Eliminating Bugs with Dependent Haskell

Noam Zilberstein
Facebook Programming Languages & Runtimes
Haskell Symposium — August 27, 2020

Introduction

- Modern software is complicated and hard to reason about
- Dependent types can express strong correctness guarantees
- Critics claim that dependent types are not practical for real world use
- In this talk, we will explore real world examples where dependent types were used to eliminate bugs in production Haskell systems

Haskell @ Facebook

- Haskell is used to write abuse detection rules as part of a system called Sigma
- These rules prevent abuse such as spam, fake accounts, and fraud
- Correctness is crucial because code is deployed to production quickly in order to mitigate adversarial threats
- Sigma is large scale (over one million requests per second)

Programming with Dependent Types

- Goal: Express more invariants at the type level
- Haskell's type system is expressive, but it is not a fully dependently typed language
  - Con: Cannot express *everything* at the type level, need singletons to connect types to runtime values
  - Pro: More powerful type inference than a dependently typed language; GHC's constraint solver can automate more invariant checking
- Interesting result: expressive types guide the programmer's thinking even when they do not prove all invariants about the code

The Thrift IDL

- Thrift is an Interface Description Language
- Developers can define data structures and Remote Procedure Calls (RPCs)
- The Thrift Compiler translates Thrift code into code in some programming language (eg Haskell, C++, Python, etc)
- Sigma rules use extensively autogenerated Thrift code to fetch additional data needed to make decisions
  - Correctness is crucial; bugs in the Thrift compiler cause abuse detection rules to behave unexpectedly

Thrift Examples

```
typedef i64 Id

struct User {
  1: Id id,
  2: string name,
  3: Pet pet,
}

enum Pet {
  Dog = 0,
  Cat = 1,
}

service MyService {
  User getUser(1: Id id)
}
```

```
type Id = Int

data User = User
  { user_id   :: Id
  , user_name :: String
  , user_pet  :: Pet
  }

data Pet = Dog | Cat

getUser :: Id -> IO User
getUser user_id = ...
```

The Haskell Thrift Compiler

- The Haskell Thrift compiler uses dependent types in its internals to express correctness invariants
- The C++ Thrift compiler is used to compile Thrift to other languages
- The C++ implementation had many more bugs than the Haskell implementation including:
    - Infinite loops
    - Accepting ill-typed inputs
    - Ambiguous behavior

Basic AST Design

- A basic AST for Thrift IDL code may define a Thrift type as shown on the right
- This AST is not very expressive
  - Is this type wellformed?
  - What does a value of type `TInt` look like?
  - Is this named type a struct or an enum? Does it even exist?

```haskell
data Type
  = TInt
  | TBool
  | TString
  | TList Type
  | TMap Type Type
  | TNamed String
```

Constrained Data Structures

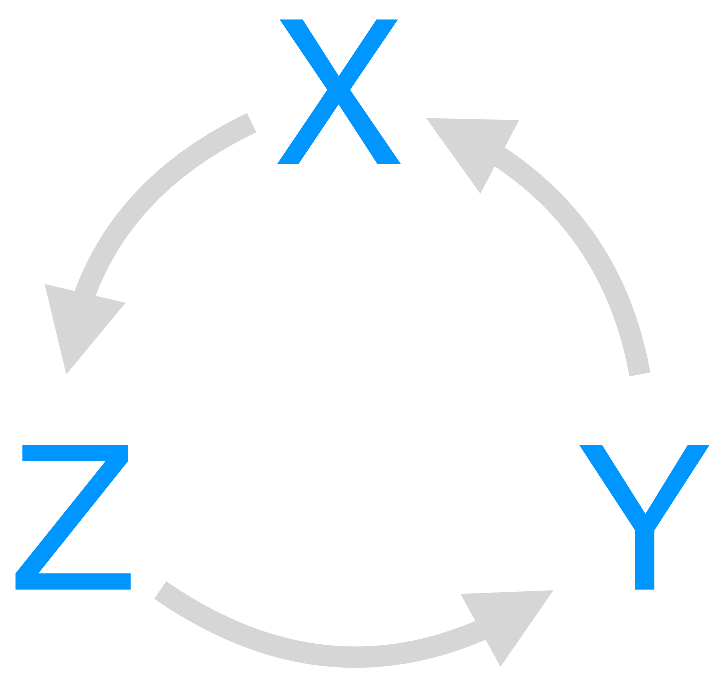
```haskell
data Status = Resolved | Unresolved

data Type (u :: Status) where
  TInt    :: Type u
  TBool   :: Type u
  TString :: Type u
  TList   :: Type u → Type u
  TMap    :: Type u → Type u → Type u

  -- Unresolved Named Type
  TNamed :: String → Type 'Unresolved

  -- Resolved Named Types
  TAlias
    :: String → Type 'Resolved → Type 'Resolved
  TStruct :: String → Type 'Resolved
  TEnum   :: String → Type 'Resolved
```

- Using GADTs and Data Kinds, we can ensure that named types get properly resolved
- Base types and collections can be either resolved of unresolved
- Named types can only be unresolved
- After typechecking, all named types must be converted to type aliases, structs, or enums

Bug: Infinite Loops

- The Thrift code on the right is invalid; the types X, Y, and Z form a loop
- When faced with this input, the C++ Thrift compiler diverged
- A correct solution requires topological sorting to find cycles
- In Haskell, the need to topological sorting was implied by the requirement for resolved types to be deeply resolved
    - ie, `TAlias "Y" (TNamed "X")` is ill-typed

```
typedef X Y
typedef Y Z
typedef Z X
```

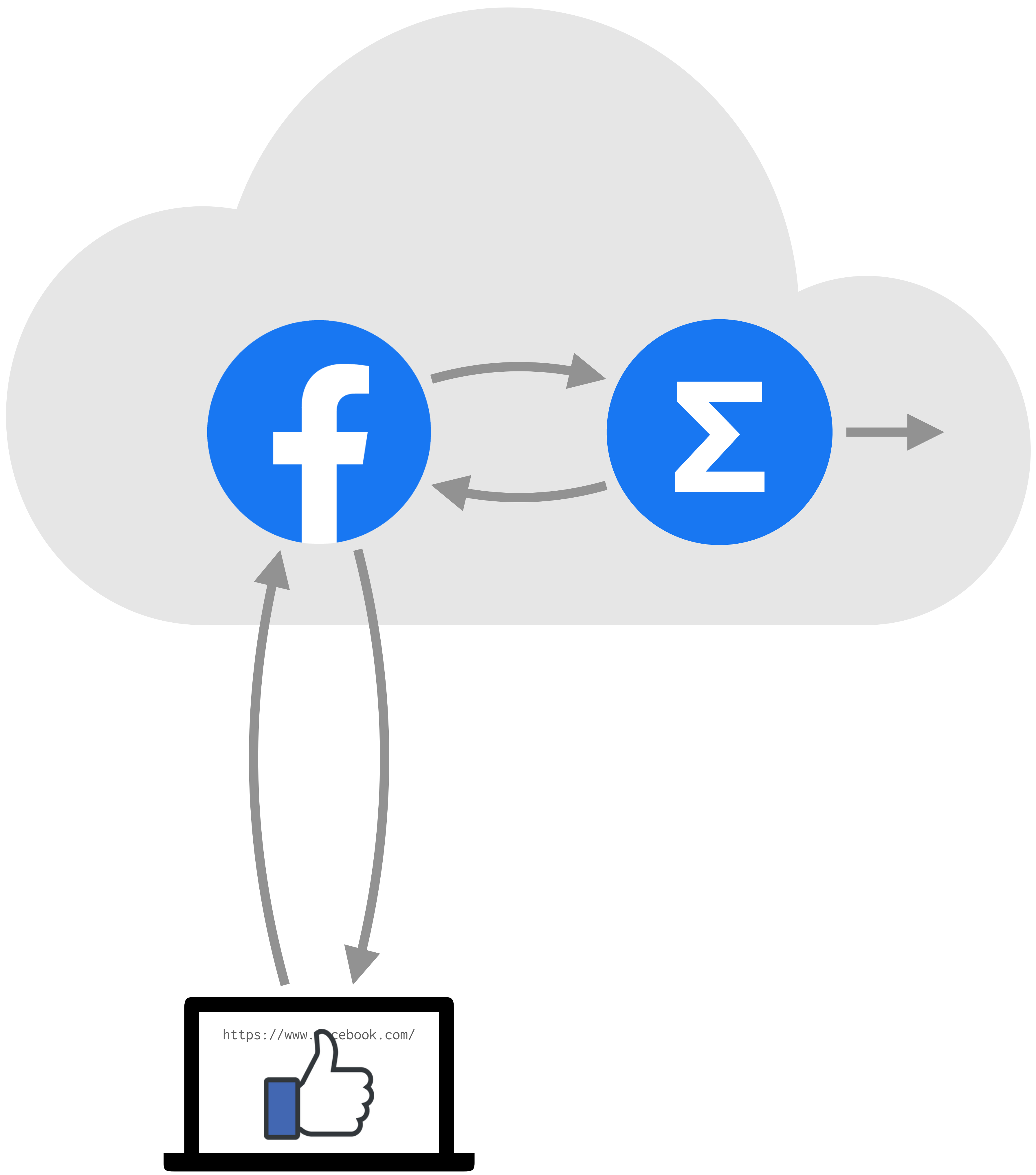


Sync vs Async Rules

- Sigma rules execute in two rounds (sync and async)
- Sync rules are run before a web request finishes and can affect the request (eg, tag with additional metadata)
- Async rules run after the request finishes and cannot affect the request (eg logging)

Sync vs Async Rules

- Sigma rules execute in two rounds (sync and async)
- Sync rules are run before a web request finishes and can affect the request (eg, tag with additional metadata)
- Async rules run after the request finishes and cannot affect the request (eg logging)



Sync vs Async Code

- We use a GADT to express which rounds a response can be used in
- Tagging a request must happen in the sync round whereas logging can happen at any time
- The code example on the right is ill-typed because it attempts to tag content in an async rule
- Before the type-level distinction was introduced, hundreds of these bugs were present in the code

```haskell
data RuleType = Sync | Async

data Response (t :: RuleType) where
  Tag :: Response 'Sync
  Log :: Response t


-- This code is ill-typed
checkScore :: Double → [Response 'Async]
checkScore score =
  if score > 0.9 then [Tag] else []
```

Expected 'Async, but got 'Sync

Associated Types

```haskell
data Status = Resolved | Unresolved
data Bottom

data Type (u :: Status) (t :: ⋆) where
  TInt    :: Type u Int
  TBool   :: Type u Bool
  TString :: Type u String
  TList   :: Type u t → Type u [t]
  TMap    :: Type u k → Type u v → Type u (Map k v)

  -- Unresolved Named Type
  TNamed :: String → Type 'Unresolved Bottom

  -- Resolved Named Types
  TAlias
    :: String → Type 'Resolved t → Type 'Resolved t
  TStruct :: String → Type 'Resolved ???
  TEnum   :: String → Type 'Resolved ???
```

- We extend the Type GADT to include a second parameter
- This parameter tells us what a wellformed value looks like
- We associate this parameter with other types in function signatures to ensure that typechecked literals are wellformed
- Wellformed values can still go wrong, but this invariant is enough to prevent most accidental errors

Associated Types

```haskell
data UntypedConst
  = IntLit Int
  | StrLit String
  | BoolLit Bool
  | ListLit [UntypedConst]
  | MapLit [(UntypedConst, UntypedConst)]
  | Ident String

data TypeConst t
  = Identifier String (Type 'Resolved t)
  | Literal t
```

```haskell
typecheckConst
  :: Type 'Resolved t
  → UntypedConst
  → Either TypeError (TypedConst t)

-- Wellformed Literals
typecheckConst TInt (IntLit n) =
  Right $ Literal n
typecheckConst TString (StrLit s) =
  Right $ Literal s

-- Ill-typed!
typecheckConst TInt (StrLit s) =
  Right $ Literal s
```

Typed Data Fetches

- Sigma uses a library called Haxl for async data fetching and caching
- Data fetch requests are represented as GADTs, each request declares its return type
- These data constructors are also used as cache keys, enabling type-safe lookups

```haskell
data Request a where
  GetName :: Id → Request String
  GetPet  :: Id → Request Pet

dataFetch :: Request a → Haxl a
cacheLookup :: Request a → Haxl a
cacheInsert :: Request a → a → Haxl ()

getName :: Int → Haxl String
getName userId =
  dataFetch $ GetName userId
```

Type-Level Schemas

```haskell
data Status = Resolved | Unresolved
data Bottom

data Type (u :: Status) (t :: ⋆) where
  TInt    :: Type u Int
  TBool   :: Type u Bool
  TString :: Type u String
  TList   :: Type u t → Type u [t]
  TMap    :: Type u k → Type u v → Type u (Map k v)

  -- Unresolved Named Type
  TNamed :: String → Type 'Unresolved Bottom

  -- Resolved Named Types
  TAlias
    :: String → Type 'Resolved t → Type 'Resolved t
  TStruct :: String → Type 'Resolved ???
  TEnum   :: String → Type 'Resolved ???
```

What can we put here?

- GHC cannot trivially check wellformedness of structs and enums
- We need to dynamically generate a representation of their types
- This is possible using type-level schemas

Struct Schemas

- Wellformed structs have wellformed values for all of their named fields
- The kind of struct schemas is a type-level list of type-level string (of kind Symbol) and type (of kind ★) pairs
- This allows us to define schemas and values for structs that can be associated using a type of kind `[(Symbol, ★)]`
- KnownSymbol allows us to get a runtime representation of the type-level string

```
data Schema (s :: [(Symbol, ★)]) where
  SNil :: Schema '[]
  SCons
    :: ∀ (name :: Symbol) t s. KnownSymbol name
    ⇒ Type 'Resolved t
    → Schema s
    → Schema ('(name, t) ': s)

data StructVal (s :: [(Symbol, ★)] where
  SVNil
    :: Schema '[]
  SVCons
    :: ∀ (name :: Symbol) t s. KnownSymbol name
    ⇒ Type 'Resolved t
    → TypeConst t
    → StructVal s
    → StructVal ('(name, t) ': s)
```

Typechecking Structs

```
typecheckStruct
  :: Schema s
  → [(UntypedConst, UntypedConst)]
  → Either TypeError (StructVal s)
typecheckStruct = ...


typecheckConst
  :: Type 'Resolved t
  → UntypedConst
  → Either TypeError (TypedConst t)

-- Struct Case
typecheckConst (TStruct _ schema) (MapLit fields) =
  Literal <$> typecheckStruct schema fields
```

```
userSchema
  :: Schema
     '[ ("id", Int)
      , ("name", String)
      , ("pet", (EnumSchema ...))
      ]
userSchema =
  SCons @"id" TInt
  (SCons @"name" TString
   (SCons @"pet" (TEnum "Pet" ...)
    SNil))
```

Enum Schemas

```
data EnumSchema (s :: [Symbol]) where
  ESNil :: EnumSchema '[]
  ESCons
    :: ∀ (name :: Symbol) s. KnownSymbol name
    ⇒ Proxy name
    → EnumSchema s
    → EnumSchema (name ': s)

data EnumVal (s :: [Symbol]) =
  ∀ n. EnumVal String (MembershipProof n s)

data MembershipProof x xs where
  PHere :: MembershipProof x (x ': xs)
  PThere
    :: MembershipProof x xs
    →  MembershipProof x (y ': xs)
```

- Wellformed enums can be one of many values
- An enum schema is a type-level list of allowed identifier names
- Typechecked enum values require a proof that the enum's identifier is a member of the schema list

Typechecking Enums

```haskell
typecheckEnum
  :: EnumSchema s
  → Proxy name
  → Maybe (MembershipProof name s)
typecheckEnum ESNil _ = Nothing
typecheckEnum (ESCons name s) name' =
  case eqT name name' of
    Just Refl → Just PHere
    Nothing → PThere <$> typecheckEnum s name'

typecheckConst
  :: Type 'Resolved t
  → UntypedConst
  → Either TypeError (TypedConst t)

-- Enum Case
typecheckConst (TEnum schema) (Ident symbol) =
  case someSymbolVal symbol of
   SomeSymbol name →
    case typecheckEnum schema name of
     Just pf → Right $ Literal $ EnumVal symbol pf
     Nothing → Left $ TypeError $ ...
```

- Typechecking an enum builds an inductive membership proof
- Building the proof introduces additional time and space complexity
- We could improve the runtime using a different type-level data structure, but it would complicate the code
- In practice, performance was not an issue

More Bugs: Enum Typechecking

- In the example on the right, the first two constants are valid, but the third is ill-typed because X has no member with value 3
- The C++ Thrift typechecker would have accepted all of these inputs because it treated enums as integers
- In Haskell, this bug would not have been possible due to the requirement of building a membership proof

```
enum X {
  A = 0,
  B = 1,
  C = 2,
}

// Valid Enum Values
const X b_int = 1
const X b_name = B

// Type Error!
const X invalid_value = 3
```

More Bugs: Implicit Coercions

```
enum Status {
  Ok = 0,
  Error = 1,
}

enum Result {
  ERROR = 0,
  OK = 1,
}

const Status error_status = ERROR
```

- In the code on the left, error_status appears to be an error, but it is actually Ok
- The C++ Thrift typechecker would have accepted this input
- In Haskell, it would be impossible to accept this code because ERROR is not a member of the schema for Status
- A bug of this nature was found in production due to the Haskell Thrift typechecker

More Bugs: Ambiguous References

- In Thrift, values from other modules must be qualified and enum values can be optionally qualified
- This leads to ambiguous behavior: is the value on the right equal to 0 or 12345?
- The C++ Thrift typechecker arbitrarily resolved these, leading to silent bugs

```
enum Animal {
  Dog = 0,
  Cat = 1,
}

// Is this 0 or 12345???
const i32 dog = Animal.Dog
```

```
// Animal.thrift

const i32 Dog = 12345
```

Schematized Inputs

```
-- Input Lookup API
lookupInput :: FromJSON a ⇒ Text → Haxl a

commenterIsFriend :: Haxl Bool
commenterIsFriend = do
  poster    ← lookupInput "PostAuthor"
  commenter ← lookupInput "CommentAuthor"
  poster `isFriendOf` commenter
```

- Sigma rules receive inputs via untyped JSON input-maps
- This code can fail in two ways:
  - The key may not be present in the input map
  - The key may be present, but with a different type
- Lookup failures are very prominent in production
- Strongly typed inputs are difficult because of code sharing

Solution: Type-Level Schemas

- Schema is encoded as a constraint
- Code sharing is easy: just implement the Has type class for any underlying input type
- Lookups are pure, they can't fail at runtime
- The getter uses a visible type application (it takes no term arguments)
  - This is a foreign concept to most Sigma developers, but the syntax is natural to use

```haskell
-- Typesafe Lookup API
get
  :: ∀ (key :: Symbol) ty input.
     Has key ty input
  ⇒ ty

commenterIsFriend
 :: ( Has "PostAuthor" Id input
    , Has "CommentAuthor" Id input
    ) ⇒ Haxl Bool
commenterIsFriend = do
  let
    poster = get @"PostAuthor"
    commenter = get @"CommentAuthor"
  poster `isFriendOf` commenter
```

Conclusion

- The increasing complexity of modern codebases makes it difficult to reason about correctness
- Using dependent types is a practical way to eliminate bugs in production
- Current and future Haskell projects should take advantage of dependent types
- Given these promising results, other languages should increase the expressivity of their type systems