Enforcing authorization checks with the type system

Ghosts of Departed Proofs (GDP) about JWTs

Joonas Laukka

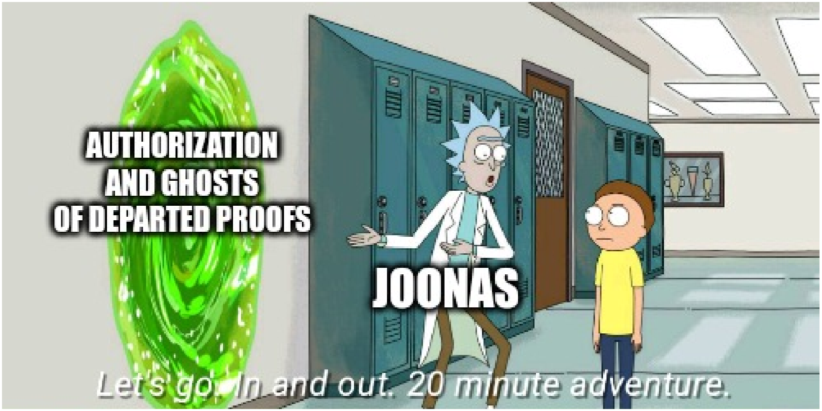
Software Developer @ RELEX Solutions

Jan 11, 2023

About this talk

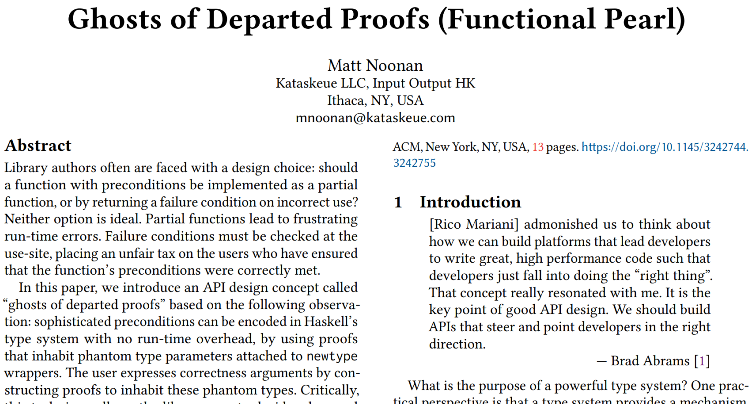
- Prerequisites:
    - Requires intermediate knowledge of Haskell
    - Haskell 2010 and many of GHC Language Extensions
- Goals:
    - Sharing my excitement with GDP
    - Solving an imaginary problem related to authorization with GDP
    - Getting all of you interested in Ghosts of Departed Proofs
- Let's go on an adventure together!

About this talk


AUTHORIZATION AND GHOSTS OF DEPARTED PROOFS
JOONAS
Let's go. In and out. 20 minute adventure.

Contents

Crash course to GDP



Ghosts of Departed Proofs (Functional Pearl)

Matt Noonan
Kataskeue LLC, Input Output HK
Ithaca, NY, USA
mnoonan@kataskeue.com

Abstract

Library authors often are faced with a design choice: should a function with preconditions be implemented as a partial function, or by returning a failure condition on incorrect use? Neither option is ideal. Partial functions lead to frustrating run-time errors. Failure conditions must be checked at the use-site, placing an unfair tax on the users who have ensured that the function's preconditions were correctly met.

In this paper, we introduce an API design concept called "ghosts of departed proofs" based on the following observation: sophisticated preconditions can be encoded in Haskell's type system with no run-time overhead, by using proofs that inhabit phantom type parameters attached to `newtype` wrappers. The user expresses correctness arguments by constructing proofs to inhabit these phantom types. Critically,

ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3242744.3242755

1 Introduction

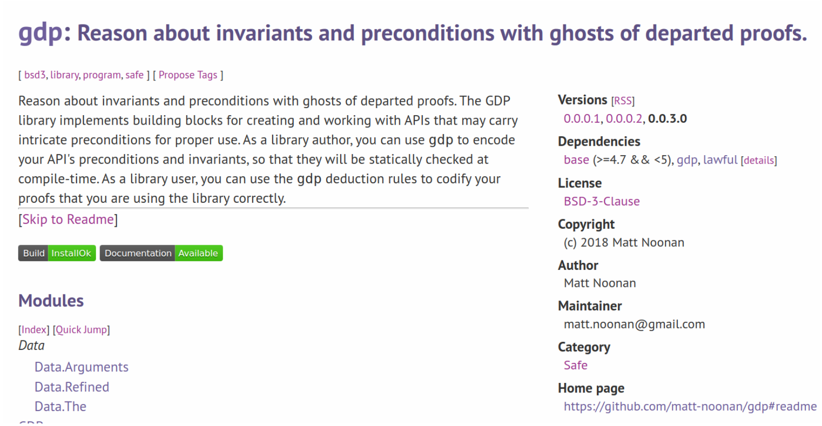
[Rico Mariani] admonished us to think about how we can build platforms that lead developers to write great, high performance code such that developers just fall into doing the "right thing". That concept really resonated with me. It is the key point of good API design. We should build APIs that steer and point developers in the right direction.

— Brad Abrams [1]

What is the purpose of a powerful type system? One practical perspective is that a type system provides a mechanism

Figure 1: Published in 2018 by Matt Noonan

Crash course to GDP



gdp: Reason about invariants and preconditions with ghosts of departed proofs.

[bsd3, library, program, safe] [Propose Tags]

Reason about invariants and preconditions with ghosts of departed proofs. The GDP library implements building blocks for creating and working with APIs that may carry intricate preconditions for proper use. As a library author, you can use gdp to encode your API's preconditions and invariants, so that they will be statically checked at compile-time. As a library user, you can use the gdp deduction rules to codify your proofs that you are using the library correctly.

[Skip to Readme]

Modules

[Index] [Quick Jump]

Data

- Data.Arguments
- Data.Refined
- Data.The

Versions [RSS]	0.0.0.1, 0.0.0.2, **0.0.3.0**
Dependencies	base (>=4.7 && <5), gdp, lawful [details]
License	BSD-3-Clause
Copyright	(c) 2018 Matt Noonan
Author	Matt Noonan
Maintainer	matt.noonan@gmail.com
Category	Safe
Home page	https://github.com/matt-noonan/gdp#readme

Figure 2: An auxiliary library is available on Hackage

Crash course to GDP

> *What is the purpose of a powerful type system? One practical perspective is that a type system provides a mechanism for enforcing program invariants at compile time.*

Examples:

- Total functions with a limited domain
    - `headSafe :: NonEmpty a -> a`
- Functions that force the caller to handle a failure condition
    - `parse :: String -> Either Error a`
- Pure functions that have local, mutable state
    - `ST` monad
- Functions that require proof of authorization
    - `fireMissiles :: Proof AllowedToFire -> IO ()`

Crash course to GDP

- Simple, everyday Haskell features already allow encoding very sophisticated invariants
    - These classic methods (such as newtype wrappers) are often sufficient
- But we're gonna play with GDP for the fun of it!
    - In the real world, the complexity of GDP itself might not be worth the value it brings (in most cases)

Crash course to GDP

- GDP library attempts to make it easy to design APIs that are
  - *safe*: we prevent the user from causing a run-time error
  - *ergonomic*: correct use of the API must not place an undue burden on the user

Fundamental features of GDP API design concept:

- Naming objects
- Properties and proofs are represented in code
- Proofs are carried by phantom type parameters
- Library-controlled APIs to create proofs
- Combinators for manipulating ghost proofs

Crash course to GDP: naming objects

```haskell
module Theory.Named (Named, type (~~), name) where

newtype Named name a = Named a
type role Named nominal nominal

-- | An infix alias for 'Named'.
type a ~~ name = Named name a

-- | Compiler conjures a unique, existential name for
-- the value 'x'.
--
-- Similar to the well known ST trick.
name :: a -> (forall name. a ~~ name -> t) -> t
name x cont = cont (coerce x)
```

Crash course to GDP: naming objects

In practice, it is as if the library has a secret supply of names, and selects one to use in a manner that is not predictable to the user.

Crash course to GDP: naming objects

Naming allows attaching values to proofs about those values

```
isPrime
  :: (Int ~~ n)
  -> Maybe (Proof (IsPrime n))

isIssuedBy
  :: (JWT ~~ token)
  -> Maybe (Proof (token `IsIssuedBy` issuer))
```

The named object can be almost anything, e.g. a function.

Crash course to GDP: naming objects

```
usePrime
  :: (Int ~~ n)
  -> Proof (IsPrime n)
  -> m ()

example x =
  name x $ \namedX -> name 2 $ \namedTwo ->
    let twoIsPrime = fromJust $ isPrime namedTwo
    in usePrime namedX twoIsPrime
```

results in compiler error

```
• Couldn't match type ‘name1’ with ‘name’
      Expected: Int ~~ name1
        Actual: Int ~~ name
• In the first argument of ‘usePrime’, namely ‘namedX’
```

Crash course to GDP: naming objects

Results of library functions can also be named.

```
newtype Inc n = Inc Defn

increment :: (Int ~~ n) -> (Int ~~ Inc n)
increment n = defn (the n + 1)
```

Crash course to GDP: unnaming objects

```haskell
newtype Named name a = Named a

class The d a | d -> a where
  the :: d -> a
  default the :: Coercible d a => d -> a
  the = coerce

instance The (Named name a) a

-- the :: (Int ~~ n) -> Int
-- the = coerce
```

This is useful when writing proof constructors.

Crash course to GDP: proofs

```haskell
module Logic.Proof (Proof, axiom) where

-- | Value of type 'Proof p' represents proof of @p@
data Proof p = QED

axiom :: Proof p
axiom = QED

module IsOdd (IsOdd, isOdd)

-- | Proposition: integer is odd
data IsOdd n

isOdd :: (Int ~~ n) -> Maybe (Proof (IsOdd n))
isOdd n = if odd (the n)
  then Just axiom
  else Nothing
```

Crash course to GDP: proofs

Values can be attached with proofs via `:::`.

```
newtype a ::: p = SuchThat a

usePrime
  :: (Int ~~ n)
  -> Proof (IsPrime n)
  -> m ()

usePrime
  :: (Int ~~ n ::: IsPrime n)
  -> m ()

(...)  :: a -> Proof p -> (a ::: p)

exorcise :: (a ::: p) -> a
conjure  :: (a ::: p) -> Proof p
```

Crash course to GDP: proofs

The proof attached to a value can be completely unrelated to it but usually there exists a clear connection:

```
-- "A value of type Int, named n, such that n is a prime"
(Int ~~ n ::: IsPrime n)

-- "A value of type Int, named n, such that the increment of n is odd"
(Int ~~ n ::: IsOdd (Inc n))

-- "A value of type Int, named n, such that the sun is shining"
(Int ~~ n ::: SunIsShining)
```

Crash course to GDP: proofs

```haskell
module Logic.Propositional where

-- Logical connectives
data p && q
data p || q
data p --> q
data p == q

-- Introduce premises
introImpl
  :: (Proof p -> Proof q)
  -> Proof (p --> q)
-- Deduction rules
elimImpl
  :: Proof p
  -> Proof (p --> q)
  -> Proof q
```

Crash course to GDP: proofs

The functions in `Logic.Propositional` allow building proofs from other proofs:

```
data IsNatural n

-- Increment of a natural number is a natural number.
naturalInc :: Proof (IsNatural n --> IsNatural (Inc n))
naturalInc = axiom

incrementNatural
  :: (Int ~~ n ::: IsNatural n)
  -> (Int ~~ Inc n ::: IsNatural (Inc n))
incrementNatural number =
  let proof       = conjure number
      incremented = increment $ exorcise number
  in incremented ...> (`elimImpl` naturalInc)
```

Crash course to GDP: proofs

Writing axioms in Haskell

Increment of a natural number is a natural number.

$$n \in \mathbb{N} \implies n + 1 \in \mathbb{N}$$

```
naturalInc
  :: Proof (IsNatural n --> IsNatural (Inc n))
naturalInc = axiom
```

Crash course to GDP: proofs

Writing axioms in Haskell

User can delete applications if his token was issued by Azure or OKTA

```
data CanDeleteApps token

canDeleteApps
  :: Proof (
       ( token `IssuedBy` "azure"
         || token `IssuedBy` "okta"
       )
       -->
       (CanDeleteApps token)
     )
canDeleteApps = axiom
```

Domain specific knowledge: What is authorization?

According to wikipedia:

> *"Authorization is the function of specifying access rights/privileges to resources."*

In practice,

> *"During operation, the system uses the access control rules to decide whether access requests from authenticated consumers shall be approved (granted) or disapproved (rejected)."*

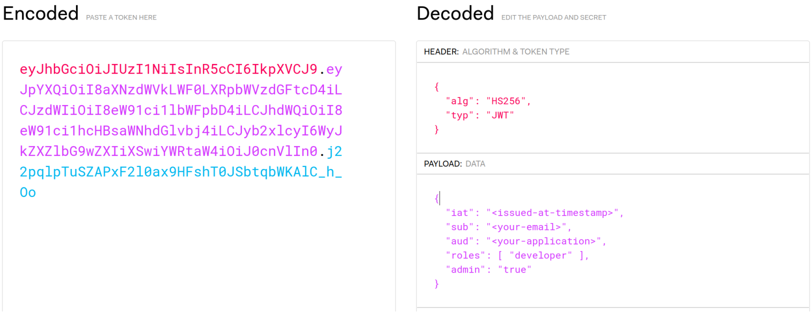
Domain specific knowledge: JSON Web Tokens

Figure 3: OAuth is an industry standard protocol for authorization

"JSON Web Tokens (JWTs) are an open, industry standard RFC 7519 method for representing claims securely between two parties."

- JSON Web Tokens are everywhere!
  - OAuth 2.0 access tokens are often JWTs
  - OpenID Connect identity tokens are always JWTs
- JWT claims can carry information needed for authorization decisions
- I chose to focus on JWTs in this talk!

Domain specific knowledge: JWT structure


Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JpYXQiOiI8aXNzdWVkLWF0LXRpbWVzdGFtcD4iL
CJzdWIiOiI8eW91ci1lbWFpbD4iLCJhdWQiOiI8
eW91ci1hcHBsaWNhdGlvbj4iLCJyb2xlcyI6WyJ
kZXZlbG9wZXIiXSwiYWRtaW4iOiJ0cnVlIn0.j2
2pqlpTuSZAPxF2l0ax9HFshT0JSbtqbWKAlC_h_
Oo
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "iat": "<issued-at-timestamp>",
  "sub": "<your-email>",
  "aud": "<your-application>",
  "roles": [ "developer" ],
  "admin": "true"
}
```

Figure 4: An example JWT

Domain specific knowledge: JWTs in Haskell ecosystem

- A hackage package called *jose* provides excellent primitive functions for
  - validating/verifying tokens
  - extracting claims from valid tokens

jose types

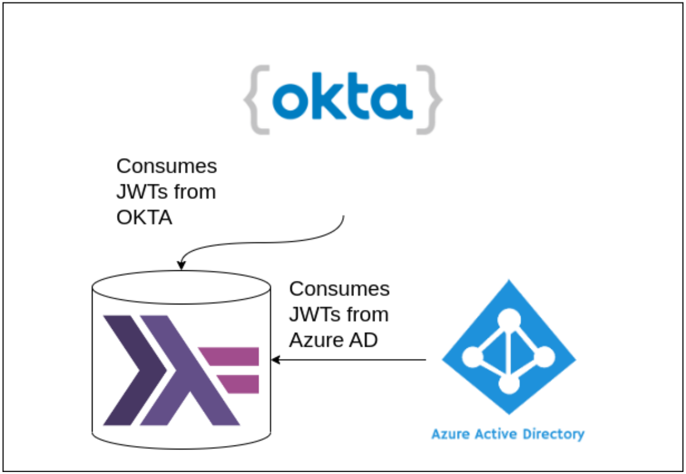
```haskell
-- | RFC 7517 §4.  JSON Web Key (JWK) Format
data JWK

-- | A digitally signed or MACed JWT
type SignedJWT = CompactJWS JWSHeader

-- | The JWT Claims Set represents a JSON object
data ClaimsSet
```

Problem statement

Let's say we are developing a Haskell application that accepts JWTs from two distinct Authorization Servers, `Azure AD` and `OKTA`.



Problem statement

- Let's assume that both Azure AD and OKTA maintain a record of which roles a user is assigned to
- And those roles are sent as claims in the JWTs issued by them
- Unfortunately, the schemas of the payloads don't quite match.

Problem statement: JWTs from Azure

The payload of a JWT issued by Azure looks like this:

```json
{
  "iss": "Azure AD",
  "sub": "<your-username>",
  "iat": "<issued-at-timestamp>",
  "aud": "<your-application>",
  "roles": [
     "<role-name>",
     "<another-role-name>"
  ]
}
```

- All of the roles that the user is assigned to are listed in the `roles` claim

Problem statement: JWTs from OKTA

The payload of a JWT issued by OKTA looks like this:

```
{
  "iss": "OKTA",
  "sub": "<your-username>",
  "iat": "<issued-at-timestamp>",
  "aud": "<your-application>",
  "<role-name>": "true",
  "<another-role-name>": "false"
}
```

- Each role has its own claim in the JWT
    - The claim is named according to the name of the role
    - The value of the claim is binary The user either (true or false)

Problem statement

Problem

Given that you receive a JWT from the user as input, how would you write a function that can only be run if the user

- is an **administrator** according to Azure?
- is an **administrator** or **developer** according to OKTA?
- is an **administrator** according to either Azure or OKTA?

Applying GDP to authorization

For a simple ad-hoc solution, we need

1. properties about JWTs such as
   - `data IssuedBy token source`
   - `data HasAzureRole claims roleName`
   - `data HasOktaRole claims roleName`
2. names such as
   - `newtype ClaimsOf token = ClaimsOf Defn`
3. properties about permissions such as
   - `data CanDeleteApps token`

Applying GDP to authorization: validation

```haskell
module Validation.IssuedBy
  (IssuedBy, ClaimsOf, getClaimsOf) where

data IssuedBy token (issuer :: Symbol)

newtype ClaimsOf token = ClaimsOf Defn

-- > getClaimsOf @"azure" token
--
-- provides a proof if token was signed by Azure
getClaimsOf
  :: forall issuer token settings m.
      ( KnownSymbol issuer, MonadReader s m
      , HasJWK issuer s
      )
  => (SignedJWT ~~ token)
  -> m (Maybe (
      ClaimsSet ~~ ClaimsOf token ::: (token `IssuedBy` issuer))
     )
```

Applying GDP to authorization: Azure role

```haskell
module Validation.Azure.HasRole
  (HasAzureRole, hasAzureRole) where

data HasAzureRole claims (roleName :: Symbol)

-- @
--   mAzureClaims <- getClaimsOf @"azure" token
--   case mAzureClaims of
--     Just claims -> hasAzureRole @"developer" claims
-- @
hasAzureRole
  :: forall roleName token. KnownSymbol roleName
  => (ClaimsSet ~~ ClaimsOf token ::: (token `IssuedBy` "azure"))
  -- ^ Only accepts claims from a JWT issued by Azure
  -> Maybe (Proof (ClaimsOf token `HasAzureRole` roleName))
```

Applying GDP to authorization: Azure role

If we mix up claims from Azure and OKTA:

```
example jwt =
  name jwt $ \namedJwt ->
    mOktaClaims <- getClaimsOf @"okta" namedJwt
    case mOktaClaims of
      Just oktaClaims -> hasAzureRole @"developer" oktaClaims
```

the compiler screams at us

```
• Couldn't match type ‘"okta"’ with ‘"azure"’
  Expected: (ClaimsSet ~~ ClaimsOf name) ::: IssuedBy name "azure"
    Actual: (ClaimsSet ~~ ClaimsOf name) ::: IssuedBy name "okta"
• In the second argument of ‘hasAzureRole’, namely ‘oktaClaims’
  In the expression: hasAzureRole @"developer" oktaClaims
```

Applying GDP to authorization: OKTA role

```
module Validation.Okta.HasRole
  (HasOktaRole, hasOktaRole) where

data HasOktaRole claims (roleName :: Symbol)

hasOktaRole
  :: forall roleName token. KnownSymbol roleName
  => (ClaimsSet ~~ ClaimsOf token ::: (token `IssuedBy` "okta"))
  -> Maybe (Proof (ClaimsOf token `HasOktaRole` roleName))
hasOktaRole claims =
  let roleClaimKey = pack $ symbolVal $ Proxy @roleName
      extraClaims  = view unregisteredClaims $ the claims
      mRoleClaim   = M.lookup roleClaimKey extraClaims
  in case mRoleClaim of
      Just (String "true") -> Just axiom
      _                    -> Nothing
```

Applying GDP to authorization: privileges

Now we can easily write clear axioms describing who is allowed to do what.

```haskell
module Authorization.Axioms where

data CanViewApps token

-- | Tokens issued by either Azure or OKTA authorize
-- the user to view applications.
canViewApps ::
  Proof (
    ((token `IssuedBy` "azure") || (token `IssuedBy` "okta"))
    -->
    (CanViewApps token)
  )
canViewApps = axiom
```

Applying GDP to authorization: privileges

```haskell
module Authorization.Axioms where

data CanDeleteApps token

-- | Tokens issued by Azure claiming that the user has "administrator"
-- role authorize the user to delete applications.
canDeleteApps ::
  Proof (
    ( token `IssuedBy` "azure"
      && (ClaimsOf token) `HasAzureRole` "administrator"
    )
    -->
    (CanDeleteApps token)
  )
canDeleteApps = axiom
```

Applying GDP to authorization: enforcement

```haskell
module Application.Delete where

deleteApplicationSafe
  :: MonadIO m
  => Proof (CanDeleteApps token)
  -> m ()
deleteApplicationSafe _ = liftIO $ putStrLn "Danger zone"

-- | One way to build the necessary proof.
buildAuthorizationProof
  :: (MonadIO m, MonadReader s m, HasJWK "azure" s)
  => SignedJWT ~~ token
  -> m (Maybe (Proof (CanDeleteApps token)))
buildAuthorizationProof jwt = runMaybeT $ do
  claims      <- MaybeT $ getClaimsOf @"azure" jwt
  proofOfRole <- MaybeT $ pure $ hasAzureRole @"administrator" claims
  let proofOfSignature = conjure claims
      proofOfAuthorization =
        (proofOfSignature `introAnd` proofOfRole)
          `elimImpl` canDeleteApps
  return proofOfAuthorization
```

Applying GDP to authorization: conclusion

What did we gain?

- Type safe authorization guarantees for protected functions
    - such as `deleteApplicationSafe`
- Clean, easy-to-read authorization rules via axioms
    - such as `canDeleteApps`
- Domain-specific proof generators about JWTs
    - such as `getClaimsOf`, `hasAzureRole`
- The user of a protected function can decide how he wants to prove the required property
    - He can use any available axioms

The idea could be brought to its logical conclusion by writing generic proof generators on any JWT since the payloads of JWTs are always JSON objects.

Language extensions

The most important GHC language extensions used in this presentation are:

```
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE AllowAmbiguousTypes   #-}
{-# LANGUAGE ScopedTypeVariables   #-}
{-# LANGUAGE TypeApplications      #-}
{-# LANGUAGE TypeOperators         #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE RankNTypes            #-}
{-# LANGUAGE RoleAnnotations       #-}
{-# LANGUAGE KindSignatures        #-}
```

Thank you!

Thank you. Questions?

https://github.com/skyvier/gdp-jwt-authorization

References

Noonan, Matt. “Ghosts of Departed Proofs” (2018). URL: https://iohk.io/en/research/library/papers/ghosts-of-departed-proofs-functional-pearls/

Charles, Oliver. “Who Authorized These Ghosts!?” (2019). URL: https://blog.ocharles.org.uk/posts/2019-08-09-who-authorized-these-ghosts.html

Bragilevsky, Vitaly. “Haskell in Depth” (2021)

Applying GDP to authorization: next steps

Improvement: generic proof generators

Domain specific proofs about claims could be built using generic proof constructors such as this.

```
module GDP.JWT where

newtype Claim key claims = Claim Defn

-- | Get a proof that the claim with key @claimKey@ in @claims@
-- has value that equals to @claimValue@.
claimEq
  :: forall claimKey. KnownSymbol claimKey
  => ClaimsSet ~~ claims
  -> ClaimValue ~~ value
  -> Maybe (Proof (Claim claimKey claims == value))
claimEq claims claimValue = undefined
```

Applying GDP to authorization: next steps

Improvement: calculating all privileges of a token at once

Currently, proof of each privilege needs to be calculated separately when it is needed. It would be cool if all privileges described in a JWT could be calculated at once:

```
data Privilege = CanDeleteApps | CanViewApps

calculatePrivileges :: SignedJWT -> Proof [Privilege]
calculatePrivileges = undefined

viewApps ::
  :: (IsMember CanViewApps privileges)
  => Proof privileges
  -> IO Apps
```

Better solution 1: dependent types

We want the GHC type checker to disqualify any code that does not run the necessary authorization checks.

In order to do that, we need to bring data about roles from the term level to the type level.

```
data UserInfo = UserInfo [Text]

fromTerm
  :: UserInfo -> User issuer (roles :: [Symbol])
fromTerm (UserInfo issuer roles) = ...
```

That's what dependent types does for us: it bridges the gap between terms and types.

Better solution 1: dependent types

Unfortunately, Haskell doesn't support dependent types natively yet. It's still possible to get some nice things done using singletons but it's not pretty:

```
type family Contains (r :: Symbol) (rs :: [Symbol]) :: Bool

data SContains :: Symbol -> [Symbol] -> Type where
  SDoes :: (Contains r rs :~: 'True) -> SContains r rs
  SDoesNot :: (Contains r rs :~: 'False) -> SContains r rs

fireMissilesSafe
  :: ( Contains "administrator" roles ~ 'True
     , IsAzure issuer
     )
  => User issuer roles
  -> IO ()
```

Better solution 1: dependent types

Dependent types are a cool tool. They allow:

- Bridging the gap between term and type level via singletons
    - There's a lot of boilerplate involved due to singleton usage
- Writing authorization requirements as function constraints

But they come with a lot of complexity. I find GDP to be just as expressive yet simpler alternative and I'll focus on that for the rest of the talk.

For those who got interested the application of DT in this domain, check out my github repository: https://github.com/skyvier/dt-jwt-authorization.