Reasoning About Program Behavior Algebraically

Stephen Diehl (@smdiehl)
London 2017
“Type models do not specify behavior. The correctness of your type model has no bearing on the correctness of the behavior you have specified. At best the type system will prevent some mechanistic failures of representation (e.g. Double vs. Int); but you still have to specify every single bit of behavior; and you still have to test every bit of behavior.”

-Uncle Bob

Motivating Quote

“Type models do not specify behavior. The correctness of your type model has no bearing on the correctness of the behavior you have specified. At best the type system will prevent some mechanistic failures of representation (e.g. Double vs. Int); but you still have to specify every single bit of behavior; and you still have to test every bit of behavior.”

-Uncle Bob

Why Haskell is Interesting

Sufficiently rich type system in order to express complex invariants.

Sufficiently advanced compiler to write real programs.
Why Haskell Is Interesting

We want to construct code carries proofs of correctness under composition.
Algebraic Reasoning

**Abstraction** & **Generality**

**Abstraction**
Whole collections of objects and their relations.

**Generality**
We want to prove things in their most general form to allow reuse.
Structures

Set operations over $S$

Examples:

$\{1, 2, 3, 4\}, \{\blacksquare, \bullet, \nabla\}$

$[\times, \cup, \cap, -, +, \circ]$
Structures + Laws

Structure + Laws

We want to identify common structure.

We want to identify common ways to abstract operations.
Newtype Wrapping

{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Plaintext = Plaintext Text
newtype Cryptotext = Cryptotext Text

encrypt :: Key -> Plaintext -> Cryptotext
decrypt :: Key -> Cryptotext -> Plaintext
Phantom Types

data Message a = Message String

data PlainText

data Encrypted

send :: Message Encrypted -> IO ()
encrypt :: Message PlainText -> Message Encrypted
decrypt :: Message Encrypted -> Message PlainText
Uninhabited Types & Coercion

data Void

initial :: Seq Void a

terminal :: Seq a Void

Will not unify with any inhabited type (sans ⊥)
Promotion

data Foo = Bar | Baz

type Bar = 'Bar

type Baz = 'Baz
GADTs

data UExpr where
  UNum :: Int -> UExpr
  UBool :: Bool -> UExpr
  UAdd :: UExpr -> UExpr -> UExpr
  UISZero :: UExpr -> UExpr

data Expr (t :: EType) where
  Num :: Int -> Expr TInt
  Bool :: Bool -> Expr TBool
  Add :: Expr TInt -> Expr TInt -> Expr TInt
  IsZero :: Expr TInt -> Expr TBool

data EType = TInt | TBool
Promotion + GADTs

{-# LANGUAGE GADTs, DataKinds, KindSignatures #-}

data Sobriety = Sober | Drunk
data Driver (a :: Sobriety) where
  Alice :: Driver 'Sober
  Bob :: Driver 'Drunk

home = ()

driveCar :: Driver 'Sober -> IO ()
driveCar driver = return home

main = driveCar Bob
Type Families

type family TypeOf (t :: EType) :: * where
  TypeOf TInt  = Int
  TypeOf TBool = Bool

data Equal :: k -> k -> * where
  Refl :: Equal t t

Compute over Types

Witness
Singletons

data SEType (t :: EType) where
    STInt  :: SEType TInt
    STBool :: SEType TBool

equalType :: SEType t1 -> SEType t2 -> Maybe (Equal t1 t2)
equalType STInt  STInt   = Just Refl
equalType STBool STBool = Just Refl
equalType _        _      = Nothing

eval :: Expr t -> TypeOf t
eval (Num n) = n
eval (Bool b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (IsZero e) = eval e == 0
Type Families + Constraint Kinds

{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE ConstraintKinds #-}

import GHC.TypeLits
import Data.Type.Equality

type Not a b = ((b == a) ~ False)

restrictUnit :: Not () a => a -> a
restrictUnit = id
Our Goal

At the export boundaries of our libraries we export interfaces that:

Make illegal states unrepresentable by enforcing invariants under composition.
Complex Multiparty Workflows

Complex programs that model $n$-party workflows executed across a settlement network the control and transfer of rights, obligations, intellectual property and digital assets.

Simulate and stress test market conditions and prove coherence to regulation and specification.
Complex Multiparty Workflows

Alice and Bob engage in a swap with agreed parameters \( \{x, y, z\} \). At a time \( t \), Bob has the right, but not obligation to trade a fixed amount of value \( x \) with Alice. If he opts to engage in the trade he must notify Alice within \( [t - y, t + y] \) interval. Both Alice and Bob defer to a third party, Carol who supplies a fixed market feed who is obligated to return a value within \( [(t - y) - z, (t + y) + z] \) or notify both parties of failure. Carol is privy to the market price but not the amount \( x \) or counterparty Alice. Upon dispute of terms a regulator Dan is obligated to settle the final trade or amend parameters.
Complex Multiparty Workflows

State Constraints

Temporal Constraints

Information Constraints
Complex Multiparty Workflows

- Local node state.
- A set of channels.
- A set of access control boundaries.
- A set of valid transitions.
Policy Checking

- State Constraints
- Temporal Constraints
- Information Constraints

✔ Satisfiability
State Constraints

A program has a fixed set of entry points which are initiated at an initial element $0$ and form a directed graph to a terminal element $1$ in a fixed set of paths.

Each path through the program alters some state.
Temporal Constraints

Concern ourselves with the sequencing of events, triggering of events, and predicates over the state transitions.

- atomic propositions: $\circ, \circ, \ldots$
- boolean combinators: $\neg \varphi, \varphi \lor \psi, \varphi \land \psi, \ldots$
- temporal modalities:
  - $X \varphi$ — “next $\varphi$”
  - $\varphi U \psi$ — “$\varphi$ until $\psi$”
  - $\text{true} U \varphi \equiv F \varphi$ — “eventually $\varphi$”
  - $\neg F \neg \varphi \equiv G \varphi$ — “always $\varphi$”
- path quantifiers:
  - $E \varphi$
  - $A \varphi$
Information Constraints

- Access rights
- Separation of duties
- Delegation chains
Process Calculi

*Formalisms that capture the essence of interaction among independent agents*

We represent processes, parallel composition of processes, synchronous communication between processes through channels, creation of fresh channels, replication of processes, and nondeterminism.

*Can also add fix-point recursion, balancing and linear types.*
import GHC.TypeLits
import Control.Distributed.Process

data Chan (n :: Symbol) = forall a . MkChan (Chan a)

data Session where
  (!) :: forall a. a -> Session -> Session  -- Send
  (?) :: forall a. a -> Session -> Session  -- Receive
  (!!) :: forall a. a -> Session -> Session  -- Output
  (+) :: Session -> Session -> Session  -- Branch
End :: Session  -- Terminate

type family Seq s t where
  Seq End s    = s
  Seq (a ::? s) t = a ::? (Seq s t)
  Seq (a ::! s) t = a ::! (Seq s t)
  Seq (a ::?! s) t = a ::?! (Seq s t)
  Seq (s1 ::+ s2) t = (Seq s1 t) ::+ (Seq s2 t)
Type-level Maps

type family Map (f :: a -> b) (as :: [a]) :: [b] where
  Map f '[] = '[]
  Map f (x ': xs) = f x ': Map f xs

data Mapping k v = k :-> v
Process Calculi

send :: Chan c -> t -> Process '[c :-> t :: End] ()
recv :: Chan c -> Process '[c :-> t ::? End] t

Consumes a Typed Channel

Creates a new typed value to be consumed.

Extends type-level map

Technique by Dominic Orchard [2016]
Information Flow Control

Information flow statically describes which describes who can view trade details at what time and under what conditions. The rules can be expressions which are *time-dependent* or *time-independent*. 
Labels

data Label = L | H

lrt :: Label -> Label -> Bool
lrt H L = False
lrt _ _ = True

class Flows (l1 :: Label) (l2 :: Label) where
instance Flows L L
instance Flows L H
instance Flows H H

data Labeled (l::Label) a

Lattice
Low → High
Security

Can Generalize
To Other Lattice Structures
Labels

If the current label is $A$, then it is only permissible to read data labeled $L$ if $\text{Flows } A \rightarrow L$ or $A \subseteq L$.

```haskell
newtype Sec l a = Sec (IORef (SecState l) -> IO a)
instance Monad (Sec l) where
  return = Sec . const . return
  (Sec ma) >>= k = Sec $ \s -> do
    a <- ma s
    case k a of Sec mb -> mb s

  taint :: Label l => l -> Sec l ()
```

Restrict the interface for read/Writes to SecState to respect taint conditions.
Static vs Dynamic Boundaries

{-# LANGUAGE ConstraintKinds #-}

class Deferrable (c :: Constraint) where
deferC :: Proxy c -> (c => a) -> a
A Little Typeclass Prolog

type family Extract (l :: LExpr Label) :: Label where
  Extract (LVal x) = x
  Extract (LJoin x y) = Join (Extract x) (Extract y)

class Flows (Extract l1) (Extract l2)
  => FlowsE (l1 :: LExpr Label) (l2 :: LExpr Label) where

instance Flows (Extract l1) (Extract l2) => FlowsE l1 l2 where
Runtime Constraint Checking

instance (CLabel l1, CLabel l2) => Deferrable (FlowsE l1 l2) where
deferC p m = case (lab p1, lab p2) of
  (LL,LL) -> m
  (LL,LH) -> m
  (LH,LH) -> m
  (LH,LL) -> panic "Flow invariant violation at runtime."
where
  p1 = Proxy :: Proxy l1
  p2 = Proxy :: Proxy l2

instance (Typeable a, Typeable b) => Deferrable (a ~ b) where
deferC p m = case eqT :: Maybe (a ~: b) of
  Just Refl -> m
  Nothing -> panic "Type error"

Technique by Dimitrios Vytiniotis [2015]
We can use types to reason about non-trivial behavior of our programs.

These are tests we don’t have to write.
Thank you.