Easy Dependently Typed Programming

A by Andor Pénzes

What are Dependent Types?

1 Expressive 🤖

- We can use dependent types as glorified assertion.
- Types are artificial constructs!
- Values are things that we can be consumed by functions, or created by functions.
- There is no distinction between types and values same as above applies for types.

2 Powerful 🔨

You can define precise constraints and encode invariants that ensure consistency.

3 Practical 💻

They allow you to catch bugs at compile-time, effort is made when articulating consistency rather than finding inconsistencies at debug time...

Idris: A warm-up

```
x0 : Int -- x0 has type of Int AND
x0 = 1   -- x0 has value of 1

t0 : Type -- t0 has type of Type AND
t0 = Int  -- t0 has value of Int
```

Question from StackOverflow

How to specify constraints when defining data types in Haskell

Asked 1 month ago   Modified 1 month ago   Viewed 66 times

0

I understand the principle of "making illegal states unrepresentable" in functional languages, but I often have troubles putting it in practice.

As an example, I am trying to define a trading book model. I've defined these data types:

```haskell
data Side = Buy | Sell
    deriving (Show, Eq)

data Order =
    Order
    {
      orderSide      :: Side
    , orderQuantity  :: Int
    , orderPrice     :: Float
    }
    deriving (Eq)

data Book =
    Book
        { buy  :: [Order]
        , sell :: [Order]
        }
    deriving (Show)
```

Basically, meaning that a `Book` is a type with two lists of orders, one per side.

However, this is perfectly valid:

```
ghci> o = Order Sell 10 92.22
ghci> Book [o] []
Book {buy = [Order {orderSide = Sell, orderQuantity = 10, orderPrice = 92.22}], sel
```

And it is also perfectly wrong.

How can I express the constraint that only `Buy` orders should go to the buy side, and `Sell` orders on the other?

Idris: No dependent types

```
data Side : Type where
  Buy  : Side
  Sell : Side

t1 : Type
t1 = Side

record Order where
  constructor MkOrder
  side     : Side
  quantity : Int
  price    : Double

t2 : Type
t2 = Order

record Book0 where
  constructor MkBook0
  buy0  : List Order
  sell0 : List Order
```

```
aBuyOrder : Order
aBuyOrder = MkOrder
  { side     = Buy
  , quantity = 10
  , price    = 42.0
  }

aSellOrder : Order
aSellOrder = MkOrder
  { side     = Sell
  , quantity = 10
  , price    = 42.0
  }

buy : Order -> Bool
buy (MkOrder Buy _ _) = True
buy _                 = False

sell : Order -> Bool
sell (MkOrder Sell _ _) = True
sell _                  = False
```

Idris: A bit of dependent types 1

```
data BuyOrder : Order -> Type where
  YesBuyOrder : BuyOrder (MkOrder Buy quantity price)

t3 : Type
t3 = BuyOrder Example.aBuyOrder

-- x3 : Example.t3
x3 : BuyOrder Example.aBuyOrder
x3 = YesBuyOrder
```

```
data SellOrder : Order -> Type where
  YesSellOrder : SellOrder (MkOrder Sell quantity price)

t4 : Type
t4 = SellOrder Example.aSellOrder

x4 : Example.t4
x4 = YesSellOrder
```

```
data OkOrders
      :  (0 {-no runtime cost-} predicate : Order -> Type)
      -> List Order
      -> Type
  where
    Nil  : OkOrders predicate []
    Cons
      :  (0 ok : predicate order)
      -> (0 okay : OkOrders predicate orders)
      -> OkOrders predicate (order :: orders)

t5 : Type
t5 = OkOrders BuyOrder [aBuyOrder, aBuyOrder]

x5 : Example.t5
x5 = Cons YesBuyOrder $ Cons YesBuyOrder $ Nil
```

Idris: A bit of dependent types 2

```
record Book where
  constructor MkBook
  buy      : List Order
  sell     : List Order
  0 sellOk  : OkOrders SellOrder sell
  0 buyOk   : OkOrders BuyOrder  buy

t6 : Type
t6 = Book
```

```
assertBuyOrder : (o : Order) -> Maybe (BuyOrder o)
assertBuyOrder (MkOrder Buy _ _) = Just YesBuyOrder
assertBuyOrder _                 = Nothing

assertBuyOrders : (os : List Order) -> Maybe (OkOrders BuyOrder os)
assertBuyOrders [] = Just Nil
assertBuyOrders (o :: os) = case (assertBuyOrder o) of
  Nothing => Nothing
  Just b => case (assertBuyOrders os) of
    Nothing => Nothing
    Just bs => Just (Cons b bs)
```

```
assertSellOrder : (o : Order) -> Maybe $ SellOrder o
assertSellOrder (MkOrder Sell _ _) = Just YesSellOrder
assertSellOrder _                  = Nothing

assertSellOrders : (os : List Order) -> Maybe $ OkOrders SellOrder os
assertSellOrders [] = Just Nil
assertSellOrders (o :: os) = case (assertSellOrder o) of
  Nothing => Nothing
  Just s  => case (assertSellOrders os) of
    Nothing => Nothing
    Just zs => Just (Cons s zs)
```

Idris: A bit of dependent types 3

```
readSellOrders : IO (List Order)
readSellOrders = ?todo1

readBuyOrders : IO (List Order)
readBuyOrders = ?todo2

createBook3 : IO (Maybe Book)
createBook3 = do
  sell <- readSellOrders
  buy  <- readBuyOrders
  pure $ do
    sellOk <- assertSellOrders sell
    buyOk  <- assertBuyOrders buy
    Just $ MkBook
      { buy = buy
      , sell = sell
      , sellOk  = sellOk
      , buyOk = buyOk
      }
```

Adding Dependent Types to Your Code

No impossible cases

With dependent type programming, we can restrict data.

Visible assertions

Client codes see all assertions.

Keep calm, be consistent

Lack of consistency proofs results in compile errors.

Example from SQL: Safe Database Access

The Problem 👀

SQL databases allow access to tables and views that are only known at runtime.

The Solution 💡

Using dependent types, we can ensure that SQL queries are correct at compile time, improving the quality and security of the system

Example 🔍

The type system can ensure that queries always reference existing tables and don't result in non-matching or inconsistent column values.

Dependent types in SQL queries

```
record Table where
  constructor MkTable
  name        : TableName
  fields      : List Field
  constraints : List Constraint
  0 validTable : ValidTable fields constraints

data ValidTable : List Field -> List Constraint -> Type where
  YesOfCourseValid : ValidTable fields constraints
  -- TODO: Implement this check

data Query : Type where
  Select
    : (  fields    : List FieldName)
    -> (  table     : Table)
    -> (1 okFields  : SelectedFieldsDefinedInTable fields table.fields)
    => (  filters   : List (FieldName, String, String))
    -> (0 okFilters : FilteredFieldsDefinedInTable filters table.fields)
    => Query

renderQuery : Query -> String
renderQuery (Select fields table filters)
  = "SELECT \{withCommas fields} FROM \{table.name}" ++
    (case filters of
      [] => ""
      fs =>
        " WHERE " ++
        (withCommas
          $ map (\(field, op, cond) => "(\{field} \{op} \{cond})") fs)) ++
        ";"
```

Example from Domain Driven Design

1

2

3

The Problem

Keeping track high level model.

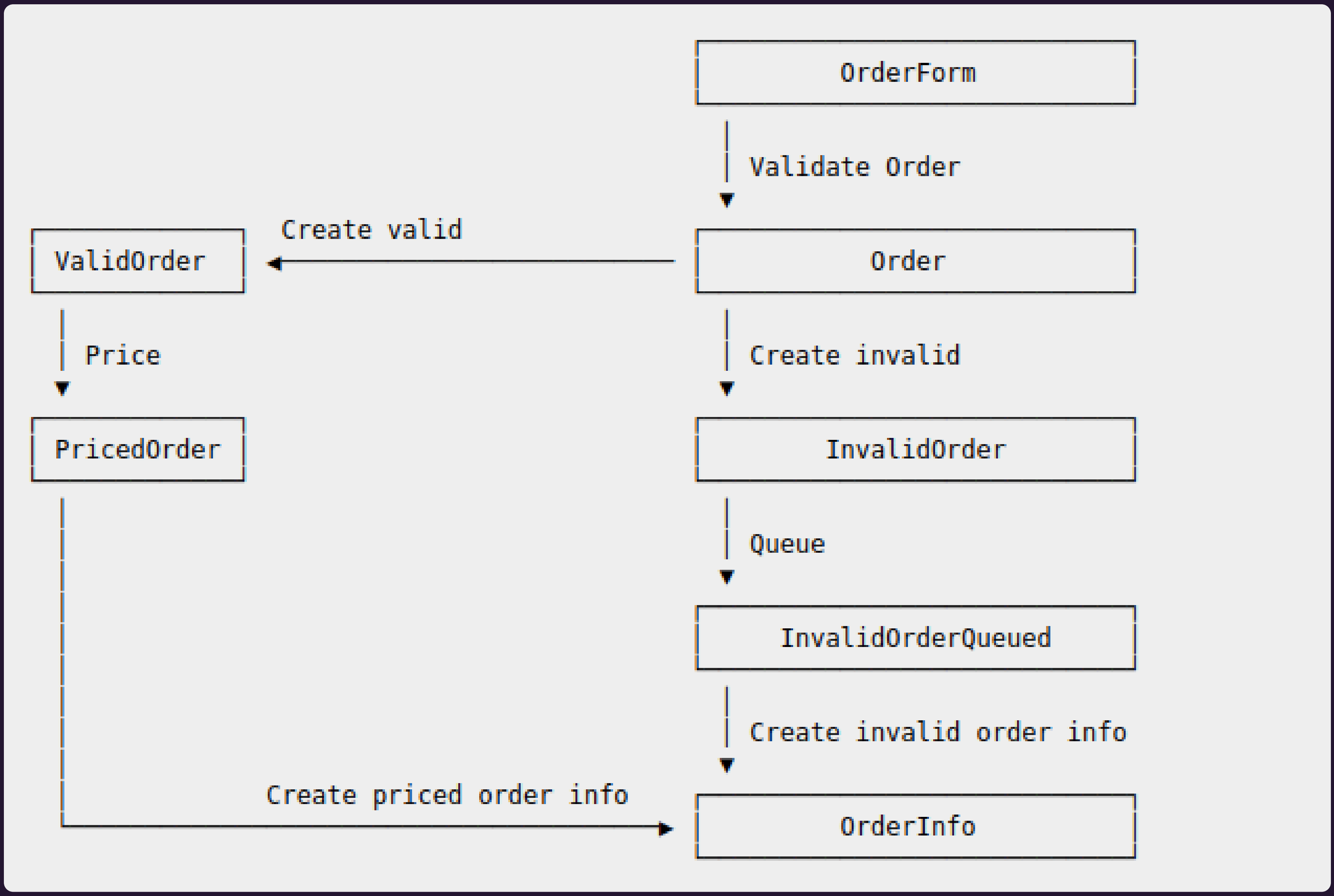
The Solution

Dependent typing can ensure that low level state transition respect high level models.

Example 🔍

See next slide.

Dependent types in DDD




```
data State
  = OrderForm
  | Order
  | ValidOrder
  | PricedOrder
  | InvalidOrder
  | InvalidOrderQueued
  | OrderInfo

data Step : State -> State -> Type where
  ValidateOrder     : Step OrderForm          Order
  AddInvalidOrder   : Step InvalidOrder       InvalidOrderQueued
  PriceOrder        : Step ValidOrder         PricedOrder
  SendAckToCustomer : Step PricedOrder        OrderInfo
  SendInvalidOrder  : Step InvalidOrderQueued OrderInfo
```

```
StateType : Overview.State -> Type
StateType OrderForm          = Domain.OrderForm
StateType Order              = Either Domain.InvalidOrder Domain.Order
StateType ValidOrder         = Domain.Order
StateType PricedOrder        = Domain.PricedOrder
StateType InvalidOrder       = Domain.InvalidOrder
StateType InvalidOrderQueued = List Domain.PlacedOrderEvent
StateType OrderInfo          = List Domain.PlacedOrderEvent

--                             s -> IO e
step : Overview.Step s e -> (StateType s) -> IO (StateType e)
step ValidateOrder     st = validateOrder st
step AddInvalidOrder   st = pure [InvalidOrderRegistered st]
step PriceOrder        st = priceOrder st
step SendAckToCustomer st = do
  ack <- acknowledgeOrder st
  placePricedOrder st
  pure $ createEvents st ack
step SendInvalidOrder  st = pure st
```

Example from STG

Semantics of STG

Internal representation of GHC runtime.

Compile Idris to STG

Haskell libraries can be used from Idris programs.

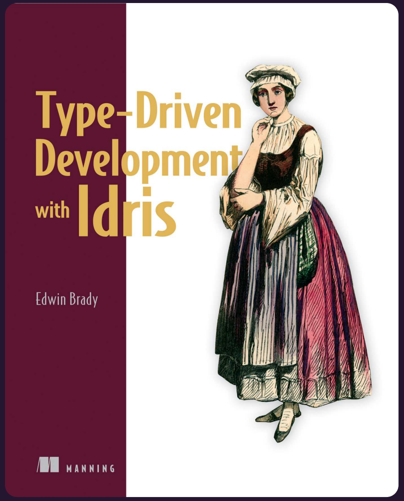
Dependent types in compilers

```
data STGExpr : Type where
 StgApp    : BinderId -> (List Arg)  -> STGExpr
 StgLit    : Lit                    -> STGExpr
 StgConApp : DataConId -> (List Arg) -> STGExpr
 StgOpApp  : PrimOp -> (List Arg)    -> STGExpr
 StgLet    : Binding -> STGExpr      -> STGExpr
 StgCase
  :  AltType
  -> STGExpr
  -> Binder
  -> (List Alt)                     -> STGExpr
```

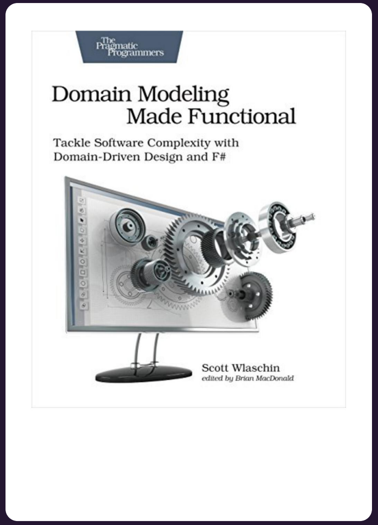
```
data STGExpr
   :  RepType {- Representation of return value -}
   -> Type where
 StgApp
  :  (qr : BinderId q) -> (Arguments qr) -> (r : RepType)
  -> STGExpr r
 StgLit
  :  (l : Lit)
  -> STGExpr (litRepType l)
 StgConApp
  :  (dr : DataConId r) -> (StgConAppArgType dr r)
  -> STGExpr (SingleValue LiftedRep)
 StgOpApp
  :  (p : PrimOp name args ret)
  -> (StgOpArgType p args)
  -> STGExpr (SingleValue ret)
 StgLet
  : (v : Binding) -> (b : STGExpr r)
  -> STGExpr (letBinderRep v b)
 StgCase
  :  (a : AltType)
  -> STGExpr (altRepType a)
  -> Binder (altRepType a)
  -> (List (Alt (altRepType a) r))
  -> STGExpr r
```

Conclusion and Further Resources

1 TDD with Idris 🔍



2 DDD Made functional



3 Going deep PLFA