Embedding Classical and Quantum Circuits in Haskell

Amr Sabry

School of Informatics and Computing
Indiana University

July 26, 2012
Motivation

- Quantum mechanics is “the most accurate physical theory of our universe”.

- Use quantum mechanics to rethink classical computing

- Physical laws as algorithms

- Conservation of Information
The laws of physics are essentially algorithms for calculation. These algorithms are significant only to the extent that they are executable in our real physical world. Our usual laws of physics depend on the mathematician’s real number system. With that comes the presumption that given any accuracy requirement, there exist a number of calculations steps, which if executed, will satisfy that accuracy requirement. But the real world is unlikely to supply us with unlimited memory of unlimited Turing machine tapes, Therefore, continuum mathematics is not executable, and physical laws which invoke that can not really be satisfactory. They are references to illusionary procedures.

Physical Laws as Algorithms

The laws of physics are essentially algorithms for calculation. These algorithms are significant only to the extent that they are executable in our real physical world. Our usual laws of physics depend on the mathematician’s real number system. With that comes the presumption that given any accuracy requirement, there exist a number of calculations steps, which if executed, will satisfy that accuracy requirement. But the real world is unlikely to supply us with unlimited memory of unlimited Turing machine tapes, Therefore, continuum mathematics is not executable, and physical laws which invoke that can not really be satisfactory. They are references to illusionary procedures.


Use Computer Science to understand Physics
Ed Fredkin pursued the idea that information must be finite in density. One day, he announced that things must be even more simple than that. He said that he was going to assume that information itself is conserved. “You’re out of your mind, Ed.” I pronounced. “That’s completely ridiculous. Nothing could happen in such a world. There couldn’t even be logical gates. No decisions could ever be made.” But when Fredkin gets one of his ideas, he’s quite immune to objections like that; indeed, they fuel him with energy. Soon he went on to assume that information processing must also be reversible — and invented what’s now called the Fredkin gate.

Ed Fredkin pursued the idea that information must be finite in density. One day, he announced that things must be even more simple than that. He said that he was going to assume that information itself is conserved. “You’re out of your mind, Ed.” I pronounced. “That’s completely ridiculous. Nothing could happen in such a world. There couldn’t even be logical gates. No decisions could ever be made.” But when Fredkin gets one of his ideas, he’s quite immune to objections like that; indeed, they fuel him with energy. Soon he went on to assume that information processing must also be reversible — and invented what’s now called the Fredkin gate.


Use Physics to understand Computer Science
Applications

Having a model of computation that is in harmony with the laws of physics would have many applications:

- Entropy analysis of circuits
- Biochemical models of computations
- Information-flow security
- Differential privacy
- Massive concurrency
Strategy and outline of this talk

- Will only present some basic ideas showing how to bring the worlds of traditional computing and physics close to each other.

- Will use the programming language Haskell to distill a model of both reversible classical circuits and quantum circuits.

- The model is executable; it highlights the similarities and differences between classical and quantum computing.

- Part of a much larger project that we can discuss offline: several students; several publications (RC 2011, RC 2012, POPL 2012); NSF grant.
- Strongly-typed functional language

- Rich type system

- Functions are **first-class** values, i.e., can be passed as arguments and returned as results
Basic Haskell types

- **Base types:** \texttt{Bool}, \texttt{Int}, \texttt{Char}, \texttt{String}, etc.

- **Pair types:** \((a, b)\)

- **(Disjoint) union types:** \texttt{Either} \(a\) \(b\)

- **Function types:** \(a \rightarrow b\)

- **List types:** \([a]\)
Basic Haskell types and terms

- **Base types:**

  - `True` :: Bool
  - `False` :: Bool
  - `3` :: Int
  - `'c'` :: Char
  - "hello" :: String

- **Pair types:**

  - `(3, True) :: (Int, Bool)`

- **(Disjoint) union types:**

  - `Left 3 :: Either Int Bool`
  - `Right False :: Either Int Bool`

- **Function types:**

  - `\x -> x+3 :: Int -> Int`
  - `\x -> if x then 3 else 4 :: Bool -> Int`

- **List types:**

  - `[3, 4, 5, 6, 7] :: [Int]`
Example Programs

- All right triangles with sides less than or equal to 10 and whose perimeter is 24:
  \[
  \begin{aligned}
  \{ (a,b,c) | \\
  c &\in [1..10], \\
  b &\in [1..c], \\
  a &\in [1..b], \\
  a^2 + b^2 &= c^2, \\
  a+b+c &= 24
  \end{aligned}
  \]

- All prime numbers up to \( m \):

\[
\text{primesTo } m = 2 : \text{sieve } [3,5..m] \quad \text{where}
\]
\[
\begin{aligned}
\text{sieve } [] &= [] \\
\text{sieve } (p:xs) &= p : \text{sieve } (xs \setminus [p,p+2*p..m])
\end{aligned}
\]
A proof of $P$ in propositional logic corresponds to a Haskell program of type $P$. 

Philosophy
Propositional Logic

- Propositions:

\[ P ::= A \mid P \land P \mid P \lor P \mid P \rightarrow P \mid P \leftrightarrow P \]

- Proofs can be represented in many ways, e.g., via truth tables:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ∨ B</th>
<th>A ∨ B</th>
<th>A → B</th>
<th>A ↔ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Curry-Howard Isomorphism

- Each proposition corresponds to a Haskell type:
  - $P \land Q$ corresponds to the pair type $(a, b)$
  - $P \lor Q$ corresponds to the disjoint union type `Either a b`
  - $P \rightarrow Q$ corresponds to the function type $a \rightarrow b$

- If a proposition is true, then there is a Haskell term of the corresponding type;

- If a proposition is false, then there is no Haskell term of the corresponding type;

- Programs are proofs
Programs are proofs

A program of type:

- \((a, b)\) is a pair of proofs; one that proves \(a\) and one that proves \(b\)

- Either \(a\) or \(b\) is either a proof of \(a\) or a proof of \(b\); we distinguish the two cases by annotating the first proof with \(\text{Left}\) and the second proof with \(\text{Right}\);

- \(a \rightarrow b\) is a procedure that converts proofs of \(a\) to proofs of \(b\)
Examples

- \(((A \land B) \rightarrow C) \rightarrow (A \rightarrow (B \rightarrow C))\) is true. Here is a Haskell term of the required type:

  \[
p :: ((a,b) -> c) -> (a -> (b -> c))
p f = \lambda a -> \lambda b -> f (a,b)
-- could be written
-- p f a b = f (a,b)
\]

- \(A \rightarrow B\) is in general not true; there is no Haskell term of type \(a \rightarrow b\) for all \(a\) and \(b\)
Haskell as a host language for
domain-specific embedded languages
Embedding Languages

Just like we can define customized logics to reason about a particular domain, we can also define customized sublanguages in Haskell to model a particular domain.

Basic idea:

- represent the data in the new domain using special Haskell types;
- represent the operations in the new domain using operations on the special types;
- restrict the operations on the special types to be only the ones provided above
Key Technologies

- Type Classes
- Monads
Type Classes
Type Classes

- Group types and operations (similar to classes in OO languages).

- A type \( a \) together with a constant \( \text{unit} :: a \) and a binary operation \( \text{binop} :: a \rightarrow a \rightarrow a \):

  ```haskell
  class Monoid a where
  unit :: a
  binop :: a -> a -> a
  ```

- Given a list of elements of type \( a \) we can fold the list starting with \( \text{unit} \) and applying \( \text{binop} \):

  ```haskell
  accList :: Monoid a => [a] -> a
  accList [] = unit
  accList (x:xs) = x `binop` accList xs
  ```
Instances of Type Classes

- The type \texttt{Int} can be made an instance of the \texttt{Monoid} class with 1 as the \texttt{unit} and multiplication as the \texttt{binop}:

  ```haskell
  instance Monoid Int where
  unit = 1
  binop = (*)
  ```

- The type \texttt{String} can be made an instance of the \texttt{Monoid} class with the empty string as the \texttt{unit} and concatenation as the \texttt{binop}:

  ```haskell
  instance Monoid String where
  unit = ""
  binop = (++)
  ```

- The type of \texttt{Int -> Int} functions can be made an instance of the \texttt{Monoid} class with the identity function as the \texttt{unit} and composition as the \texttt{binop}:

  ```haskell
  data Fun = Fun (Int -> Int)
  ap (Fun f) a = f a

  instance Monoid Fun where
  unit = Fun (\x -> x)
  binop (Fun f) (Fun g) = Fun (f . g)
  ```
The same function `accList` can be used at different types to fold over lists of different kinds of elements:

```haskell
testInt :: Int
testInt = accList [1..10]

testString :: String
testString = accList ["abc","def","gh"]

testFun :: Int
testFun = ap (accList [Fun succ, Fun fact]) 3
    where fact n = accList [1..n]
```
Monads
Define a **new layer** of computations that is distinct from the functional layer;

**return** allows you to move to the monad layer;

**bind** or **>>=** or **do** allows you to sequence computations in the monad layer;

generally impossible to move back from the monad layer to the functional layer
Monads

Just a type class:

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Special syntax: instead of writing:

```
e1 >>= \x1 -> e2 >>= \x2 -> return e
```

we can write:

```
do x1 <- e1
   x2 <- e2
   return e
```
Find all the triples of numbers \([1..7]\) that add up to 15:

```haskell
findNums :: [(Int, Int, Int)]
findNums = do
    a <- [1..7] -- generate choices
    b <- [1..7]
    c <- [1..7]
    if (a+b+c == 15)
        then return (a, b, c) -- success
        else [] -- backtrack
```
Monads: State

`nodups` is a pure function; internally it uses a hashtable and performs allocation, reading, and writing of pointers. All these effects are tracked and encapsulated by the type system.

```haskell
nodups :: [String] -> [String]
nodups ss = runST (do ht <- HT.new
               loop ht ss)

where
  loop :: HT.HashTable s String () -> [String] -> ST s [String]
  loop ht [] = return []
  loop ht (s:ss) = do seen <- HT.lookup ht s
                     case seen of
                       Nothing -> do HT.insert ht s ()
                                     rss <- loop ht ss
                             return (s : rss)
                       Just _ -> loop ht ss
```

Amr Sabry (School of Informatics and Computing Indiana University)
Embedding Classical and Quantum Circuits in Haskell
July 26, 2012 27 / 41
Classical Circuits
Data and wires

The type of classical values:

\[
\textbf{data} \ \text{Classical} \ a = \text{CVal} \ a
\]

An abstract type of wires:

\[
\textbf{class Eq} \ a \Rightarrow \text{Wire} \ a \ \textbf{where}
\]
\[
\text{basis} :: [a]
\]

\[
\textbf{instance} \ \text{Wire} \ \text{Bool} \ \textbf{where}
\]
\[
\text{basis} = [\text{False}, \text{True}]
\]

\[
\textbf{instance} \ (\text{Wire} \ a, \text{Wire} \ b) \Rightarrow \text{Wire} \ (a, b) \ \textbf{where}
\]
\[
\text{basis} = [(a, b) | a <- \text{basis}, b <- \text{basis}]
\]

\[
\textbf{instance} \ (\text{Wire} \ a, \text{Wire} \ b, \text{Wire} \ c) \Rightarrow \text{Wire} \ (a, b, c) \ \textbf{where}
\]
\[
\text{basis} = [(a, b, c) | a <- \text{basis}, b <- \text{basis}, c <- \text{basis}]
\]
Parallel and Sequential Composition of Circuits

-- parallel composition

class Parallel m where
    (<>*) :: m a -> m b -> m (a,b)

instance Parallel Classical where
    (CVal b1) <*> (CVal b2) = CVal (b1,b2)

-- sequential composition

class Sequential m where
    vreturn :: Wire a => a -> m a
    (@>>=) :: (Wire a, Wire b) => m a -> (a -> m b) -> m b

instance Sequential Classical where
    vreturn b = CVal b
    (CVal b) @>>= f = f b
Classical Circuits

A abstract hardware circuit embedded in a Haskell type:

```haskell
data Circuit m a where

  CPrm :: Wire a => Int -> (a -> m a) -> Circuit m a

  Cseq :: Wire a => Circuit m a -> Circuit m a -> Circuit m a

  Cpar :: (Wire a, Wire b) =>
           Circuit m a -> Circuit m b -> Circuit m (a, b)

  Cmux :: Wire a => Circuit m a -> Circuit m a -> Circuit m (Bool, a)
```

The circuit description is parametrized over the type of wires `Wire` and the special monad `m` that specifies how the circuits are composed in sequence and in parallel.
Classical Circuits: Examples

class (Parallel m, Sequential m) => CMonad m
instance CMonad Classical

notPrm :: CMonad m => Bool -> m Bool
notPrm b = vreturn (not b)

idPrm :: (CMonad m, Wire a) => a -> m a
idPrm a = vreturn a

xorC :: CMonad m => Circuit m (Bool,Bool)
xorC = Cmux (CPrm 1 notPrm) (CPrm 1 idPrm)

toffoliC :: CMonad m => Circuit m (Bool,(Bool,Bool))
toffoliC = Cmux xorC (CPrm 2 idPrm)
Reversible Circuits

- Toffoli gate (or Fredkin gate) are universal

- \textbf{and, or, nand, nor, etc.} can be simulated

- Simulation idea: use reversible circuit; fix some inputs to constants; and ignore some outputs
meaningC :: (CMonad m, Wire a) => Circuit m a -> a -> m a
meaningC (CPrm _ f) a = f a
meaningC (Cseq c1 c2) a = meaningC c1 a >>= \b ->
    meaningC c2 b
meaningC (Cpar c1 c2) (a,b) = meaningC c1 a <*> meaningC c2 b
meaningC (Cmux c1 c2) (b,bs) = if b
    then vreturn True <*> meaningC c1 bs
    else vreturn False <*> meaningC c2 bs
Example

test1 :: Bool -> Bool -> Bool -> Classical (Bool, (Bool, Bool))
test1 b1 b2 b3 = meaningC toffoliC (b1, (b2, b3))
Quantum Circuits
type PA = Complex Double

type Vec a = a -> PA

data Quantum a = QVal (Vec a)
untag (QVal f) = f

instance Parallel Quantum where
  (QVal v1) <*> (QVal v2) = QVal (\ (a1,a2) -> v1 a1 * v2 a2)

instance Sequential Quantum where
  vreturn b = QVal (\a -> if a==b then 1 else 0)
  (QVal v) @>>= f =
    QVal (\b -> sum [ v a * untag (f a) b | a <- basis ])

instance CMonad Quantum

Amr Sabry (School of Informatics and Computing Indiana University)
A quantum computation over booleans manipulates values which are vectors from booleans to complex probability amplitudes.

Quantum states are unit vectors (and identified up to global phase)

All the differences between classical and quantum computation are abstracted in the choice of the type parameter $m$
Examples

test2 = QVal v >>= meaningC toffoliC
    where v (False, (False, False)) = 1
          v _ = 0

test3 = QVal v >>= meaningC toffoliC
    where v (False, (False, False)) = 1
          v (True, (True, True)) = 1
          v _ = 0
Conclusions

• Haskell is the language of choice of much interesting research. Check recent editions of the Haskell Symposium and International Conference on Functional Programming (ICFP)

• Quantum computing, at its core, is not much different from classical computing with (special) computational effects;

• Ideas can be further developed to design “quantum programming languages”
More

Visit my web page:

http://www.cs.indiana.edu/~sabry