleDown = nudgeSpeed
    | paddleUp   = negate nudgeSpeed
    | otherwise  = 0
```
type ScreenWidth = 256
type ScreenHeight = 200

type X = Index ScreenWidth
type Y = Index ScreenHeight

type Color = (Unsigned 8, Unsigned 8, Unsigned 8)

draw :: St -> X -> Y -> Color
draw MkSt{..} x y
  | isWall = white
  | isPaddle = blue
  | isBall = yellow
  | otherwise = if _gameOver then red else gray
High-level simulation

Everything we have so far is normal (and nice!) Haskell code
High-level simulation

Everything we have so far is normal (and nice!) Haskell code – we can run it as such, using e.g. SDL2 for IO:

```haskell
main :: IO ()
main =
    flip evalStateT initState $
    withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape
        modify $ updateState $ MkInputs
            { paddleUp    = keyDown ScancodeUp
            , paddleDown  = keyDown ScancodeDown
            }
        gets $ rasterizePattern . draw
```
High-level simulation

Everything we have so far is normal (and nice!) Haskell code – we can run it as such, using e.g. SDL2 for IO:
Everything we have so far is normal (and nice!) Haskell code – we can run it as such, using e.g. SDL2 for IO:

```haskell
videoParams = MkVideoParams
  { windowTitle = "Pong"
  , screenScale = 1
  , screenRefreshRate = 60
  , reportFPS = True
  }
```
High-level simulation

Everything we have so far is normal (and nice!) Haskell code – we can run it as such, using e.g. SDL2 for IO:
2. Hardware implementation
Field-Programmable Gate Arrays

FPGA is a collection of logic gates and registers where the connections are electronically configurable

* A chip fab on your desk

Physical connectors with appropriate electric properties to peripherals

Wide range of performance, price and built-in IO
FPGA is a collection of logic gates and registers where the connections are electronically configurable.

*A chip fab on your desk*

Physical connectors with appropriate electric properties to peripherals.

Wide range of performance, price and built-in IO.

Horrible toolchains taking Verilog or VHDL input.
Clash: Haskell to Hardware (FPGAs, ASICs)

- Signal :: Domain -> Type -> Type
- instance Applicative (Signal dom)
- register :: a -> Signal dom a -> Signal dom a

Register transfer-level model:

\[ y = \text{register } y_0 (\text{fun } \langle $ \rangle x \langle ** \rangle y) \]
Video output via VGA

- VGA: old analog video standard
- Sweet spot of theoretical simplicity and widespread support both by displays and FPGA development boards
- Three separate color channels
- Separate horizontal and vertical sync lines
Reusable VGA signal generator

Type-level tracking of resolution & pixel clock:

data VGADriver dom w h = VGADriver
  { vgaSync :: VGASync dom
    , vgaX    :: Signal dom (Maybe (Index w))
    , vgaY    :: Signal dom (Maybe (Index h))
  }

vgADriver
  :: (HiddenClockResetEnable dom, KnownNat w, KnownNat h)
  => (DomainPeriod dom ~ ps)
  => VGATimings ps w h
  -> VGADriver dom w h

vga640x480at60 :: VGATimings (HzToPeriod 25_175_000) 640 480
- Typed coordinate transformations: `center`, `scale`

```plaintext
vgax :: _ (Maybe (Index 640))
scale (SNat @2) . center $ vgaX :: _ (Maybe (Index 256))
```
-- Template Haskell macro to generate pixel clock
createDomain vSystem
  { vName="Dom25"
  , vPeriod = hzToPeriod 25_175_000
  }

topEntity :: "CLK_25MHZ" ::: Clock Dom25
            -> "RESET" ::: Reset Dom25
            -> "BTN_UP" ::: Signal Dom25 (Active High)
            -> "BTN_DOWN" ::: Signal Dom25 (Active High)
            -> "VGA" ::: VGAOut Dom25 8 8 8

topEntity = withEnableGen board
Pong circuit in Clash

board (fmap fromActive -> up) (fmap fromActive -> down) =
  vgaOut vgaSync rgb
where
  VGADriver{..} = vgaDriver vga640x480at60
  frameEnd = isFalling False (isJust <$> vgaY)

inputs = MkInputs <$> up <*> down

st = regEn initState frameEnd $ updateState <$> inputs <*> st

rgb = withBorder <$> pure (0,0,0) <*> (draw <$> st) <*> x <*> y
where
  (x, _) = scale (SNat @2) . center $ vgaX
  (y, _) = scale (SNat @2) . center $ vgaY
Low-level simulation

- *Broke*: predetermined inputs, batch processing of outputs
- *Woke*: interactive simulation
Low-level simulation

- **Broke**: predetermined inputs, batch processing of outputs
- **Woke**: interactive simulation

- `signalAutomaton` for periodwise simulation
- VGA signal interpreter
Low-level simulation

- **Broke**: predetermined inputs, batch processing of outputs
- **Woke**: interactive simulation (1/3 fps)
- **Bespoke**: good performance with Verilator (6 fps)

**Clashilator**: FFI to Verilator simulation
- Simulator UI (incl. VGA interpreter) still in Haskell
Hardware synthesis

- Clash → Verilog → FPGA bitstream
- Upload to FPGA using vendor tools
- Turnkey build system (clash-shake):
  - Papilio One (Xilinx Spartan 3)
  - Papilio Pro (Xilinx Spartan 6)
  - Nexys A7-50T (Xilinx Artix 7)
  - Coming soon: Arrow DECA (Intel MAX 10) by Dylan Thinnes

DEMO VIDEO: Pong running on real hardware
Hardware synthesis

- Clash → Verilog → FPGA bitstream
- Upload to FPGA using vendor tools
- Turnkey build system (clash-shake):
  - Papilio One (Xilinx Spartan 3)
  - Papilio Pro (Xilinx Spartan 6)
  - Nexys A7-50T (Xilinx Artix 7)
  - Coming soon: Arrow DECA (Intel MAX 10) by Dylan Thinnes
  - Contribute support for YOUR development board!

DEMO VIDEO: Pong running on real hardware
Hardware synthesis

- Clash $\rightarrow$ Verilog $\rightarrow$ FPGA bitstream
- Upload to FPGA using vendor tools
- Turnkey build system (clash-shake):
  - Papilio One (Xilinx Spartan 3)
  - Papilio Pro (Xilinx Spartan 6)
  - Nexys A7-50T (Xilinx Artix 7)
  - Coming soon: Arrow DECA (Intel MAX 10) by Dylan Thinnes
  - Contribute support for YOUR development board!

**DEMO VIDEO**: Pong running on real hardware
Retrocomputing with Clash: Haskell for FPGA Hardware Design

Using Haskell’s tools of abstraction to their fullest potential

Full implementation of various fun retrocomputing devices:

- Desktop calculator
- Pong
- Brainfuck as machine code
- CHIP-8
- Intel 8080 CPU
- Space Invaders arcade machine
- Compucolor II home computer

https://retrocla.sh/
Thank you!

Questions?

https://retrocla.sh/