

# Reversible & irreversible classical computation code

Jonathan Grattage

Code for reversible circuits, used in chapter 2, reversible classical computation, and the code for **FCC** morphisms, used in Chapter 4, **FCC**: Reversible & irreversible classical computation.

## 1 Import Statements

```
import Control.Monad
```

## 2 A datatype of reversible circuits

```
data Circ = Not
  | Wire [Int]
  | Par  Circ Circ
  | Seq  Circ Circ
  | Cond Circ Circ
  deriving (Show, Eq)
```

## 3 The *arity* function

```
arity      ∈ Circ → Maybe Int
arity (Not)    = Just 1
arity (Wire  $\vec{x}$ ) = do guard (chkPerm  $\vec{x}$ )
                  return (length  $\vec{x}$ )
arity (Cond  $x\ y$ ) = do  $m \leftarrow$  arity  $x$ 
                       $n \leftarrow$  arity  $y$ 
                      guard ( $m \equiv n$ )
                      return ( $1 + m$ )
arity (Par  $x\ y$ )  = do  $m \leftarrow$  arity  $x$ 
                       $n \leftarrow$  arity  $y$ 
                      return ( $m + n$ )
arity (Seq  $x\ y$ )  = do  $m \leftarrow$  arity  $x$ 
                       $n \leftarrow$  arity  $y$ 
                      guard ( $m \equiv n$ )
                      return ( $m$ )

chkPerm ∈ [Int] →  $\mathbb{N}_2$ 
chkPerm  $\vec{x}$  = (and [elem  $x\ \vec{x} \mid x \leftarrow [0..length\ \vec{x} - 1]$ ])
```

## 4 Example Circuits

```
cnotC ∈ Circ
cnotC = Cond Not (Wire [0])
toffoliC ∈ Circ
toffoliC = Cond cnotC (Wire [0,1])
```

```
fredkinC ∈ Circ
fredkinC = Cond (Wire [1,0]) (Wire [0,1])
```

## 5 Boolean vector and matrix types

```
type Vec = [N2]
type Mat = Vec → Vec
```

## 6 A compiler from circuits into matrices

```
comp          ∈ Circ → Maybe Mat
comp (Not)    = return (λv → case v of [True]  → [False]
                                       [False] → [True]
                                       -       → error "unreachable")

comp (Seq x y) = do m ← comp y
                  n ← comp x
                  return (m ∘ n)
comp (Wire p̄)  = return (λvs → [vs !! p | p ← p̄])
comp (Par x y) = do ax ← arity x
                  m  ← comp x
                  n  ← comp y
                  return (λvs → let (a, b) = splitAt ax vs
                                   in m a ++ n b)
comp (Cond x y) = do m ← comp x
                   n ← comp y
                   return (λ(v : vs) → if v then True : (m vs)
                                         else False : (n vs))
```

## 7 The circuit evaluator

```
eval ∈ Circ → Vec → Maybe Vec
eval c v = do a ← arity c
              guard (a ≡ length v)
              m ← comp c
              return (m v)
```

## 8 FCC implementation in Haskell

```
data FCC = FCC { a, b, h, g ∈ Int, φ ∈ Circ }
validFCC ∈ FCC → Maybe FCC
validFCC f = do let ah = Just (a f + h f)
                 bg = Just (b f + g f)
                 guard (arity (φ f) ≡ ah ∧ (ah ≡ bg))
                 return f
```

## 9 Classical circuit optimiser

```
rCircuit ∈ Circ → Circ
rCircuit x | rCirc x ≡ x = x
           | otherwise  = rCircuit (rCirc x)
```

$$\begin{aligned}
rCirc & \in Circ \rightarrow Circ \\
rCirc (Seq (Wire \vec{x}) (Wire \vec{y})) & = Wire (seqPerm \vec{x} \vec{y}) \\
rCirc (Par (Wire \vec{x}) (Wire \vec{y})) & = Wire (\vec{x} \# [y + length \vec{x} \mid y \leftarrow \vec{y}]) \\
rCirc (Seq x (Wire \vec{x})) & \begin{cases} \vec{x} \equiv [0..length \vec{x} - 1] = rCirc x \\ \text{otherwise} = Seq (rCirc x) (Wire \vec{x}) \end{cases} \\
rCirc (Seq (Wire \vec{x}) x) & \begin{cases} \vec{x} \equiv [0..length \vec{x} - 1] = rCirc x \\ \text{otherwise} = Seq (Wire \vec{x}) (rCirc x) \end{cases} \\
rCirc (Cond x y) & \begin{cases} x \equiv y = Par (Wire [0]) (rCirc x) \\ \text{otherwise} = Cond (rCirc x) (rCirc y) \end{cases} \\
rCirc (Seq (Par x y) (Par a b)) & \begin{cases} arity x \equiv arity a = rCirc (Par (Seq x a) (Seq y b)) \\ \text{otherwise} = (Seq (rCirc (Par x y)) (rCirc (Par a b))) \end{cases} \\
rCirc (Seq (Cond x y) (Cond a b)) & = rCirc (Cond (Seq x a) (Seq y b)) \\
rCirc (Par x (Par y z)) & = rCirc (Par (Par x y) z) \\
rCirc (Not) & = Not \\
rCirc (Seq x y) & = (Seq (rCirc x) (rCirc y)) \\
rCirc (Par x y) & = (Par (rCirc x) (rCirc y)) \\
rCirc (Wire \vec{x}) & = Wire \vec{x} \\
seqPerm & \in [Int] \rightarrow [Int] \rightarrow [Int] \\
seqPerm \vec{x} \vec{y} \mid \vec{x} \equiv [0..lx] = \vec{y} & \\
\mid \vec{y} \equiv [0..ly] = \vec{x} & \\
\mid \text{otherwise} = sortp \vec{x} \vec{y} & \\
\text{where } lx = (length \vec{x}) - 1 & \\
ly = (length \vec{y}) - 1 & \\
sortp & \in [Int] \rightarrow [Int] \rightarrow [Int] \\
sortp \vec{x} [y] & = [\vec{x} !! y] \\
sortp \vec{x} (y : \vec{y}) & = (\vec{x} !! y) : sortp \vec{x} \vec{y}
\end{aligned}$$