Strongly Typed Domain Specific Embedded Languages

Lennart Augustsson

Standard Chartered Bank

lennart@augustsson.net

Overview

- Mostly Haskell
  - Types, types, types
- A sampling of DSELs I've made
  - LLVM bindings
  - Paradise, Excel generation
  - Bluespec, hardware design

Who am I?

- Languages over the years
  - 1990-1995, hbc – the first Haskell compiler
  - 1995-1996, R@VE – a DSL for airline crew scheduling
  - 1997-1998, Delf – a DSL for (Swedish) tax calculation
  - 2000-2005, Bluespec – a DSL for hardware design
  - 2006-2008, Paradise – a DSEL for pricing models
  - 2008-, more DSELs

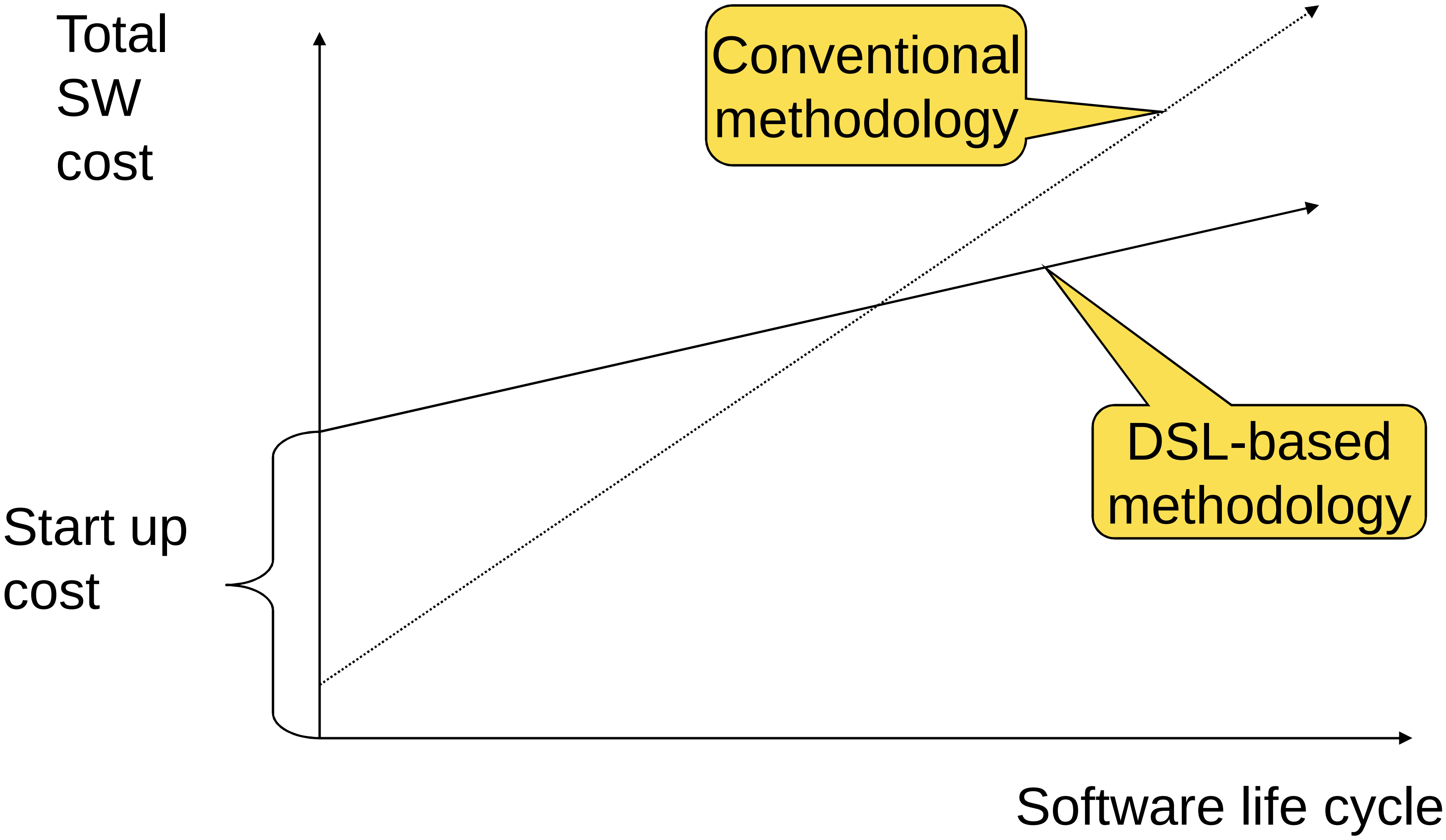
What is a Domain Specific Language?

A programming language tailored for a particular application domain, which captures precisely the semantics of the application domain -- no more, no less.

> *A DSL allows one to develop software for a particular application domain quickly, and effectively, yielding programs that are easy to understand, reason about, and maintain.*

Hudak

The Cost Argument



The Problem with DSLs

•DSLs tend to grow: adding procedures, modules, data structures…

•Language design is *difficult* and *time-consuming*; large parts are not domain specific.

•Implementing a compiler is *costly* (code-generation, optimisation, type-checking, error messages…)

Start up costs may be substantial!

Domain Specific *Embedded* Languages

Why not *embed* the DSL as a library in an existing *host* language?

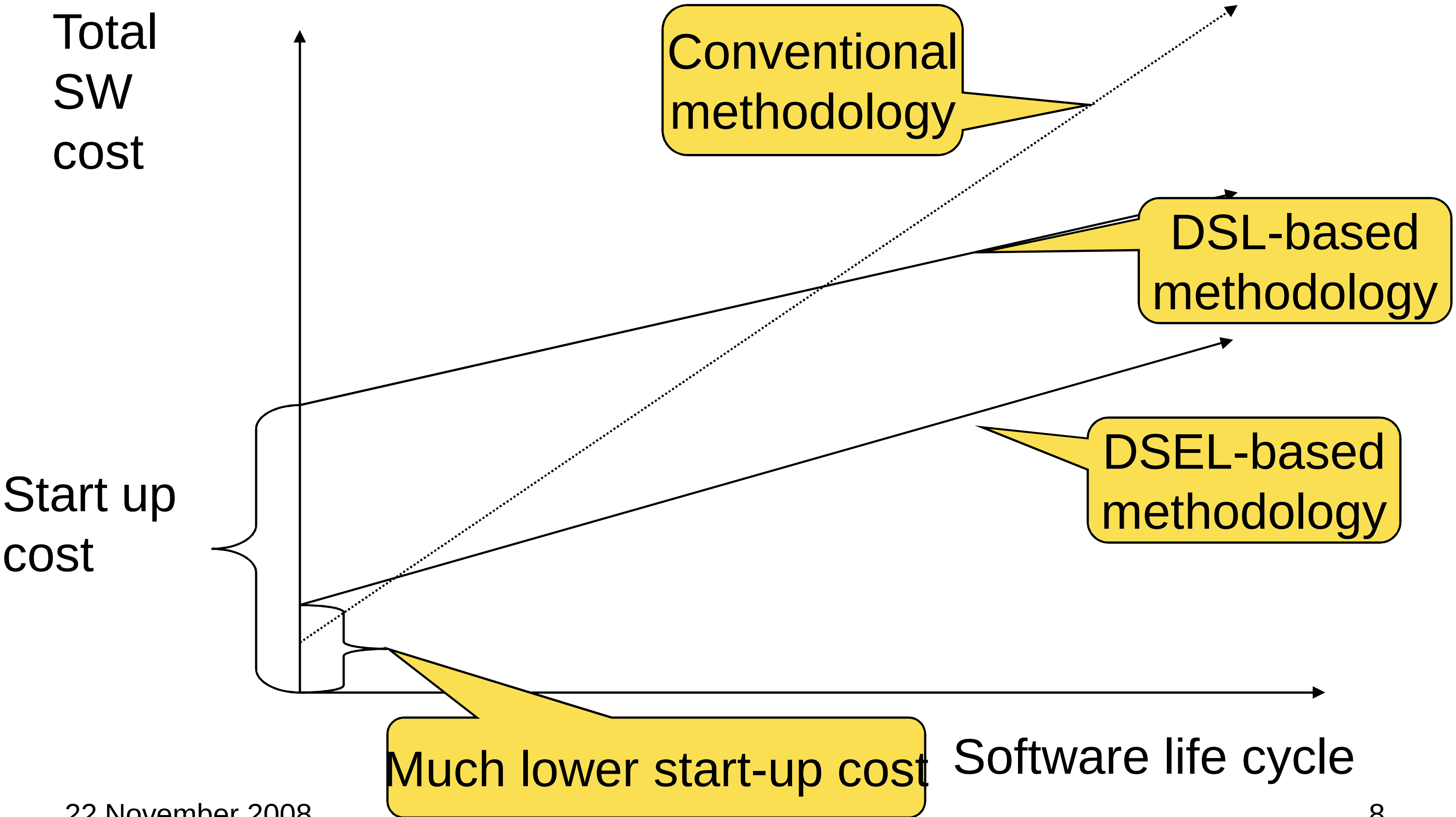
+

- Inherit non-domain-specific parts of the design.
- Inherit compilers and tools.
- Uniform "look and feel" across many DSLs
- DSLs integrated with full programming language, and with each other.

-

- Constrained by host language (syntax, type system, etc).
- Error messages.

The Cost Argument Again



What makes a good host language?

- Light weight syntax
  - Because we want to tailor the syntax
  - Haskell, Lisp, Ruby, Python, Smalltalk, Scala, …
- Easy to create suspensions
  - Because we want to make control structures
  - Haskell, Lisp, Ruby, Smalltalk, Scala, …
- Powerful and malleable type system
  - Haskell, Scala, …

Why strong typing?

- Helps in designing software.
- Eliminates a lot of testing.
- More efficient.
- Easier to refactor.

DS*E*L

- There are two kinds of embeddings:
  - Shallow embedding, the DSEL uses the values and types of the host language.
  - Deep embedding, the DSEL builds an abstract syntax tree, using its own types.

Shallow/deep embedding

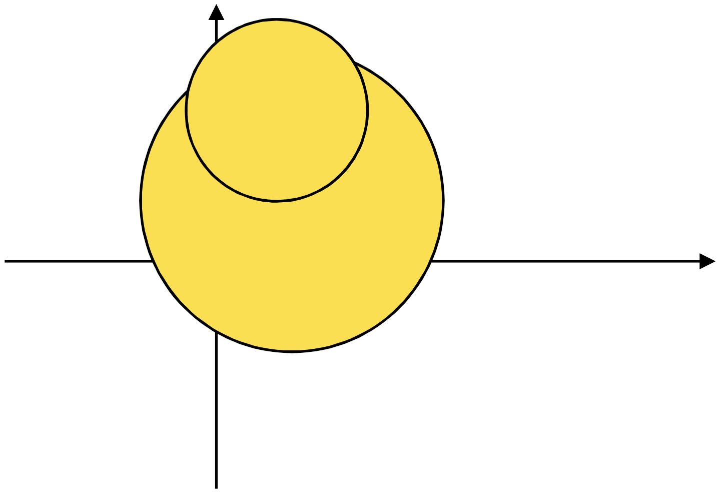
- A language for drawing circles

```
twoCircles = do
    circle (2, 2) 4
    circle (1.5, 4) 2.5
```

- Draws two circles at the given coordinates and with the given radius.

Shallow embedding

- Running the program draws the circles



- Type of the **`circle`** function

```
circle :: (Double, Double) -> Double -> IO ()
```

- Uses ordinary Haskell types

Deep embedding

- Running the program generates an abstract syntax tree:

```
Stmts [Circle (Dbl 2, Dbl 2) (Dbl 4),
       Circle (Dbl 1.5, Dbl 4) (Dbl 2.5)]
```

- Type of the **`circle`** function

```
circle :: (Expr Double, Expr Double) ->
              Expr Double -> Stmt ()
```

- Uses “embedded” types (GADTs or phantom types)
- Allows further processing of the program.

Shallow/deep embedding

- Shallow embedding is easier
- Deep embedding allows more processing
- Deep embedding is trickier to make strongly typed.
- The example program, twoCircles, has no notion of what embedding it is.
- In fact, in can be both!

```
twoCircles :: (CircleMonad m) => m ()
```

The Haskell type system (not a Haskell tutorial)

- Base types
  - **`Int, Int8, Int16, …`**
  - **`Word, Word8, Word16, …`**
  - **`Integer`**
  - **`Char`**
  - **`Float, Double`**
- Function type
  - **`S -> T`**

The Haskell type system (not a Haskell tutorial)

- Data types
  - Enumerations
    - **`data Color = Red | Green | Blue`**
  - Records
    - ```
      data Coord =
          Coord { x :: Double, y :: Double }
      ```
  - Unions
    - ```
      data Shape = Circle { radius :: Int }
                 | Rect { width,height :: Int }
      ```

The Haskell type system (not a Haskell tutorial)

- Data types
  - Recursive types
    - **`data ListOfInt = Nil | Cons Int ListOfInt`**
  - Parameterized types
    - ```
      data BinTree a = Empty
         | Node { left, right :: BinTree a,
                      value :: a }
      ```
  - List
    - **`[a]`**
  - Tuples
    - **`(a,b), (a,b,c), (a,b,c,d), …`**
      ```

The Haskell type system (not a Haskell tutorial)

- Type variables
  - Used to express parametric polymorphism
  - ```
    swap :: (a, b) -> (b, a)
    swap (x, y) = (y, x)
    ```
  - ```
    id :: a -> a
    id x = x
    ```
  - `length :: [a] -> Int`
  - `map :: (a -> b) -> [a] -> [b]`

The Haskell type system (not a Haskell tutorial)

- Type classes
  - What is the type of == ?
    - Almost any two values of the same type can be compared.
  - What is the type of + ?
    - Types like Int and Double can be added.
  - Why not traditional overloading?
    - Type inference, e.g., **`refl x = x == x`**
- Haskell type classes are collections of types
  - I.e., more like OO interfaces than classes.

The Haskell type system (not a Haskell tutorial)

- == again
  - `(==) :: a -> a -> Bool`
    - WRONG! All values cannot be compared.
  - `(==) :: (Eq a) => a -> a -> Bool`

A context.
Constrains a.

The Haskell type system (not a Haskell tutorial)

- Declaring Eq

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  (==) = primIntEqual

instance Eq Double where
  (==) = primDoubleEqual
```

The Haskell type system (not a Haskell tutorial)

- More Eq

```
instance (Eq a, Eq b) => Eq (a, b) where
  (x,y)==(z,w) = x==z && y==w

instance (Eq a) => Eq [a] where
  []     == []       =  True
  (x:xs) == (y:ys)   =  x==y && xs==ys
  _      == _        =  False
```

The Haskell type system (not a Haskell tutorial)

- What about + ?

Whoa! Overloaded on the return type.

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  fromInteger :: Integer -> a

instance Num Double where
  (+) = primDoubleAdd
  (-) = primDoubleSub
  (*) = primDoubleMul
  fromInteger = primDoubleFromInteger
```

The Haskell type system (not a Haskell tutorial)

- What about numeric literals?
  - Writing, e.g., 42 in Haskell really means **`(fromInteger 42)`**
  - Allows each type to treat literals the way it likes.
  - Arbitrary precision for the literal.
  - Great for DSEL! Can use numeric literals for new numeric types.
  - ```
    inc :: (Num a) => a -> a
    inc x = x + 1
    ```
    ```

A small example

```
-- Solving a quadratic equation,
-- i.e. a*x^2 + b*x + c = 0
solve (a, b, c) = ((-b+a)/(2*r), (-b-r)/(2*a))
  where r = sqrt(b^2 – 4*a*c)
```

```
solve :: (Floating a) => (a, a, a) -> (a, a)
```

Case Study, LLVM

- LLVM (Low Level Virtual Machine) is an assembly language (in SSA form).
- Programming language bindings allow code to be generated by a batch compiler or a JIT.
- LLVM API is a large set of procedures to create instructions, basic blocks, etc.
- Bindings exist for, e.g., C++, O'Caml, Haskell

Case Study, LLVM

- Text file syntax:

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}

/* Corresponding C code */
int mul_add(int x, int y, int z) {
  return x * y + z;
}
```

Case Study, LLVM

- In C++:

```
Constant* c = mod->getOrInsertFunction("mul_add",
/*ret type*/                                       IntegerType::get(32),
/*args*/                                           IntegerType::get(32),
                                                   IntegerType::get(32),
                                                   IntegerType::get(32),
                                                   NULL);
Function* mul_add = cast<Function>(c);
Function::arg_iterator args = mul_add->arg_begin();
Value* x = args++;
Value* y = args++;
Value* z = args++;
BasicBlock* block = BasicBlock::Create("entry", mul_add);
IRBuilder builder(block);
Value* tmp = builder.CreateBinOp(Instruction::Mul,
                                 x, y, "tmp");
Value* tmp2 = builder.CreateBinOp(Instruction::Add,
                                  tmp, z, "tmp2");
builder.CreateRet(tmp2);
```

Case Study, LLVM

- In Haskell:

```
mul_add :: CodeGen (Int32 -> Int32 -> Int32 ->
                    IO Int32)
mul_add = createFunction $ \ x y z -> do
      createBasicBlock
      tmp  <- mul x y
      tmp2 <- add tmp z
      ret tmp2
```

Case Study, LLVM

- So what about types?
- LLVM has a rich type system
  - *integer*: i1, …, i8, …, i16, … i32, …
  - *floating*: float, double, …
  - *first class*: *integer*, *floating*, *pointer*, *array*, …
  - *primitive*: label, void, *floating*
  - *derived*: *integer*, *array*, *function*, *pointer*, …
  - *array*: [<# elements> x <elementtype>]
  - *function*: <returntype>(<parameter list>)
  - …

Case Study, LLVM

- Samples instructions:
  - **`ret void`**

    `ret <type> <value>`
    - <type> must be *first class*
  - <result> = **`add`** <ty> <op1>, <op2>
    - Arguments must be *integer*, *floating*, or *vector*
  - <result> = **`xor`** <ty> <op1>, <op2>
    - Arguments must be *integer* or *vector*
  - <result> = **`call`** <ty> <fnptrval>(<args>)
    - Args must be *first class*, function must match args

Case Study, LLVM

- The C++ code enforces very few of the type restrictions.
- What happens if we make a type error?
  - Caught by a runtime sanity check, exception thrown.
  - Uncaught, segmentation fault or just a wrong answer.

Case Study, LLVM

- Introduce type classes for LLVM types

```
class IsType a where
    typeRef :: a -> TypeRef
class (IsType a) => IsArithmetic a
class (IsArithmetic a) => IsInteger a
class (IsArithmetic a) => IsFloating a
class (IsType a) => IsPrimitive a
class (IsType a) => IsFirstClass a
class (IsType a) => IsFunction a
```

Case Study, LLVM

- Put corresponding Haskell types in classes

```
instance IsType Double where typeRef _ = doubleType
instance IsType ()     where typeRef _ = voidType
instance IsType Bool   where typeRef _ = int1Type
instance IsType Int8   where typeRef _ = int8Type
instance IsType Int16  where typeRef _ = int16Type
instance IsType Int32  where typeRef _ = int32Type
…
instance (IsType a) => IsType (Ptr a) where
    typeRef ~(Ptr a) = pointerType (typeRef a)

instance (IsFirstClass a, IsFunction b) =>
         IsType (a->b) where …
```

Case Study, LLVM

- Put corresponding Haskell types in classes

```
instance IsArithmetic Double
instance IsArithmetic Int32
…
instance IsFloating Double
…
instance IsInteger Int32
…
```

- And a few more pages of this

Case Study, LLVM

- Instructions functions simply call the (type unsafe) LLVM functions via FFI.
- Some instruction types

```
add  :: (IsArithmetic a) => a -> a -> CodeGen r a
xor  :: (IsInteger a)    => a -> a -> CodeGen r a
ret  :: (IsFirstClass a) => r -> CodeGen r ()
call :: (CallArgs f g)   => Function f -> g
```

Case Study, LLVM

- Conclusions
  - Haskell makes it possible to make a strongly typed interface to external libraries.
  - Complex types and relationships can be encoded with type classes.

Case Study, Excel

- Paradise, a DSEL for generating Excel
- Why?
  - Excel is terrible for software reuse.
  - Copy & paste only “abstraction” mechanism
  - But Excel is a familiar UI; people like it
  - So don’t write Excel, generate it
- Actually two DSELs
  - Computation
  - Layout

Case Study, Excel

- Example: two inputs, output the sum
- Computation

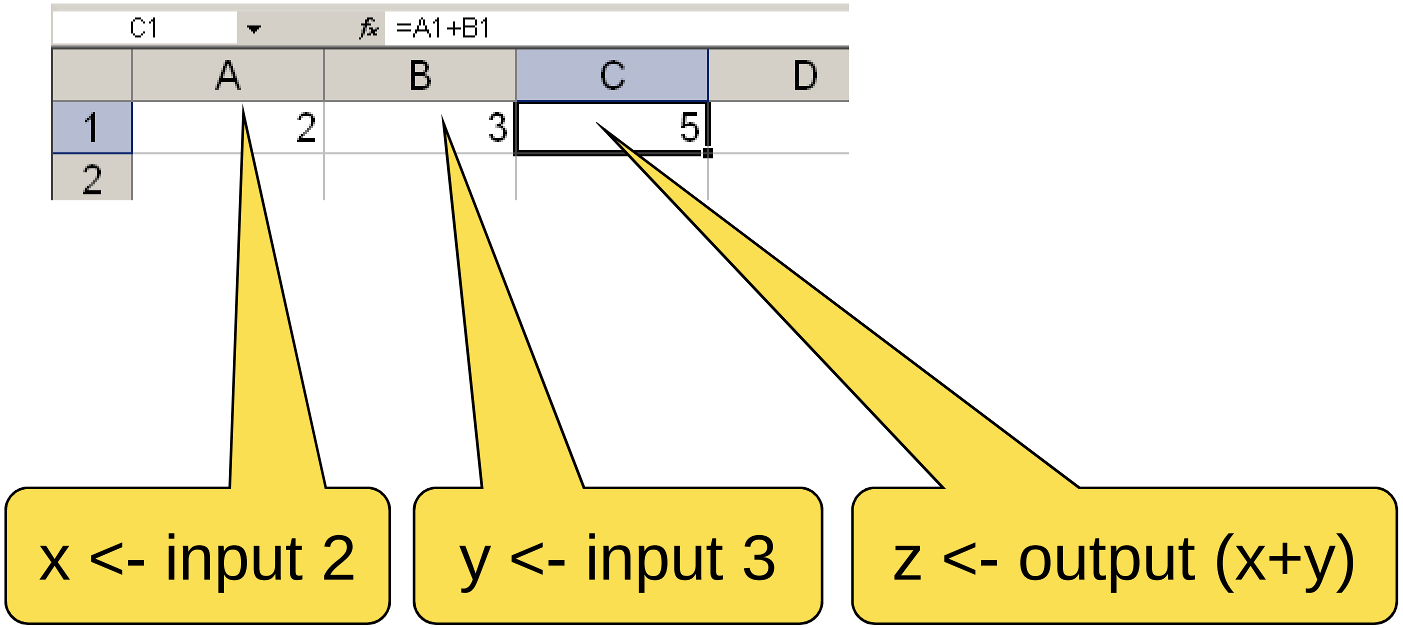
```
example = do
  x <- input 2
  y <- input 3
  z <- output (x+y)
```

- Layout

```
  return (row [view x, view y, view z])
```

Case Study, Excel

- Running this Haskell code *generates* an Excel sheet



Case Study, Excel

- Excel is dynamically typed, few types:
  - **`double, string, bool (+ errors)`**
  - Many serious Excel users have additional types representing objects (via Excel addins), but encoded as, e.g., strings.

Case study, Excel

- Deep embedding
  - Need AST
  - Running the DSEL code generates a spreadsheet.
- We need an AST for Excel

Case study, Excel

- Type of Excel expressions

```
data Exp =
       LitDbl Double
     | LitStr String
     | LitBol Bool
     | Apply Func [Exp]
     | Var Id
type Func = String
type Id = String
```

- But this is not type safe!
  - E.g., `Apply “not” [LitDbl 1.2]`

Case study, Excel

- Trick, use “phantom types”.
- I.e., create a well typed wrapper, and only expose this to the user.

```
data E a = E Exp
```

Type variable not mention on the right. It’s a phantom.

Case study, Excel

- Make a numeric instance.

```
instance Num (E Double) where
    E x + E y = E (Apply "+" [x, y])
    E x – E y = E (Apply "-" [x, y])
    E x * E y = E (Apply "*" [x, y])
    fromInteger i = E (LitDbl (fromInteger i))
```

- So now

```
1 + 2 * 3 :: E Double
```

  is

```
E (Apply "+" [LitDbl 1.0,
              Apply "*" [LitDbl 2.0,
                         LitDbl 3.0]])
```

Case study, Excel

- Types for **`input`** and **`output`**

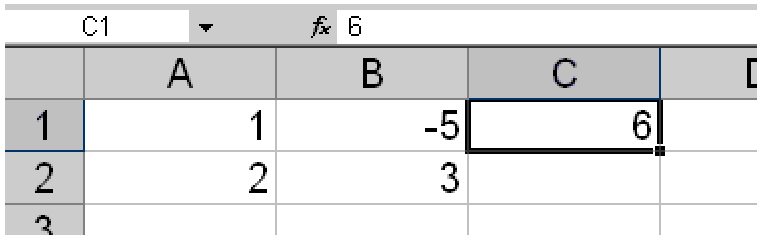
Monad to handle cell identity

```
class Cell a where …
instance Cell Double where …
instance Cell String where …
instance Cell Bool where …

input  :: (Cell a) => E a -> Gen (E a)
output :: (Cell a) => E a -> Gen (E a)

instance (Cell a,Cell b) => Cell (a,b)
instance (Cell a,Cell b,Cell c) => Cell (a,b,c)
```

Case study, Excel

- A little reuse, solving quadratics in Excel

```
solve = do
   abc <- input (1, 0, 0)
   rs  <- output (solve abc)
   return (column [view abc, view rs])
```



- Note, the same code (even compiled!) for **`solve`** will work in the Excel code.

Case study, Excel

- Conclusions
  - Type classes are useful to encode various restrictions.
  - An unityped deep embedding can be made type safe with phantom types.

Case study, Bluespec

- Bluespec is a hardware design language
  – www.bluespec.com
- Bluespec is a DSL
- The main features of Bluespec can be done as a DSEL in Haskell
  – In fact Atom is a DSEL similar to Bluespec
- In hardware bits are important
  – Need to know the number of bits a value needs when stored.

Case study, Bluespec

- A snippet of code
  - Defines two registers to hold values.
  - Defines a rule that produces some combinational logic that executes when applicable.

```
stupidAdder = do
    x <- mkReg (42 :: Int8)
    y <- mkReg (12 :: Int8)
    rule (x > 0) $ do
        x <== x – 1
        y <== y + 1
```

Case study, Bluespec

- What can we store in a register?
  Anything that can be turned into a fixed number of bits.
- Here is how we can express this with Haskell types:

```
class Bits a where
    type Size a
    toBits   :: a -> Bit (Size a)
    fromBits :: Bit (Size a) -> a
```

Case study, Bluespec

- We need to express sizes in types, as to make bit width statically typed.
- Haskell does not have a notion of numbers on the type level, we have to build it.
- For simplicity, we use unary encoding of numbers.

```
data Zero
data Succ n

type One = Succ Zero
type Two = Succ One
```

Case study, Bluespec

- We want be able to convert from the type level to the value level.

```
class Nat a where
    toValue :: a -> Int
instance Nat Zero where
    toValue _ =
instance (Nat n) => Nat (Succ n) where
    toValue _ = 1 + toValue (undefined :: n)

-- typical use
  … toValue (undefined :: T) …
```

Case study, Bluespec

- Type level addition.
  - Yes, the syntax is weird.

```
type family Add m n
type instance Add Zero n = n
type instance Add (Succ m) n = Succ (Add m n)
```

Case study, Bluespec

- Primitive type of bit vectors

```
data Bit

append :: Bit m -> Bit n -> Bit (Add m n)
split :: Bit (Add m n) -> (Bit m, Bit n)

toInt :: Bit n -> Integer
fromInt :: Integer -> Bit n
```

Case study, Bluespec

- Some instances

```
class Bits a where
    type Size a
    toBits   :: a -> Bit (Size a)
    fromBits :: Bit (Size a) -> a

instance Bits Bool where
    type Size Bool = One
    toBits x = fromInt (if b then 1 else 0)
    fromBits b = toInt b == 1
```

Case study, Bluespec

- Some instances, cont

```
instance (Bits a, Bits b) => Bits (a, b) where
   type Size (a, b) = Add (Size a) (Size b)

   toBits (x, y) = append (toBits x) (toBits y)

   fromBits b = (fromBits bx, fromBits by)
      where (bx, by) = split b
```

Case study, Bluespec

- Conclusions
  - Complicated concepts like numbers and addition can be encoded at the type level.

Conclusions

- DSELs are great.
- Strongly typed DSELs are even greater.
- Haskell types can encode very complex type systems.

Questions?