Title: Tutorial 2B: Introduction to Hyperion; Bootstrapping 3D Ising Island

Speakers: Aike Liu

Collection: Mini-Course of Numerical Conformal Bootstrap

Date: April 25, 2023 - 3:30 PM

URL: https://pirsa.org/23040141

Hyperion Tutorial 1

Bootstrap Mixed Ising

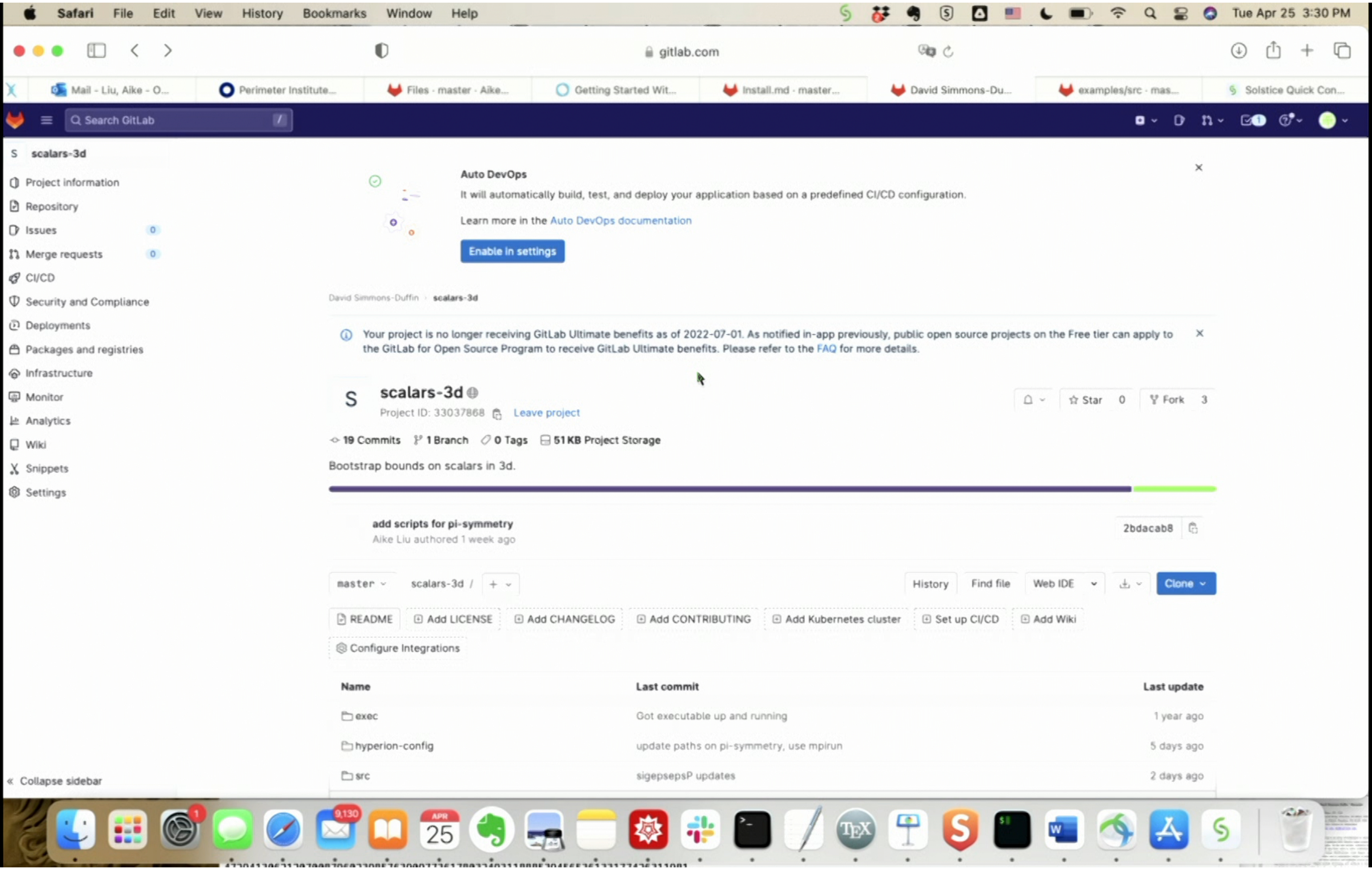
Bootstrap Mini Course: Tutorial 2b

Aike Liu

California Institute of Technology

Caltech PERIMETER INSTITUTE

SIMONS FOUNDATION



Auto DevOps

It will automatically build, test, and deploy your application based on a predefined CI/CD configuration.

Learn more in the Auto DevOps documentation

David Simmons-Duffin › scalars-3d

Your project is no longer receiving GitLab Ultimate benefits as of 2022-07-01. As notified in-app previously, public open source projects on the Free tier can apply to the GitLab for Open Source Program to receive GitLab Ultimate benefits. Please refer to the FAQ for more details.

scalars-3d

Project ID: 33037868 Leave project

19 Commits 1 Branch 0 Tags 51 KB Project Storage

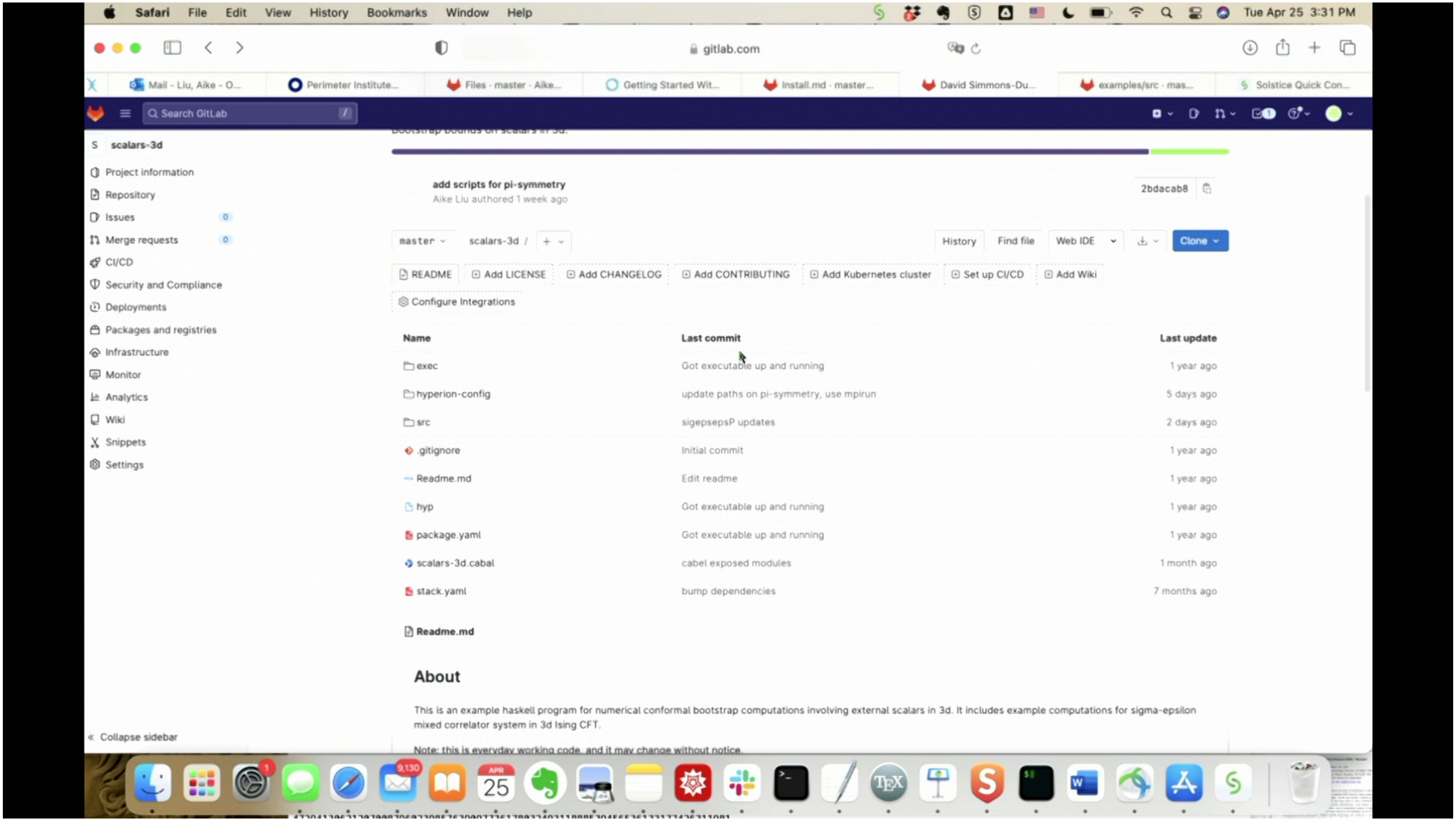
Bootstrap bounds on scalars in 3d.

add scripts for pi-symmetry

Aike Liu authored 1 week ago

2bdacab8

Name	Last commit	Last update
exec	Got executable up and running	1 year ago
hyperion-config	update paths on pi-symmetry, use mpirun	5 days ago
src	sigepsepsP updates	2 days ago



Bootstrap bounds on scalars in 3d.

add scripts for pi-symmetry

Aike Liu authored 1 week ago

2bdacab8

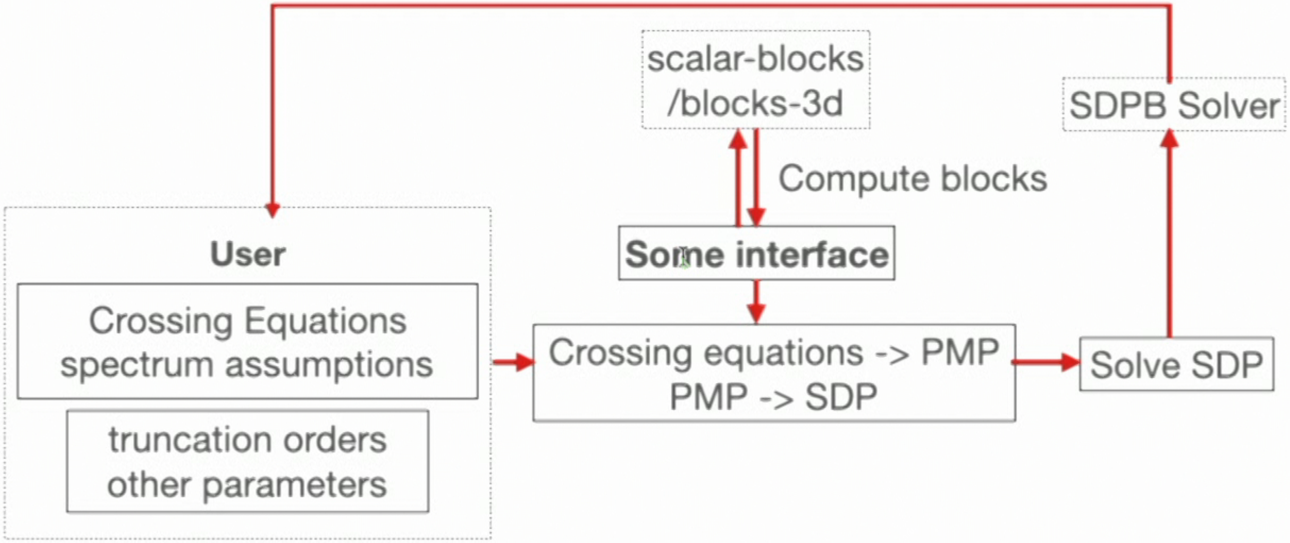
Name	Last commit	Last update
exec	Got executable up and running	1 year ago
hyperion-config	update paths on pi-symmetry, use mpirun	5 days ago
src	sigepsepsP updates	2 days ago
.gitignore	Initial commit	1 year ago
Readme.md	Edit readme	1 year ago
hyp	Got executable up and running	1 year ago
package.yaml	Got executable up and running	1 year ago
scalars-3d.cabal	cabel exposed modules	1 month ago
stack.yaml	bump dependencies	7 months ago

Readme.md

About

This is an example haskell program for numerical conformal bootstrap computations involving external scalars in 3d. It includes example computations for sigma-epsilon mixed correlator system in 3d Ising CFT.

Note: this is everyday working code, and it may change without notice.



User

Crossing Equations
spectrum assumptions

truncation orders
other parameters

scalar-blocks
/blocks-3d

Compute blocks

Some interface

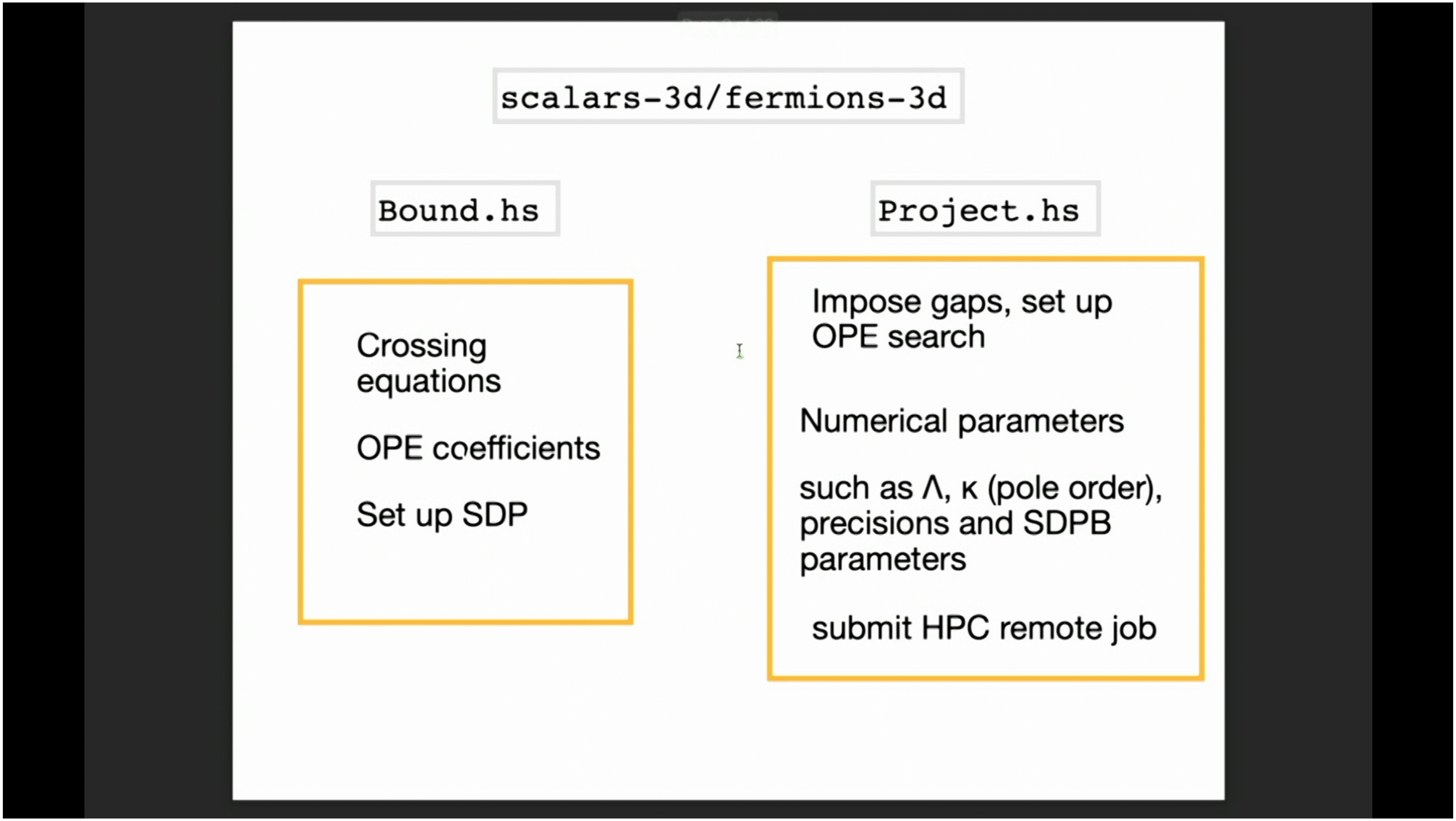
Crossing equations -> PMP
PMP -> SDP

Solve SDP

SDPB Solver

PMP: Positive Matrix Programming

SDP: Semidefinite Programming



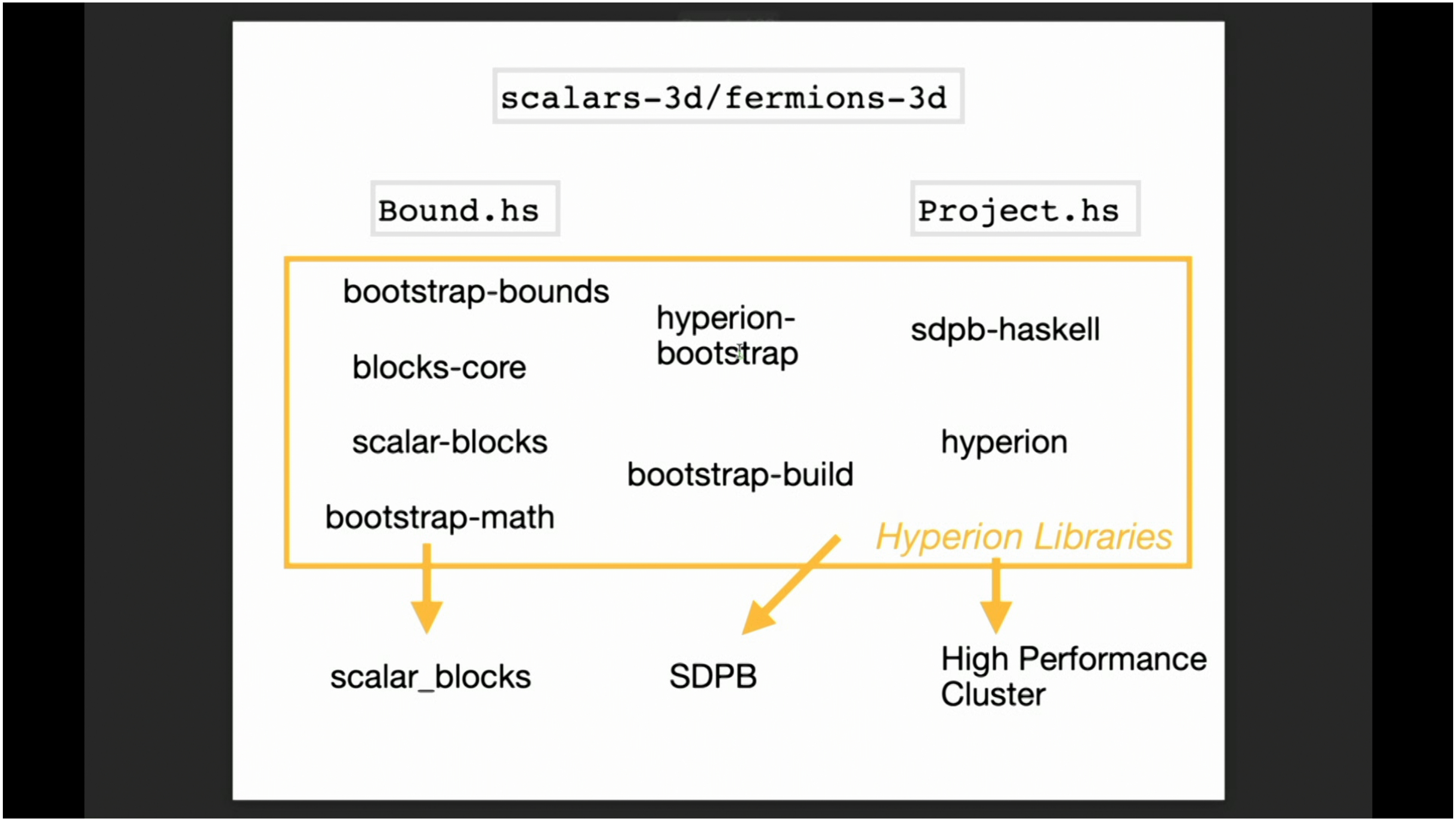
`scalars-3d/fermions-3d`

`Bound.hs`

- Crossing equations
- OPE coefficients
- Set up SDP

`Project.hs`

- Impose gaps, set up OPE search
- Numerical parameters
- such as Λ, κ (pole order), precisions and SDPB parameters
- submit HPC remote job



`scalars-3d/fermions-3d`

`Bound.hs`

`Project.hs`

bootstrap-bounds

blocks-core

scalar-blocks

bootstrap-math

hyperion-bootstrap

bootstrap-build

sdpb-haskell

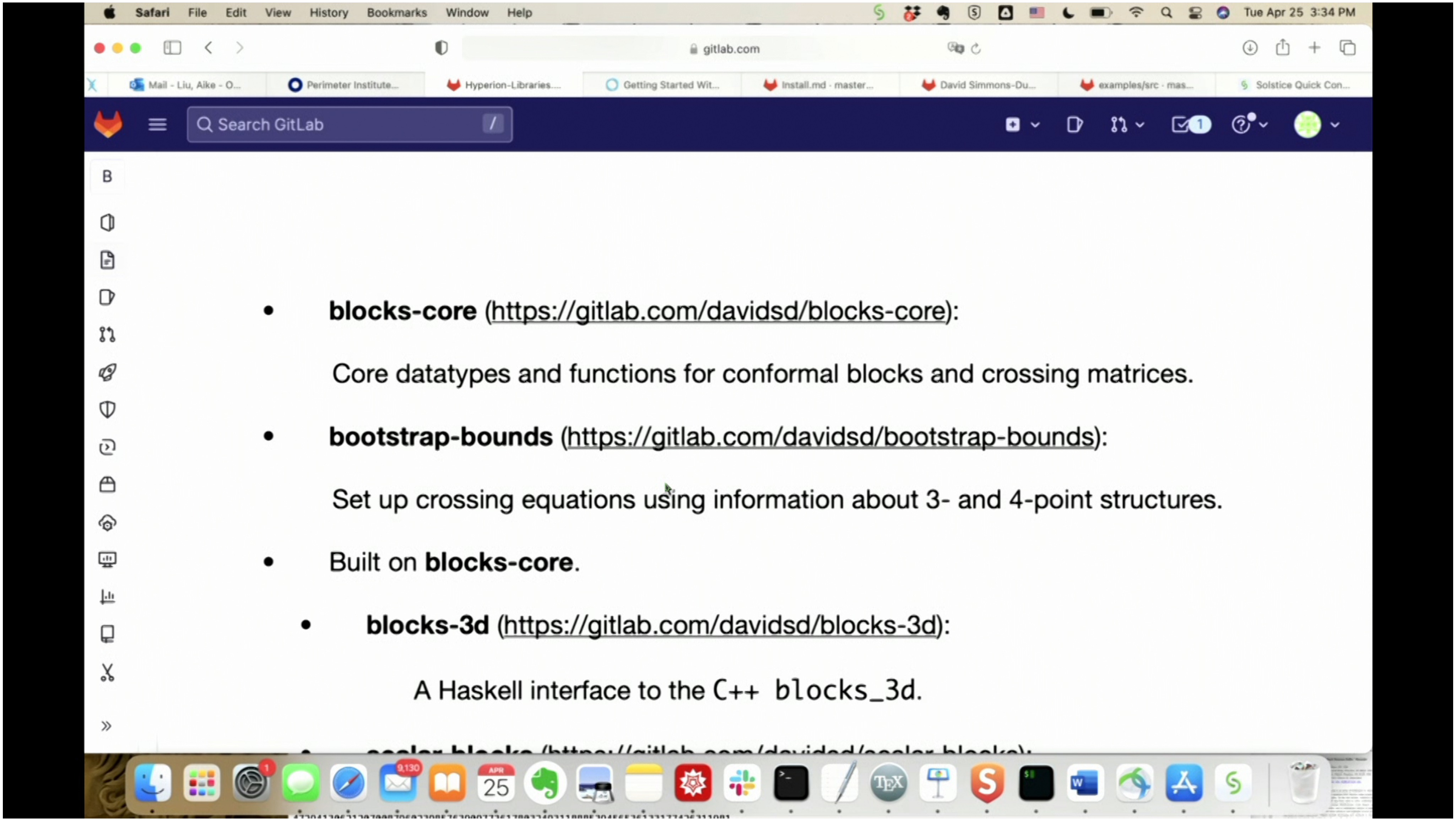
hyperion

Hyperion Libraries

scalar_blocks

SDPB

High Performance Cluster



- **blocks-core** (https://gitlab.com/davidsd/blocks-core):

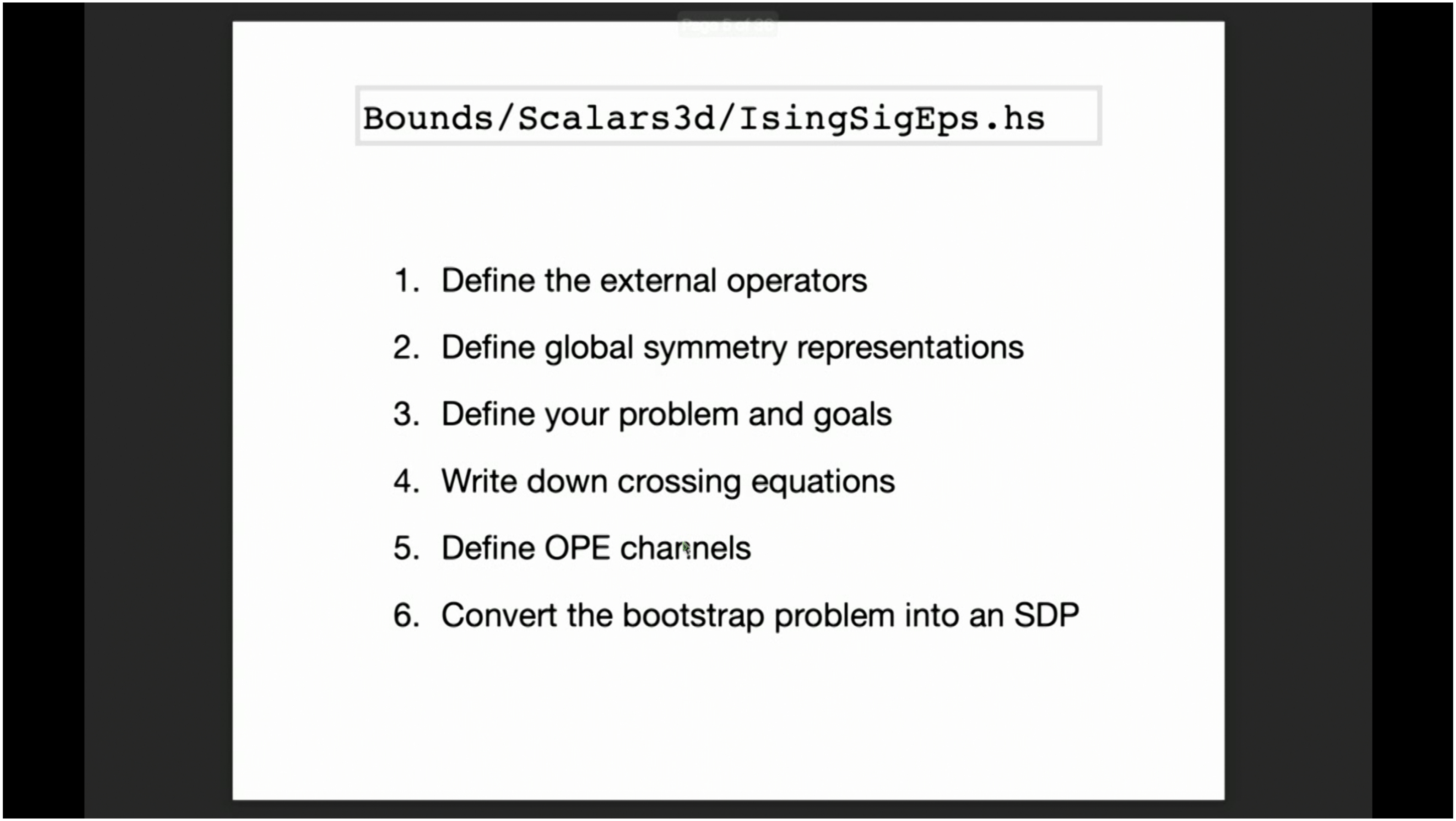
  Core datatypes and functions for conformal blocks and crossing matrices.

- **bootstrap-bounds** (https://gitlab.com/davidsd/bootstrap-bounds):

  Set up crossing equations using information about 3- and 4-point structures.

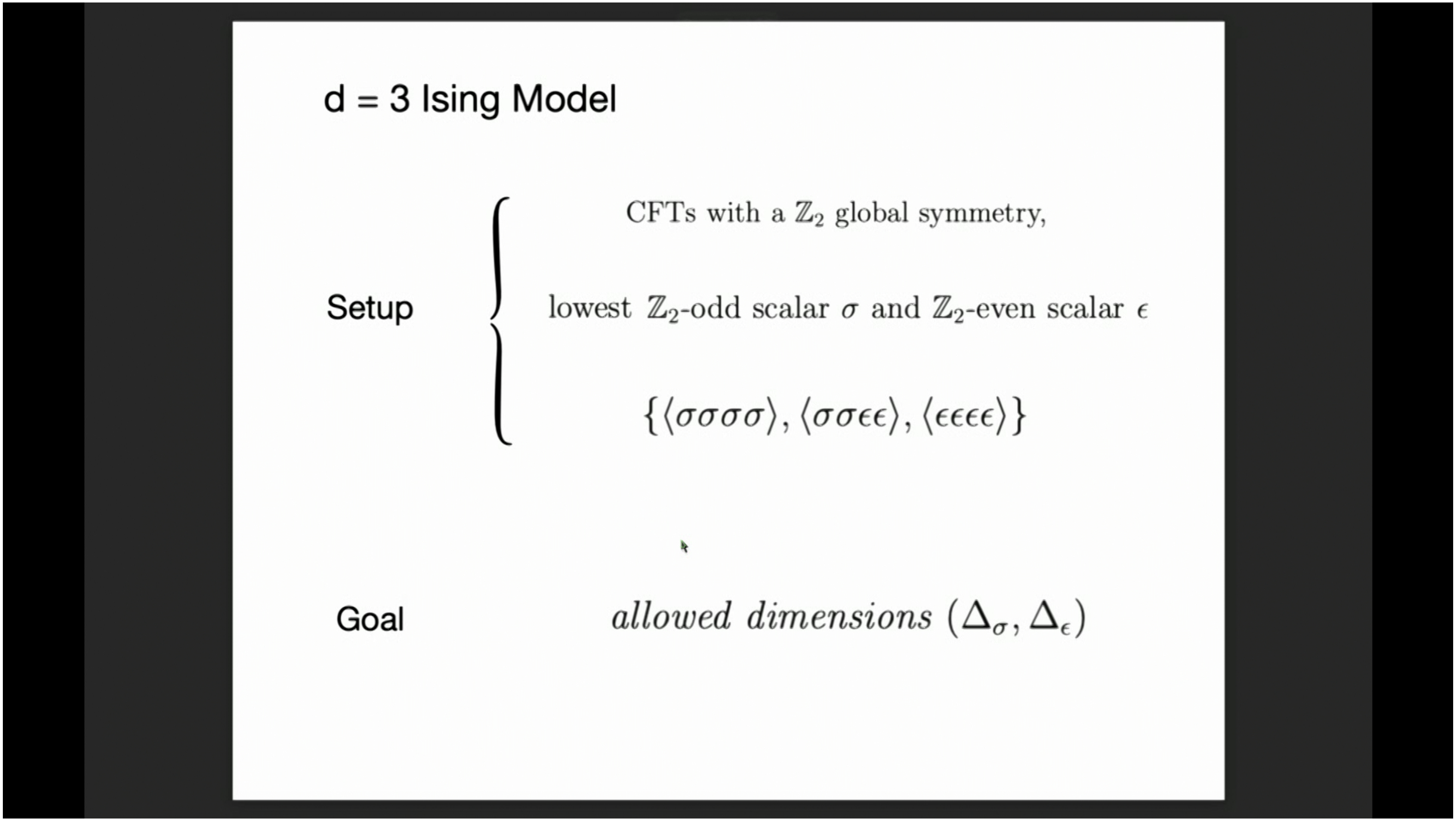
- Built on **blocks-core**.
  - **blocks-3d** (https://gitlab.com/davidsd/blocks-3d):

    A Haskell interface to the C++ `blocks_3d`.



`Bounds/Scalars3d/IsingSigEps.hs`

1. Define the external operators
2. Define global symmetry representations
3. Define your problem and goals
4. Write down crossing equations
5. Define OPE channels
6. Convert the bootstrap problem into an SDP



d = 3 Ising Model

Setup

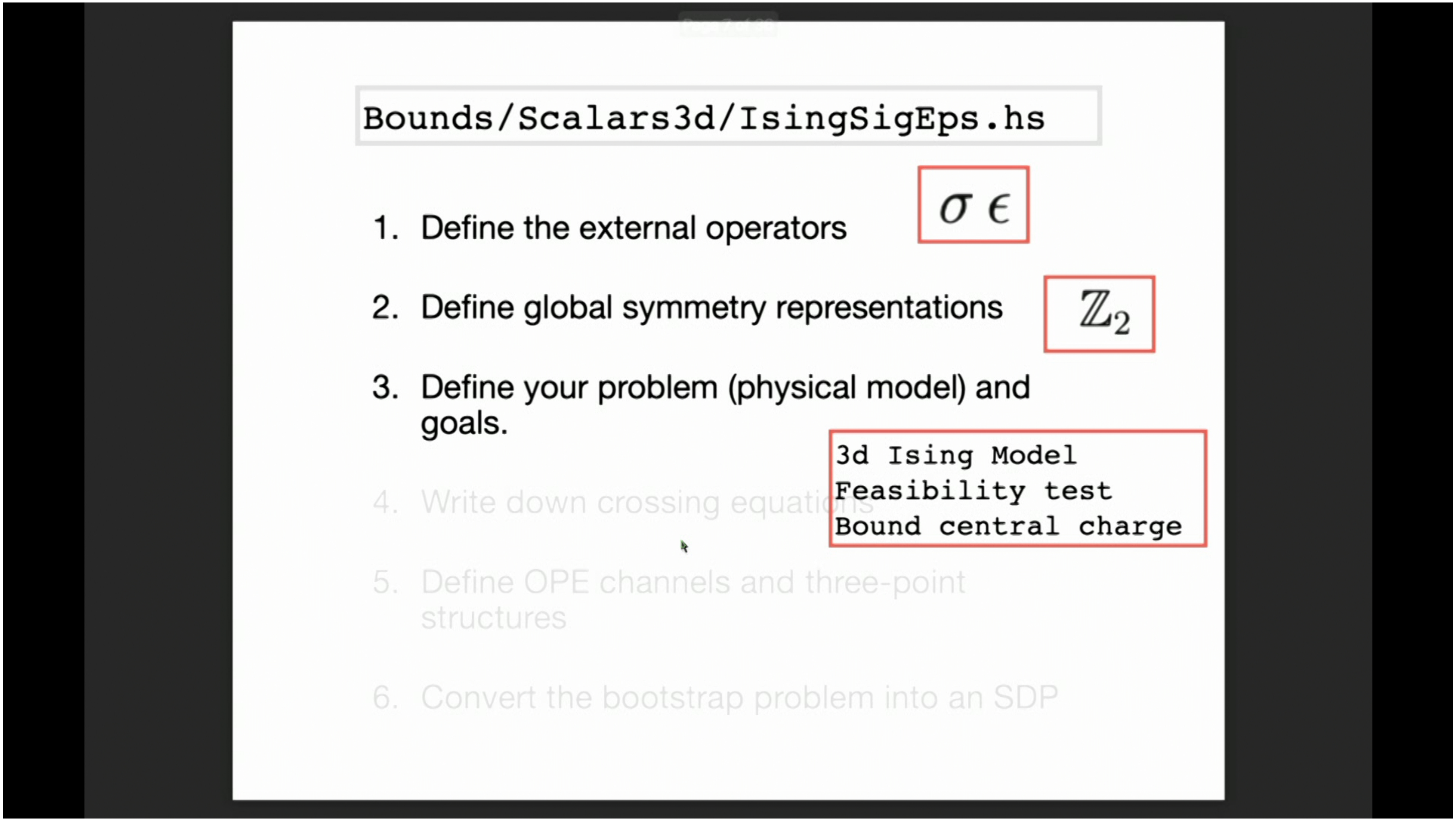
CFTs with a $\mathbb{Z}_2$ global symmetry,

lowest $\mathbb{Z}_2$-odd scalar σ and $\mathbb{Z}_2$-even scalar ϵ

$$\{\langle\sigma\sigma\sigma\sigma\rangle, \langle\sigma\sigma\epsilon\epsilon\rangle, \langle\epsilon\epsilon\epsilon\epsilon\rangle\}$$

Goal

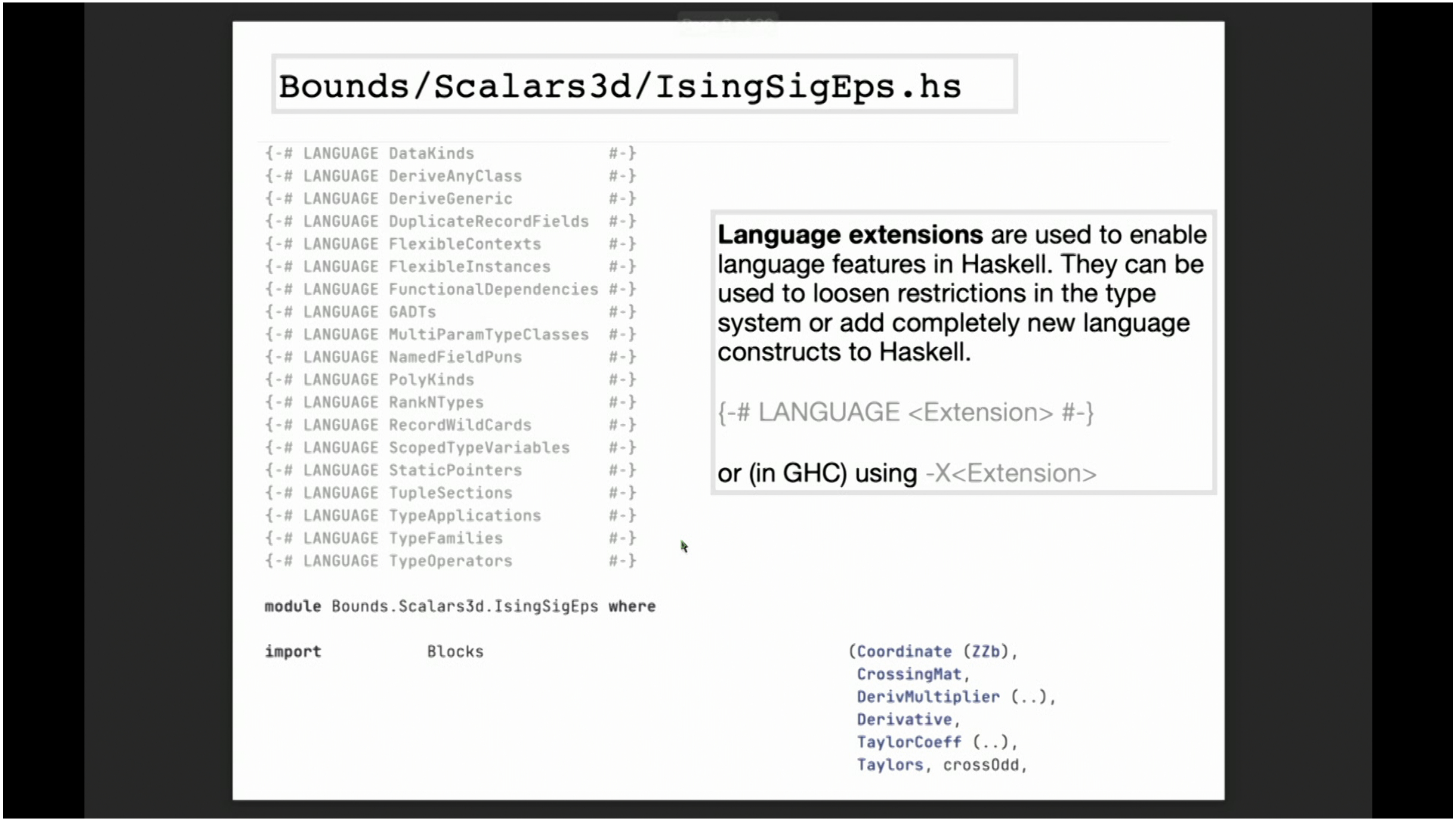
allowed dimensions $(\Delta_\sigma, \Delta_\epsilon)$



`Bounds/Scalars3d/IsingSigEps.hs`

1. Define the external operators $\sigma \; \epsilon$
2. Define global symmetry representations $\mathbb{Z}_2$
3. Define your problem (physical model) and goals.
4. Write down crossing equations
5. Define OPE channels and three-point structures
6. Convert the bootstrap problem into an SDP

```
3d Ising Model
Feasibility test
Bound central charge
```



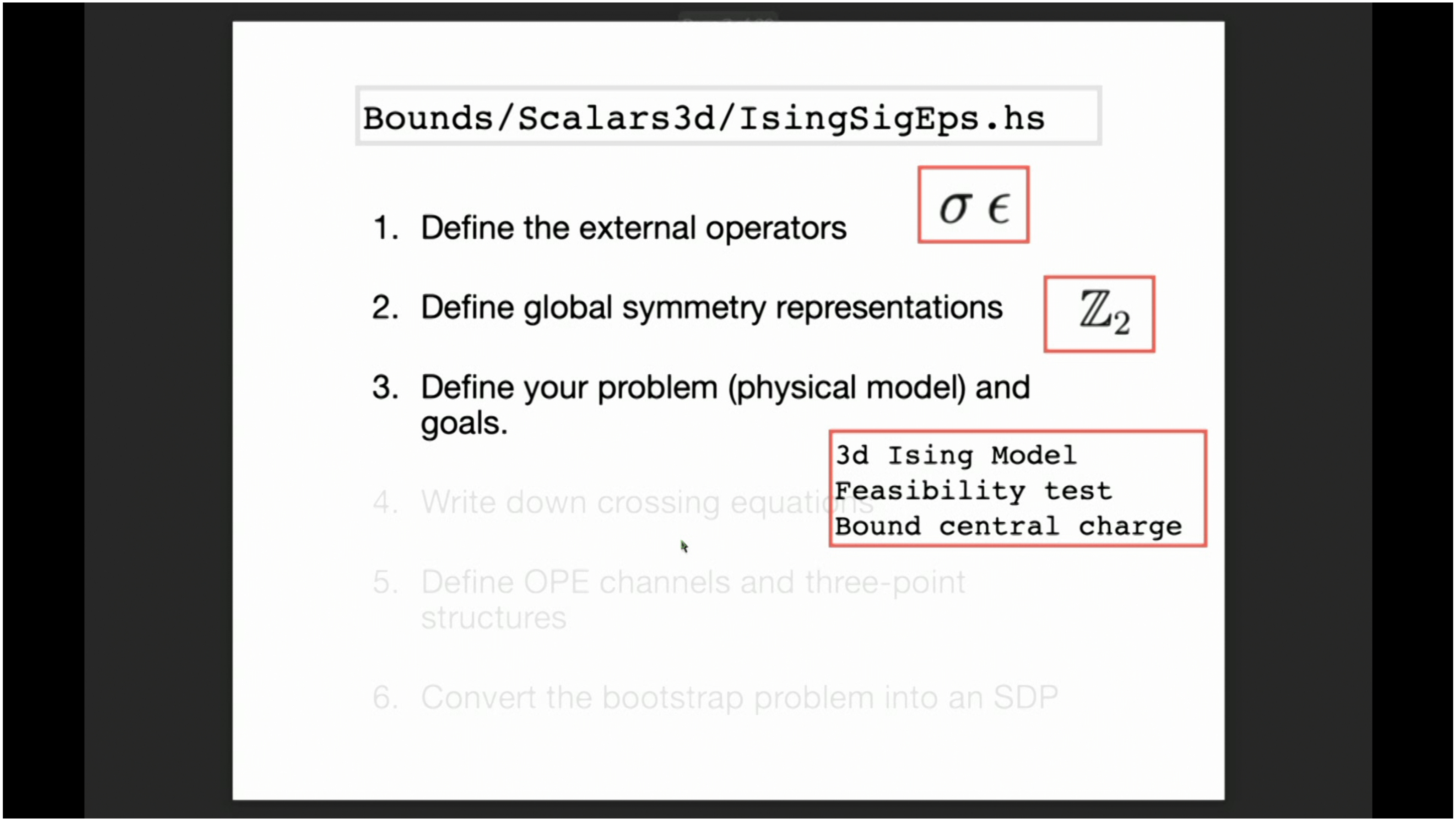
Bounds/Scalars3d/IsingSigEps.hs

```
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE DeriveAnyClass         #-}
{-# LANGUAGE DeriveGeneric          #-}
{-# LANGUAGE DuplicateRecordFields  #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE GADTs                  #-}
{-# LANGUAGE MultiParamTypeClasses  #-}
{-# LANGUAGE NamedFieldPuns         #-}
{-# LANGUAGE PolyKinds              #-}
{-# LANGUAGE RankNTypes             #-}
{-# LANGUAGE RecordWildCards        #-}
{-# LANGUAGE ScopedTypeVariables    #-}
{-# LANGUAGE StaticPointers         #-}
{-# LANGUAGE TupleSections          #-}
{-# LANGUAGE TypeApplications       #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE TypeOperators          #-}

module Bounds.Scalars3d.IsingSigEps where

import           Blocks                                       (Coordinate (ZZb),
                                                               CrossingMat,
                                                               DerivMultiplier (..),
                                                               Derivative,
                                                               TaylorCoeff (..),
                                                               Taylors, crossOdd,
```

Language extensions are used to enable language features in Haskell. They can be used to loosen restrictions in the type system or add completely new language constructs to Haskell.

`{-# LANGUAGE <Extension> #-}`

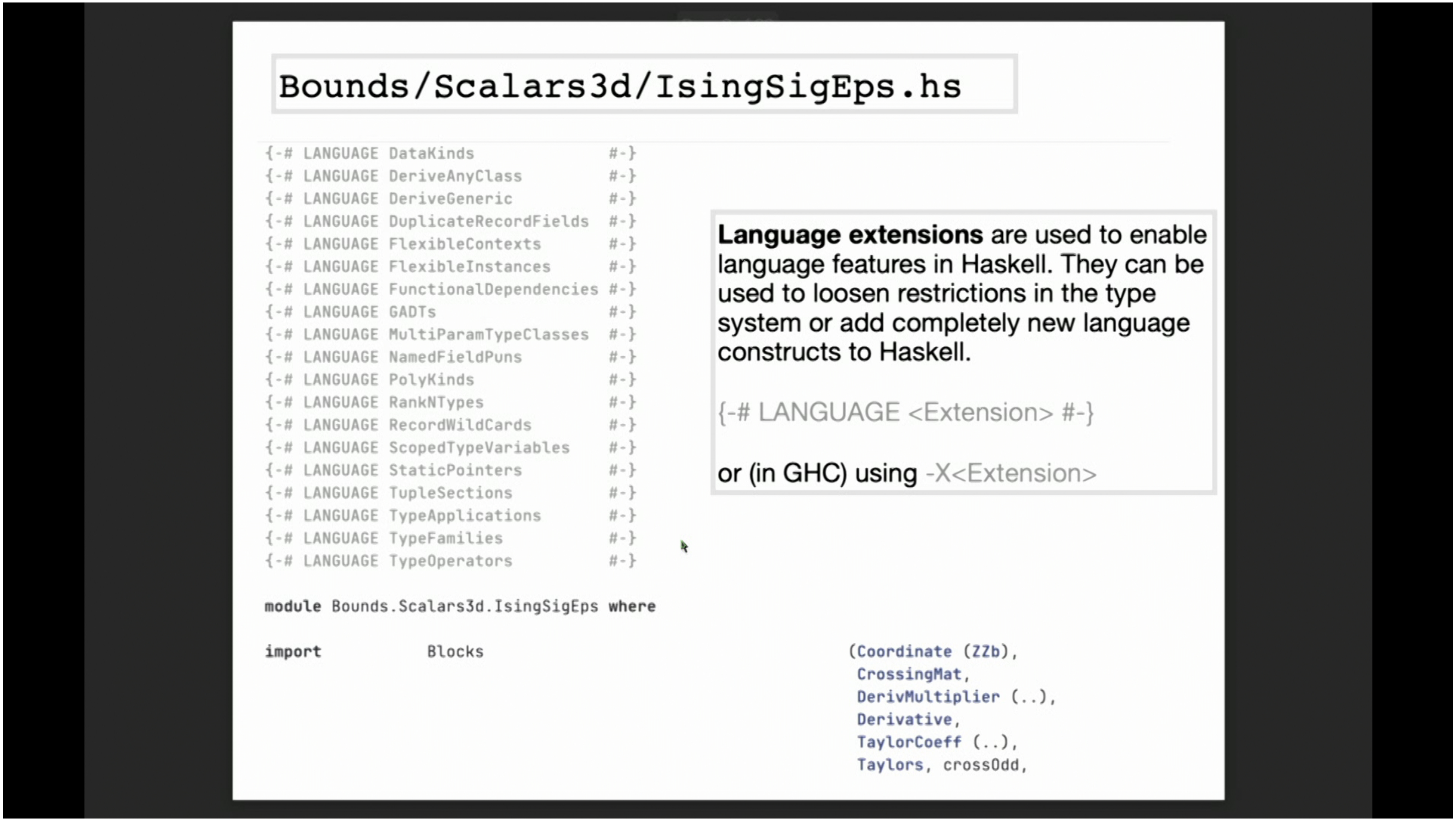
or (in GHC) using `-X<Extension>`



`Bounds/Scalars3d/IsingSigEps.hs`

1. Define the external operators $\boxed{\sigma\ \epsilon}$
2. Define global symmetry representations $\boxed{\mathbb{Z}_2}$
3. Define your problem (physical model) and goals.
4. Write down crossing equations
5. Define OPE channels and three-point structures
6. Convert the bootstrap problem into an SDP

```
3d Ising Model
Feasibility test
Bound central charge
```



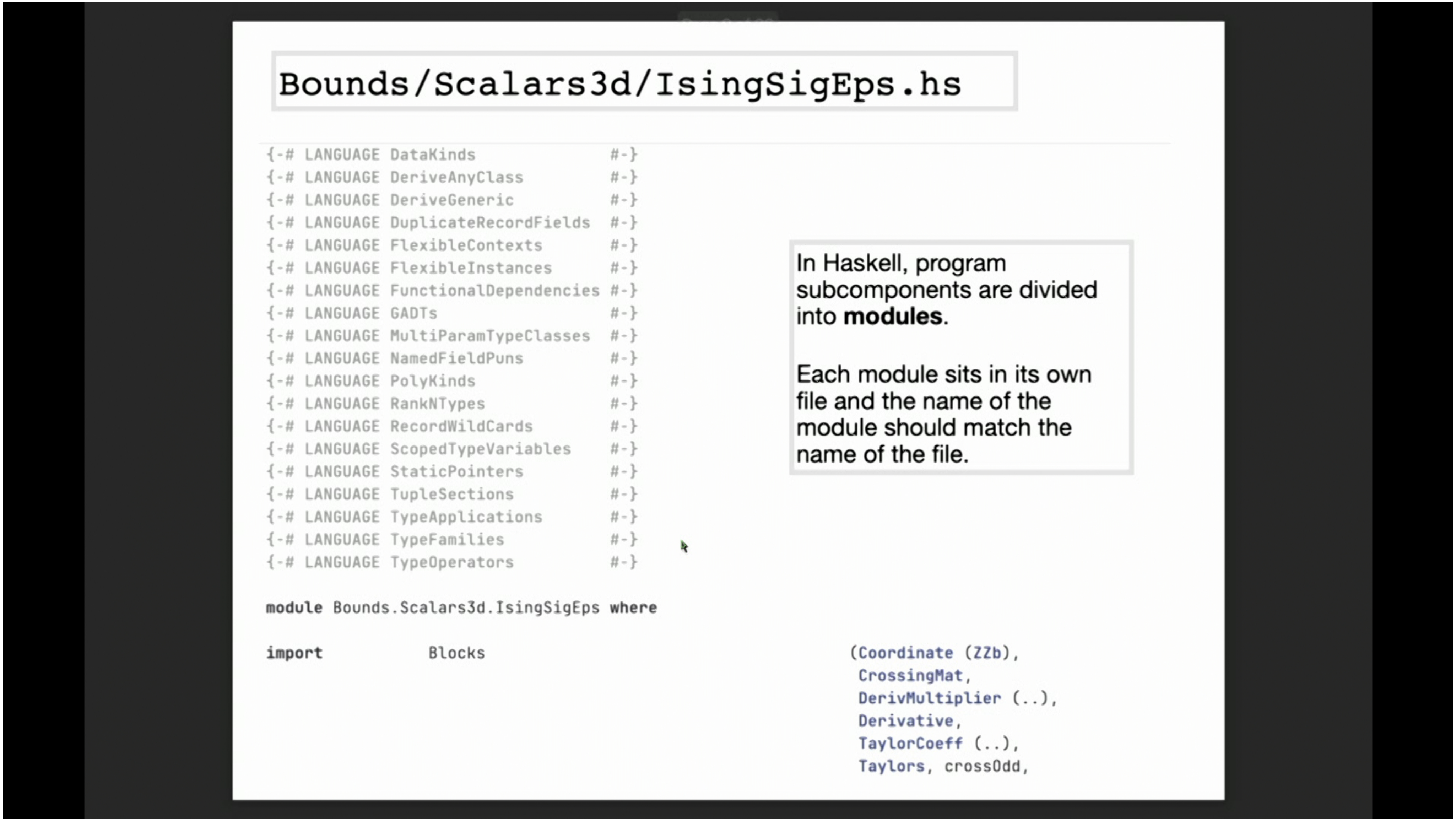
Bounds/Scalars3d/IsingSigEps.hs

```haskell
{-# LANGUAGE DataKinds                #-}
{-# LANGUAGE DeriveAnyClass           #-}
{-# LANGUAGE DeriveGeneric            #-}
{-# LANGUAGE DuplicateRecordFields    #-}
{-# LANGUAGE FlexibleContexts         #-}
{-# LANGUAGE FlexibleInstances        #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE GADTs                    #-}
{-# LANGUAGE MultiParamTypeClasses    #-}
{-# LANGUAGE NamedFieldPuns           #-}
{-# LANGUAGE PolyKinds                #-}
{-# LANGUAGE RankNTypes               #-}
{-# LANGUAGE RecordWildCards          #-}
{-# LANGUAGE ScopedTypeVariables      #-}
{-# LANGUAGE StaticPointers           #-}
{-# LANGUAGE TupleSections            #-}
{-# LANGUAGE TypeApplications         #-}
{-# LANGUAGE TypeFamilies             #-}
{-# LANGUAGE TypeOperators            #-}

module Bounds.Scalars3d.IsingSigEps where

import           Blocks                                  (Coordinate (ZZb),
                                                           CrossingMat,
                                                           DerivMultiplier (..),
                                                           Derivative,
                                                           TaylorCoeff (..),
                                                           Taylors, crossOdd,
```

Language extensions are used to enable language features in Haskell. They can be used to loosen restrictions in the type system or add completely new language constructs to Haskell.

{-# LANGUAGE <Extension> #-}

or (in GHC) using -X<Extension>

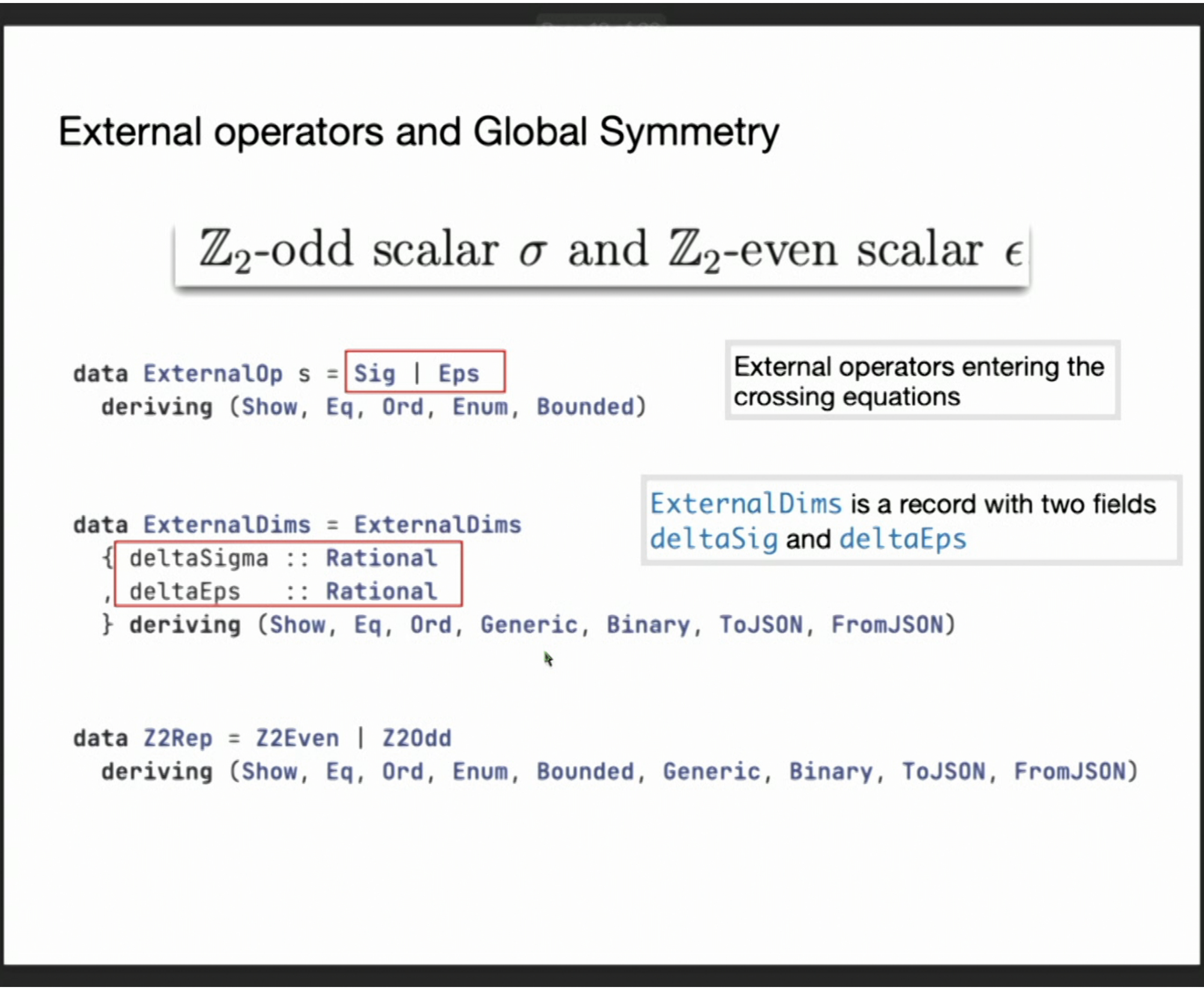


Bounds/Scalars3d/IsingSigEps.hs

```
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE DeriveAnyClass         #-}
{-# LANGUAGE DeriveGeneric          #-}
{-# LANGUAGE DuplicateRecordFields  #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE GADTs                  #-}
{-# LANGUAGE MultiParamTypeClasses  #-}
{-# LANGUAGE NamedFieldPuns         #-}
{-# LANGUAGE PolyKinds              #-}
{-# LANGUAGE RankNTypes             #-}
{-# LANGUAGE RecordWildCards        #-}
{-# LANGUAGE ScopedTypeVariables    #-}
{-# LANGUAGE StaticPointers         #-}
{-# LANGUAGE TupleSections          #-}
{-# LANGUAGE TypeApplications       #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE TypeOperators          #-}

module Bounds.Scalars3d.IsingSigEps where

import           Blocks                                    (Coordinate (ZZb),
                                                            CrossingMat,
                                                            DerivMultiplier (..),
                                                            Derivative,
                                                            TaylorCoeff (..),
                                                            Taylors, crossOdd,
```

In Haskell, program subcomponents are divided into **modules**.

Each module sits in its own file and the name of the module should match the name of the file.



External operators and Global Symmetry

$$\mathbb{Z}_2\text{-odd scalar } \sigma \text{ and } \mathbb{Z}_2\text{-even scalar } \epsilon$$

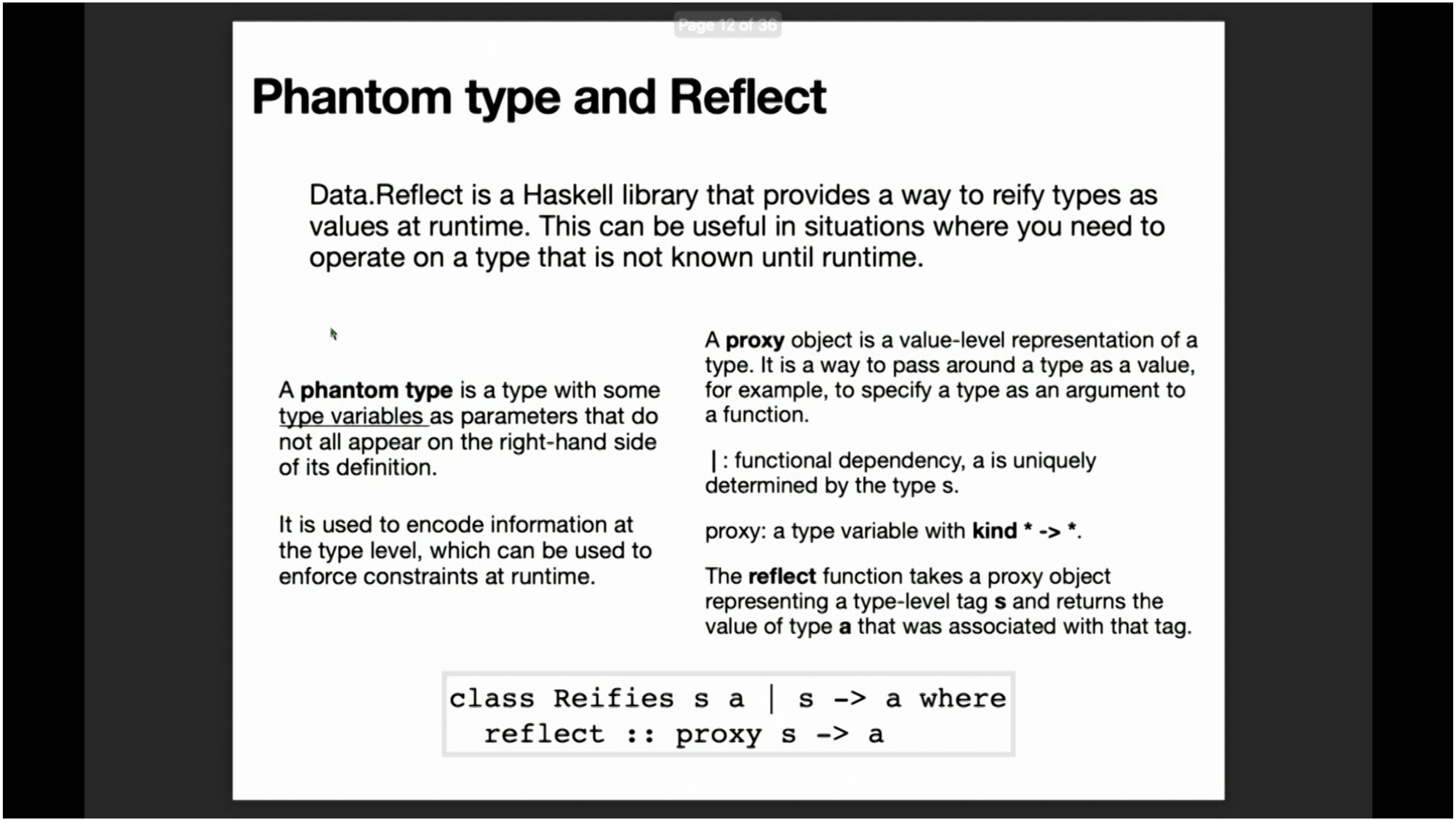
```
data ExternalOp s = Sig | Eps
  deriving (Show, Eq, Ord, Enum, Bounded)
```

External operators entering the crossing equations

```
data ExternalDims = ExternalDims
  { deltaSigma :: Rational
  , deltaEps   :: Rational
  } deriving (Show, Eq, Ord, Generic, Binary, ToJSON, FromJSON)
```

`ExternalDims` is a record with two fields `deltaSig` and `deltaEps`

```
data Z2Rep = Z2Even | Z2Odd
  deriving (Show, Eq, Ord, Enum, Bounded, Generic, Binary, ToJSON, FromJSON)
```



Phantom type and Reflect

Data.Reflect is a Haskell library that provides a way to reify types as values at runtime. This can be useful in situations where you need to operate on a type that is not known until runtime.

A **phantom type** is a type with some type variables as parameters that do not all appear on the right-hand side of its definition.

It is used to encode information at the type level, which can be used to enforce constraints at runtime.

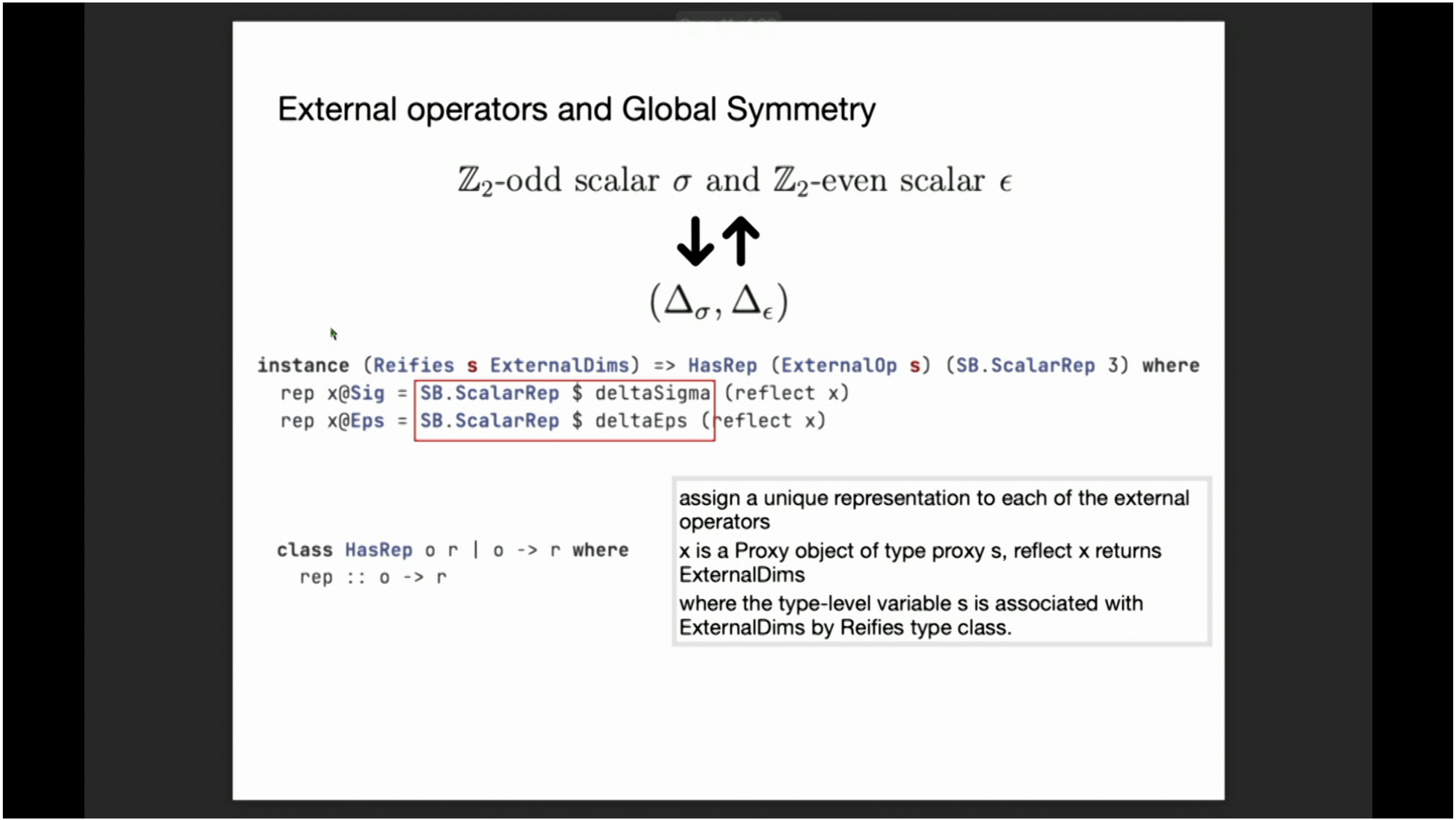
A **proxy** object is a value-level representation of a type. It is a way to pass around a type as a value, for example, to specify a type as an argument to a function.

| : functional dependency, a is uniquely determined by the type s.

proxy: a type variable with **kind * -> ***.

The **reflect** function takes a proxy object representing a type-level tag **s** and returns the value of type **a** that was associated with that tag.

```
class Reifies s a | s -> a where
  reflect :: proxy s -> a
```



External operators and Global Symmetry

$$\mathbb{Z}_2\text{-odd scalar } \sigma \text{ and } \mathbb{Z}_2\text{-even scalar } \epsilon$$

$$\downarrow\uparrow$$

$$(\Delta_\sigma, \Delta_\epsilon)$$

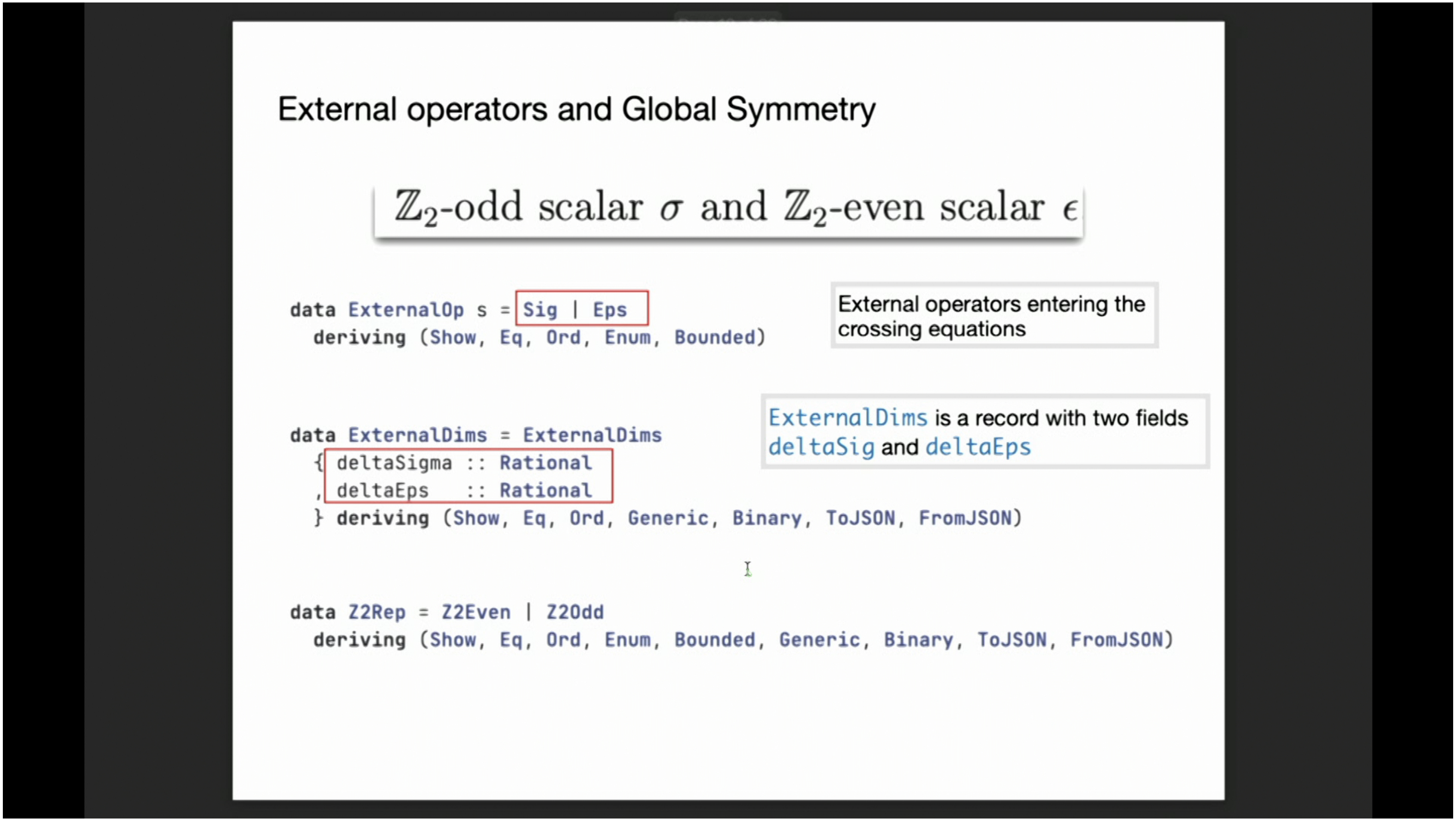
```
instance (Reifies s ExternalDims) => HasRep (ExternalOp s) (SB.ScalarRep 3) where
  rep x@Sig = SB.ScalarRep $ deltaSigma (reflect x)
  rep x@Eps = SB.ScalarRep $ deltaEps (reflect x)
```

```
class HasRep o r | o -> r where
  rep :: o -> r
```

assign a unique representation to each of the external operators

x is a Proxy object of type proxy s, reflect x returns ExternalDims

where the type-level variable s is associated with ExternalDims by Reifies type class.



External operators and Global Symmetry

$\mathbb{Z}_2$-odd scalar σ and $\mathbb{Z}_2$-even scalar ϵ

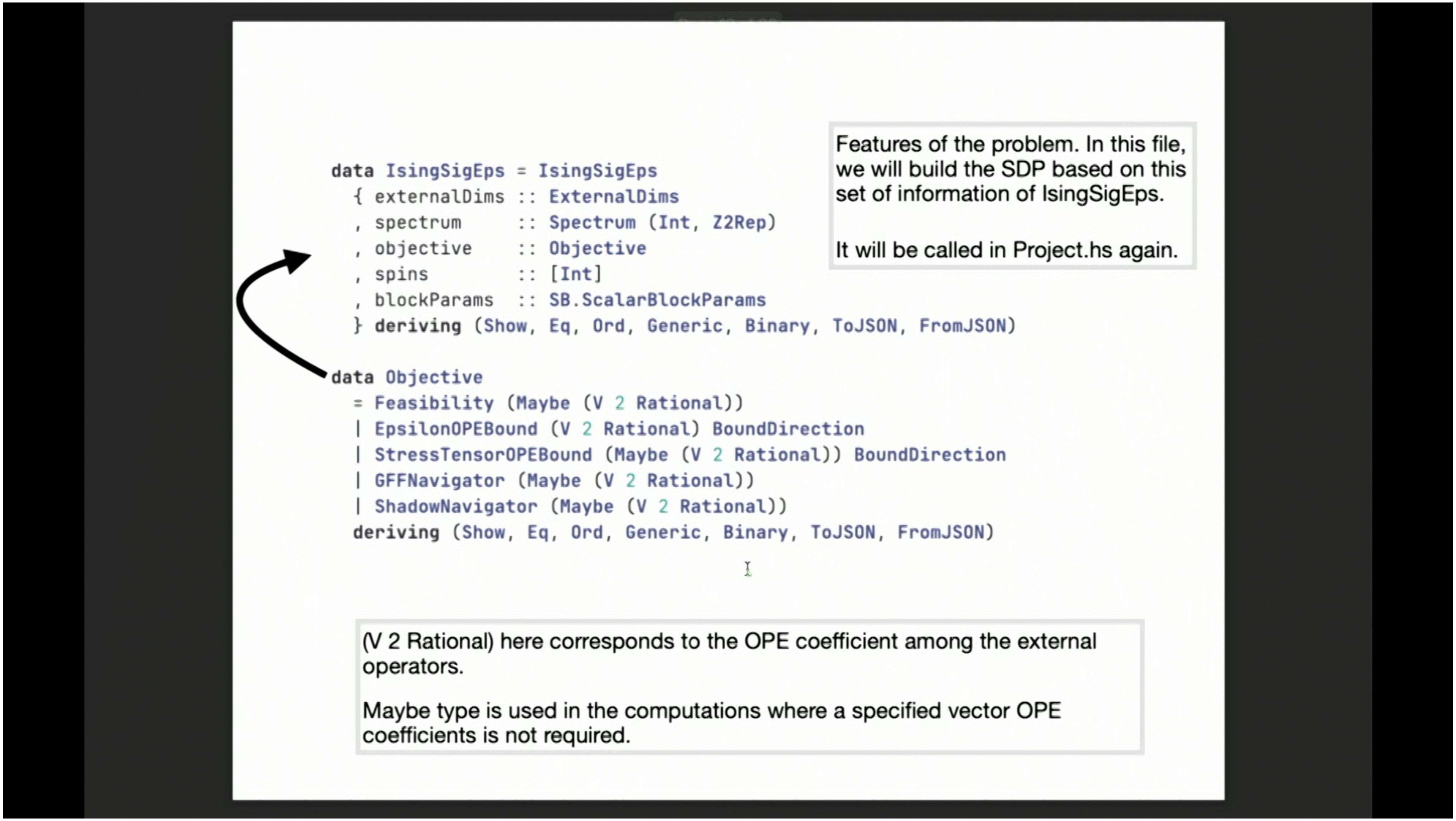
```
data ExternalOp s = Sig | Eps
  deriving (Show, Eq, Ord, Enum, Bounded)
```

External operators entering the crossing equations

```
data ExternalDims = ExternalDims
  { deltaSigma :: Rational
  , deltaEps   :: Rational
  } deriving (Show, Eq, Ord, Generic, Binary, ToJSON, FromJSON)
```

`ExternalDims` is a record with two fields `deltaSig` and `deltaEps`

```
data Z2Rep = Z2Even | Z2Odd
  deriving (Show, Eq, Ord, Enum, Bounded, Generic, Binary, ToJSON, FromJSON)
```



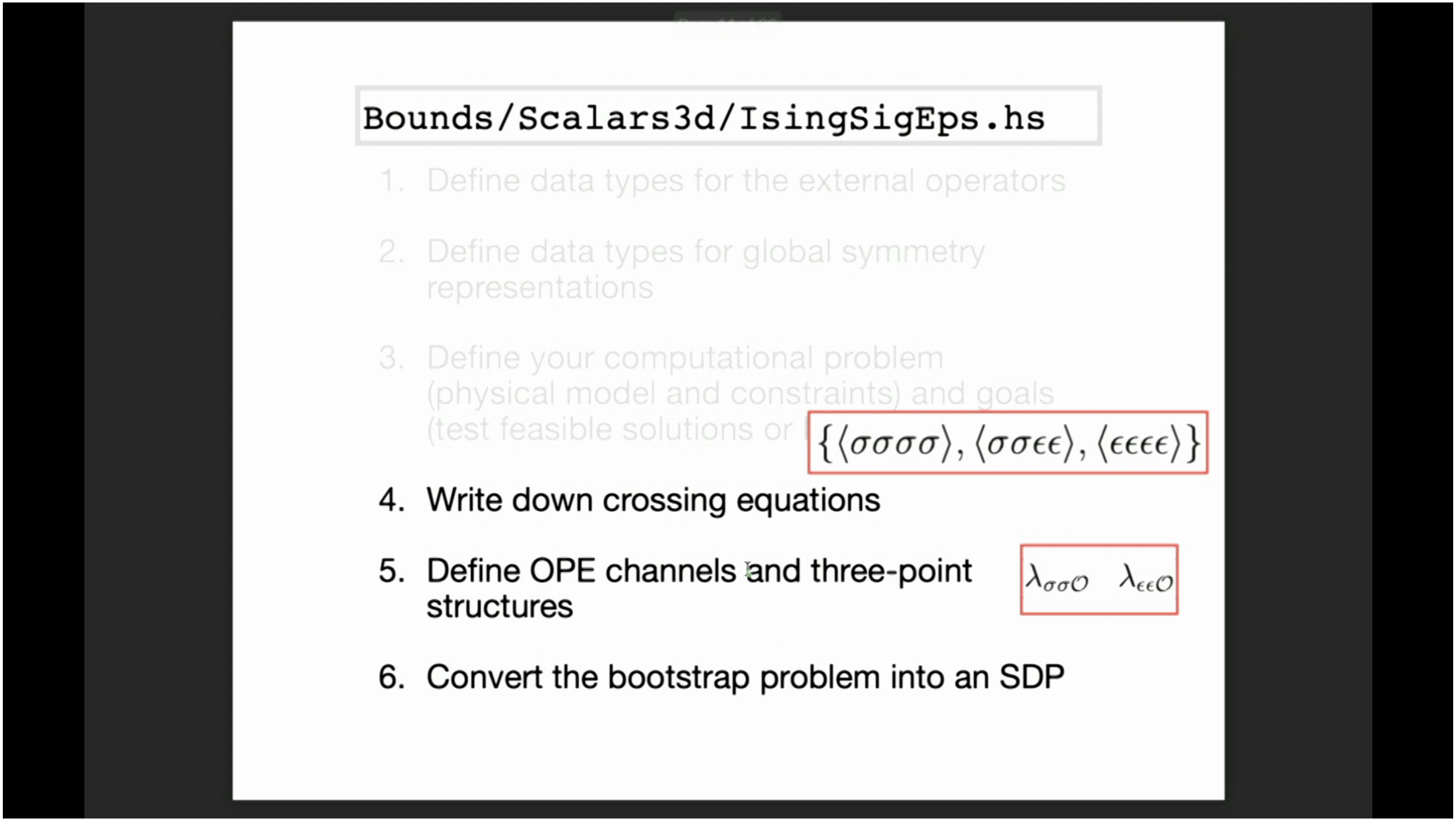
Features of the problem. In this file, we will build the SDP based on this set of information of IsingSigEps.

It will be called in Project.hs again.

```
data IsingSigEps = IsingSigEps
  { externalDims :: ExternalDims
  , spectrum     :: Spectrum (Int, Z2Rep)
  , objective    :: Objective
  , spins        :: [Int]
  , blockParams  :: SB.ScalarBlockParams
  } deriving (Show, Eq, Ord, Generic, Binary, ToJSON, FromJSON)

data Objective
  = Feasibility (Maybe (V 2 Rational))
  | EpsilonOPEBound (V 2 Rational) BoundDirection
  | StressTensorOPEBound (Maybe (V 2 Rational)) BoundDirection
  | GFFNavigator (Maybe (V 2 Rational))
  | ShadowNavigator (Maybe (V 2 Rational))
  deriving (Show, Eq, Ord, Generic, Binary, ToJSON, FromJSON)
```

(V 2 Rational) here corresponds to the OPE coefficient among the external operators.

Maybe type is used in the computations where a specified vector OPE coefficients is not required.



`Bounds/Scalars3d/IsingSigEps.hs`

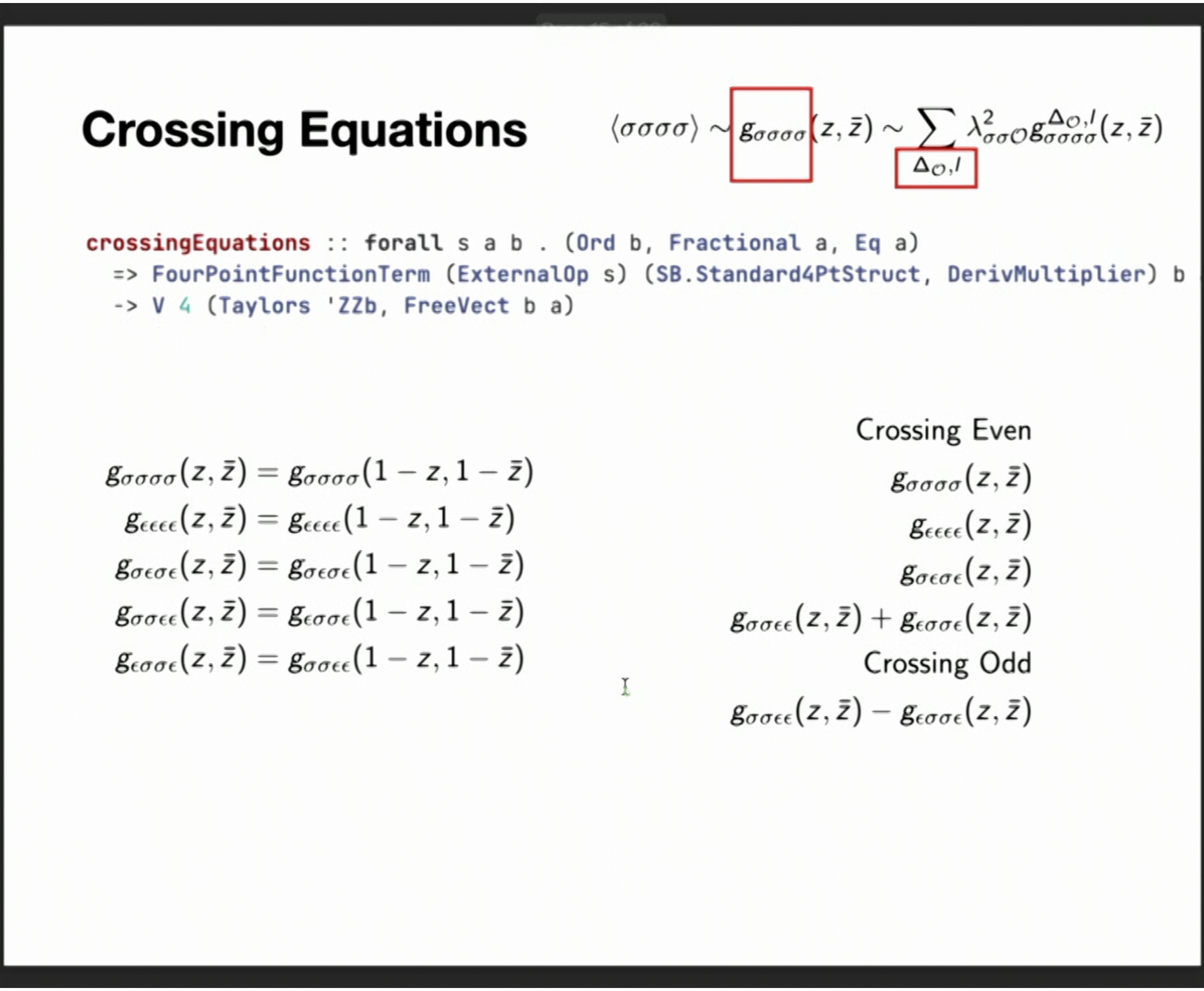
1. Define data types for the external operators
2. Define data types for global symmetry representations
3. Define your computational problem (physical model and constraints) and goals (test feasible solutions or

$$\{\langle\sigma\sigma\sigma\sigma\rangle, \langle\sigma\sigma\epsilon\epsilon\rangle, \langle\epsilon\epsilon\epsilon\epsilon\rangle\}$$

4. **Write down crossing equations**
5. **Define OPE channels and three-point structures**

$\lambda_{\sigma\sigma\mathcal{O}}$ $\lambda_{\epsilon\epsilon\mathcal{O}}$

6. **Convert the bootstrap problem into an SDP**



Crossing Equations

$$\langle\sigma\sigma\sigma\sigma\rangle \sim \boxed{g_{\sigma\sigma\sigma\sigma}}(z,\bar{z}) \sim \sum_{\boxed{\Delta_{\mathcal{O}},l}} \lambda^2_{\sigma\sigma\mathcal{O}} g^{\Delta_{\mathcal{O}},l}_{\sigma\sigma\sigma\sigma}(z,\bar{z})$$

```
crossingEquations :: forall s a b . (Ord b, Fractional a, Eq a)
  => FourPointFunctionTerm (ExternalOp s) (SB.Standard4PtStruct, DerivMultiplier) b
  -> V 4 (Taylors 'ZZb, FreeVect b a)
```

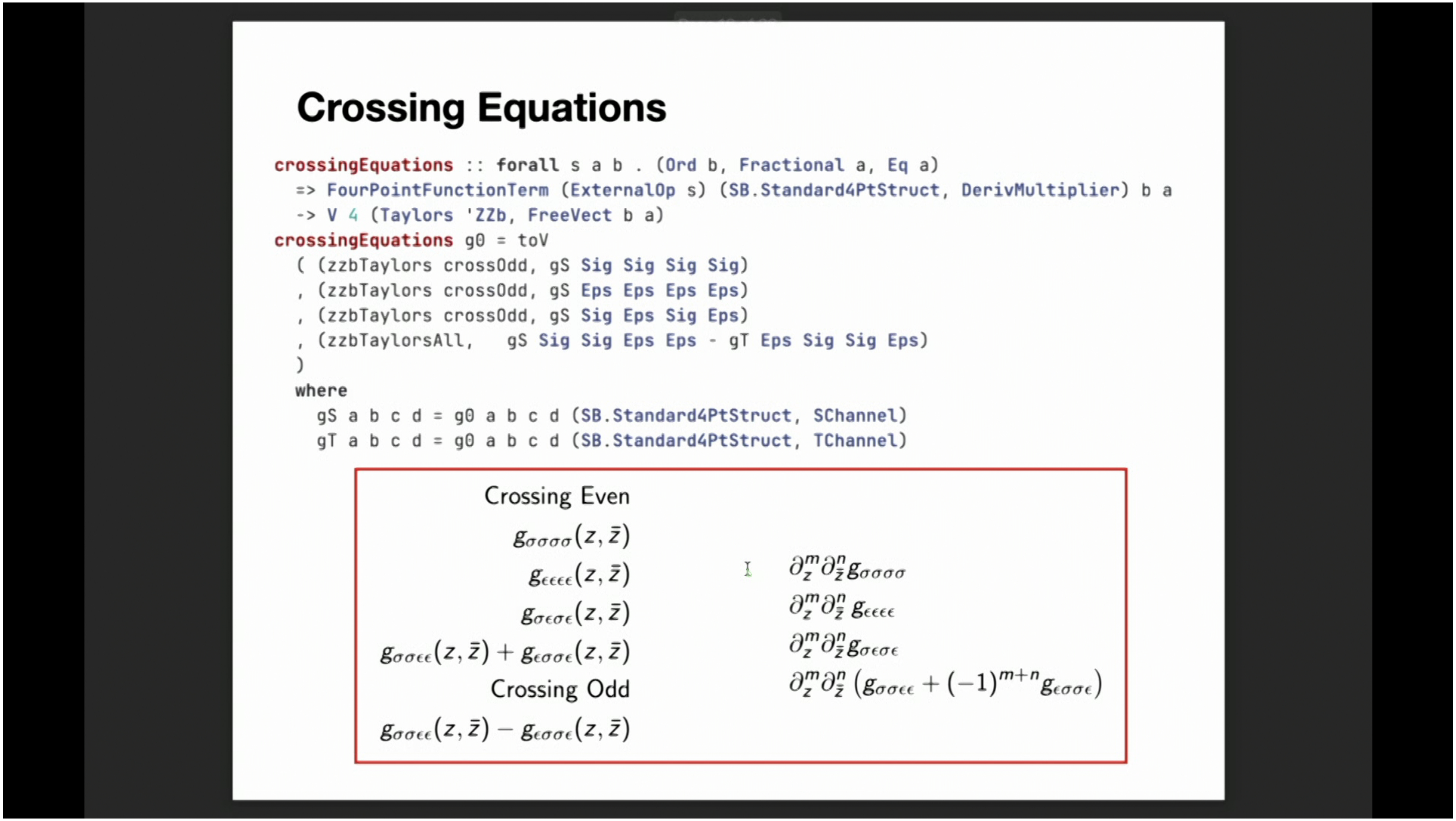
$$g_{\sigma\sigma\sigma\sigma}(z,\bar{z}) = g_{\sigma\sigma\sigma\sigma}(1-z,1-\bar{z})$$
$$g_{\epsilon\epsilon\epsilon\epsilon}(z,\bar{z}) = g_{\epsilon\epsilon\epsilon\epsilon}(1-z,1-\bar{z})$$
$$g_{\sigma\epsilon\sigma\epsilon}(z,\bar{z}) = g_{\sigma\epsilon\sigma\epsilon}(1-z,1-\bar{z})$$
$$g_{\sigma\sigma\epsilon\epsilon}(z,\bar{z}) = g_{\epsilon\sigma\sigma\epsilon}(1-z,1-\bar{z})$$
$$g_{\epsilon\sigma\sigma\epsilon}(z,\bar{z}) = g_{\sigma\sigma\epsilon\epsilon}(1-z,1-\bar{z})$$

Crossing Even

$$g_{\sigma\sigma\sigma\sigma}(z,\bar{z})$$
$$g_{\epsilon\epsilon\epsilon\epsilon}(z,\bar{z})$$
$$g_{\sigma\epsilon\sigma\epsilon}(z,\bar{z})$$
$$g_{\sigma\sigma\epsilon\epsilon}(z,\bar{z}) + g_{\epsilon\sigma\sigma\epsilon}(z,\bar{z})$$

Crossing Odd

$$g_{\sigma\sigma\epsilon\epsilon}(z,\bar{z}) - g_{\epsilon\sigma\sigma\epsilon}(z,\bar{z})$$



Crossing Equations

```
crossingEquations :: forall s a b . (Ord b, Fractional a, Eq a)
  => FourPointFunctionTerm (ExternalOp s) (SB.Standard4PtStruct, DerivMultiplier) b a
  -> V 4 (Taylors 'ZZb, FreeVect b a)
crossingEquations g0 = toV
  ( (zzbTaylors crossOdd, gS Sig Sig Sig Sig)
  , (zzbTaylors crossOdd, gS Eps Eps Eps Eps)
  , (zzbTaylors crossOdd, gS Sig Eps Sig Eps)
  , (zzbTaylorsAll,   gS Sig Sig Eps Eps - gT Eps Sig Sig Eps)
  )
  where
    gS a b c d = g0 a b c d (SB.Standard4PtStruct, SChannel)
    gT a b c d = g0 a b c d (SB.Standard4PtStruct, TChannel)
```

Crossing Even

$$g_{\sigma\sigma\sigma\sigma}(z,\bar{z}) \qquad \partial_z^m \partial_{\bar{z}}^n g_{\sigma\sigma\sigma\sigma}$$

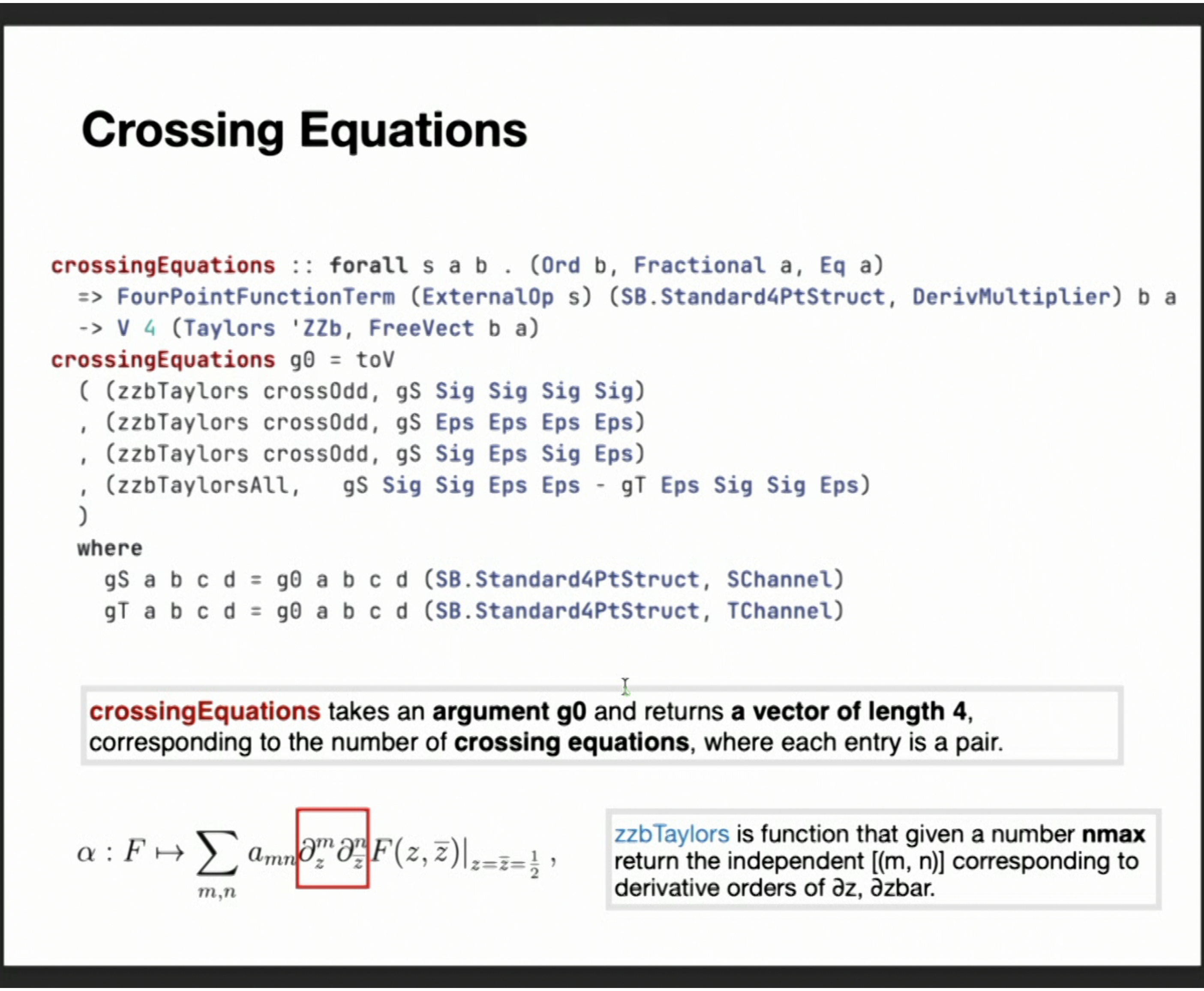
$$g_{\epsilon\epsilon\epsilon\epsilon}(z,\bar{z}) \qquad \partial_z^m \partial_{\bar{z}}^n g_{\epsilon\epsilon\epsilon\epsilon}$$

$$g_{\sigma\epsilon\sigma\epsilon}(z,\bar{z}) \qquad \partial_z^m \partial_{\bar{z}}^n g_{\sigma\epsilon\sigma\epsilon}$$

$$g_{\sigma\sigma\epsilon\epsilon}(z,\bar{z}) + g_{\epsilon\sigma\sigma\epsilon}(z,\bar{z}) \qquad \partial_z^m \partial_{\bar{z}}^n \left(g_{\sigma\sigma\epsilon\epsilon} + (-1)^{m+n} g_{\epsilon\sigma\sigma\epsilon}\right)$$

Crossing Odd

$$g_{\sigma\sigma\epsilon\epsilon}(z,\bar{z}) - g_{\epsilon\sigma\sigma\epsilon}(z,\bar{z})$$



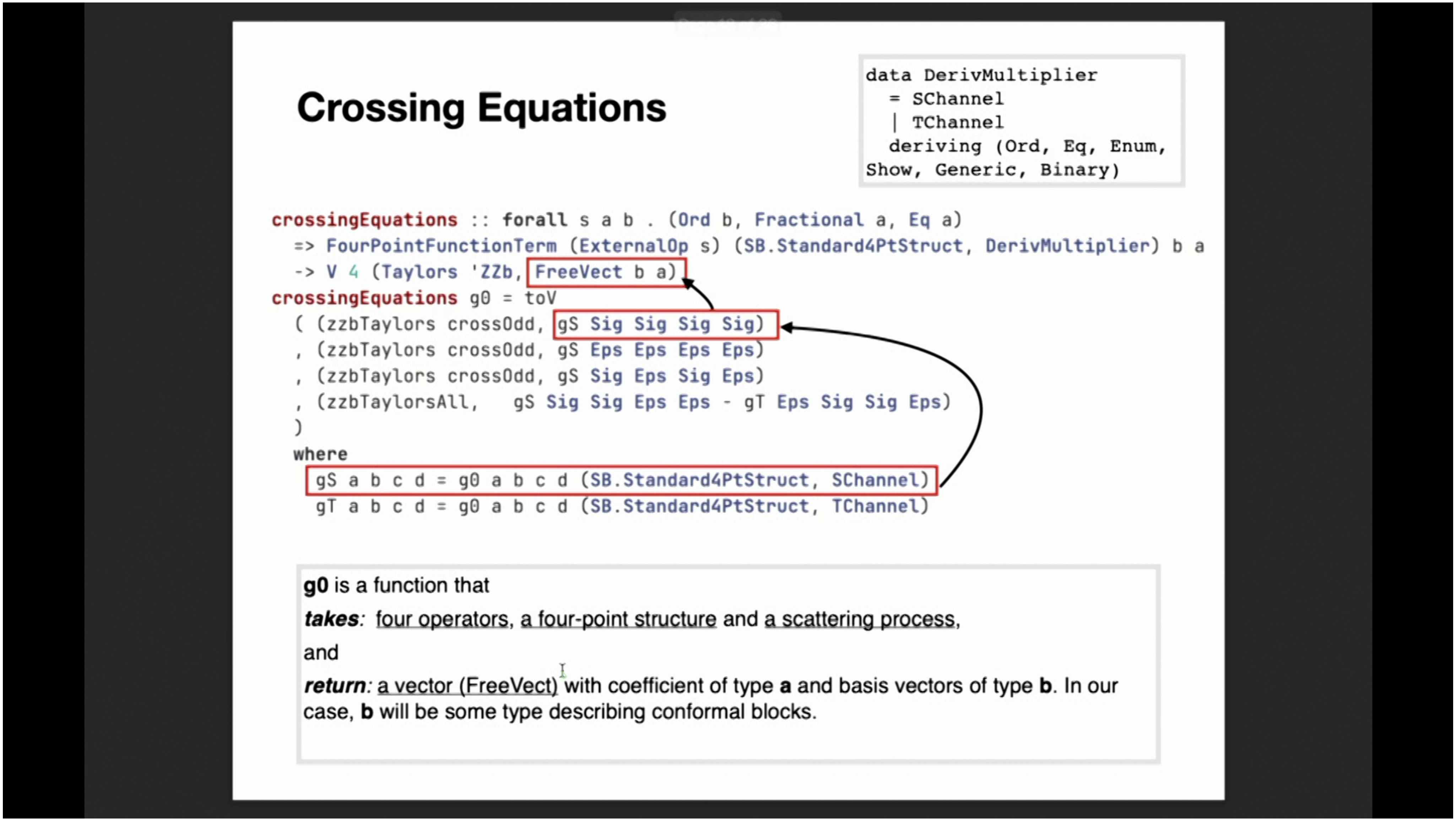
Crossing Equations

```
crossingEquations :: forall s a b . (Ord b, Fractional a, Eq a)
  => FourPointFunctionTerm (ExternalOp s) (SB.Standard4PtStruct, DerivMultiplier) b a
  -> V 4 (Taylors 'ZZb, FreeVect b a)
crossingEquations g0 = toV
  ( (zzbTaylors crossOdd, gS Sig Sig Sig Sig)
  , (zzbTaylors crossOdd, gS Eps Eps Eps Eps)
  , (zzbTaylors crossOdd, gS Sig Eps Sig Eps)
  , (zzbTaylorsAll,   gS Sig Sig Eps Eps - gT Eps Sig Sig Eps)
  )
  where
    gS a b c d = g0 a b c d (SB.Standard4PtStruct, SChannel)
    gT a b c d = g0 a b c d (SB.Standard4PtStruct, TChannel)
```

crossingEquations takes an **argument g0** and returns **a vector of length 4**, corresponding to the number of **crossing equations**, where each entry is a pair.

$$\alpha : F \mapsto \sum_{m,n} a_{mn} \boxed{\partial_z^m \partial_{\bar{z}}^n} F(z,\bar{z})|_{z=\bar{z}=\frac{1}{2}} \, ,$$

zzbTaylors is function that given a number **nmax** return the independent [(m, n)] corresponding to derivative orders of ∂z, ∂zbar.



Crossing Equations

```
data DerivMultiplier
  = SChannel
  | TChannel
  deriving (Ord, Eq, Enum,
Show, Generic, Binary)
```

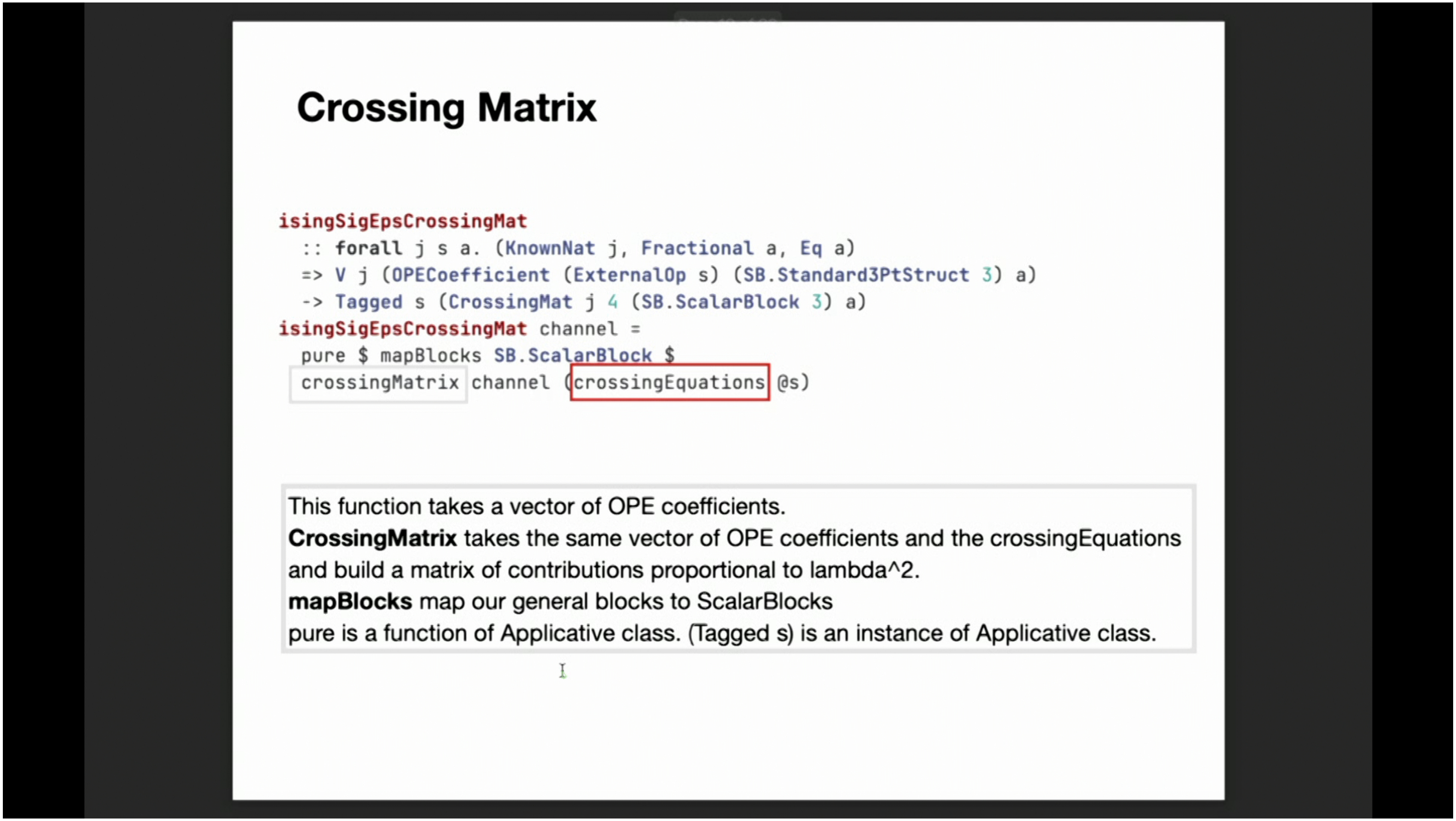
```
crossingEquations :: forall s a b . (Ord b, Fractional a, Eq a)
  => FourPointFunctionTerm (ExternalOp s) (SB.Standard4PtStruct, DerivMultiplier) b a
  -> V 4 (Taylors 'ZZb, FreeVect b a)
crossingEquations g0 = toV
  ( (zzbTaylors crossOdd, gS Sig Sig Sig Sig)
  , (zzbTaylors crossOdd, gS Eps Eps Eps Eps)
  , (zzbTaylors crossOdd, gS Sig Eps Sig Eps)
  , (zzbTaylorsAll,   gS Sig Sig Eps Eps - gT Eps Sig Sig Eps)
  )
  where
    gS a b c d = g0 a b c d (SB.Standard4PtStruct, SChannel)
    gT a b c d = g0 a b c d (SB.Standard4PtStruct, TChannel)
```

g0 is a function that

***takes**:* four operators, a four-point structure and a scattering process,

and

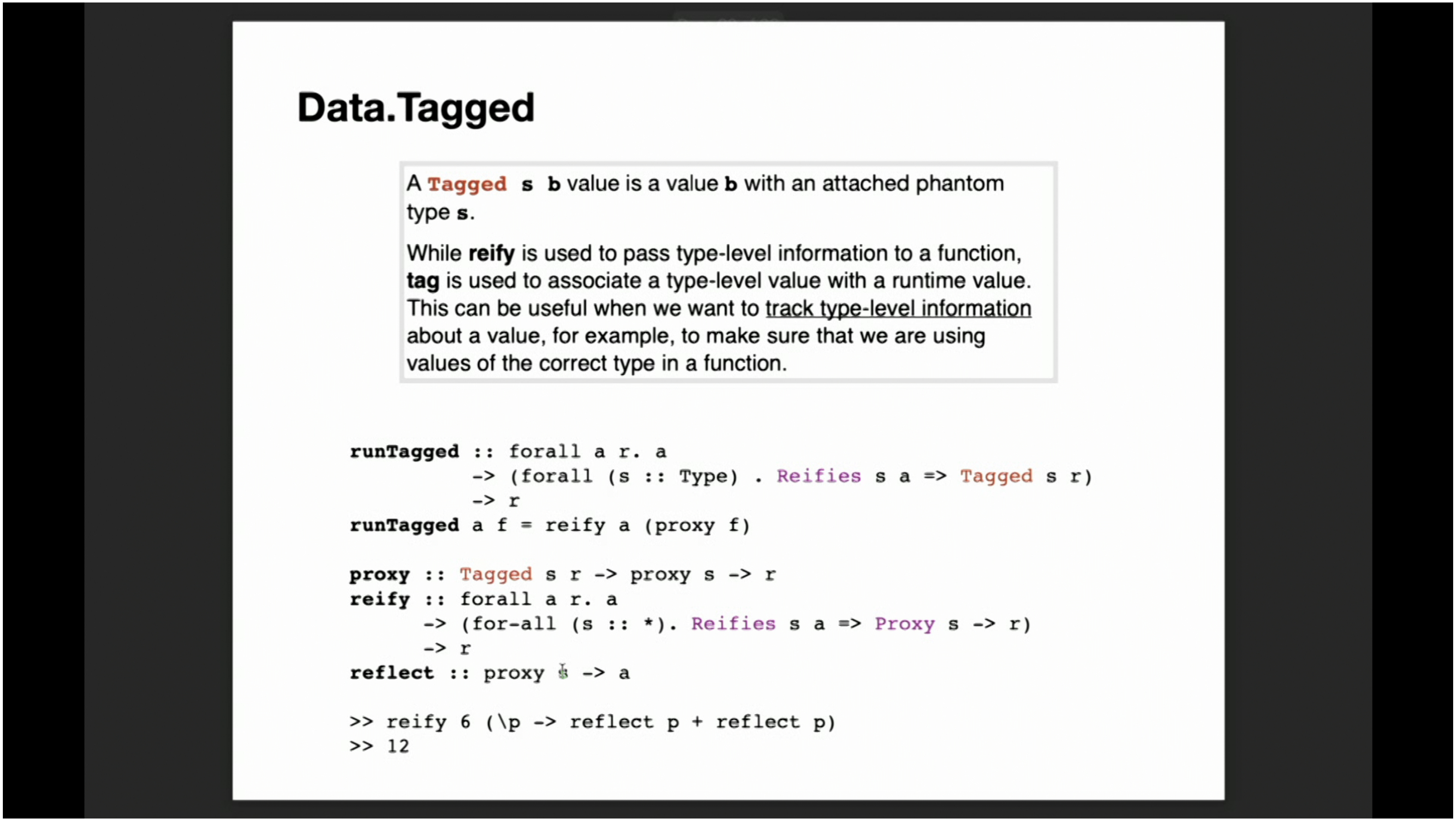
***return**:* a vector (FreeVect) with coefficient of type **a** and basis vectors of type **b**. In our case, **b** will be some type describing conformal blocks.



Crossing Matrix

```
isingSigEpsCrossingMat
  :: forall j s a. (KnownNat j, Fractional a, Eq a)
  => V j (OPECoefficient (ExternalOp s) (SB.Standard3PtStruct 3) a)
  -> Tagged s (CrossingMat j 4 (SB.ScalarBlock 3) a)
isingSigEpsCrossingMat channel =
  pure $ mapBlocks SB.ScalarBlock $
  crossingMatrix channel (crossingEquations @s)
```

This function takes a vector of OPE coefficients.
CrossingMatrix takes the same vector of OPE coefficients and the crossingEquations and build a matrix of contributions proportional to lambda^2.
mapBlocks map our general blocks to ScalarBlocks
pure is a function of Applicative class. (Tagged s) is an instance of Applicative class.

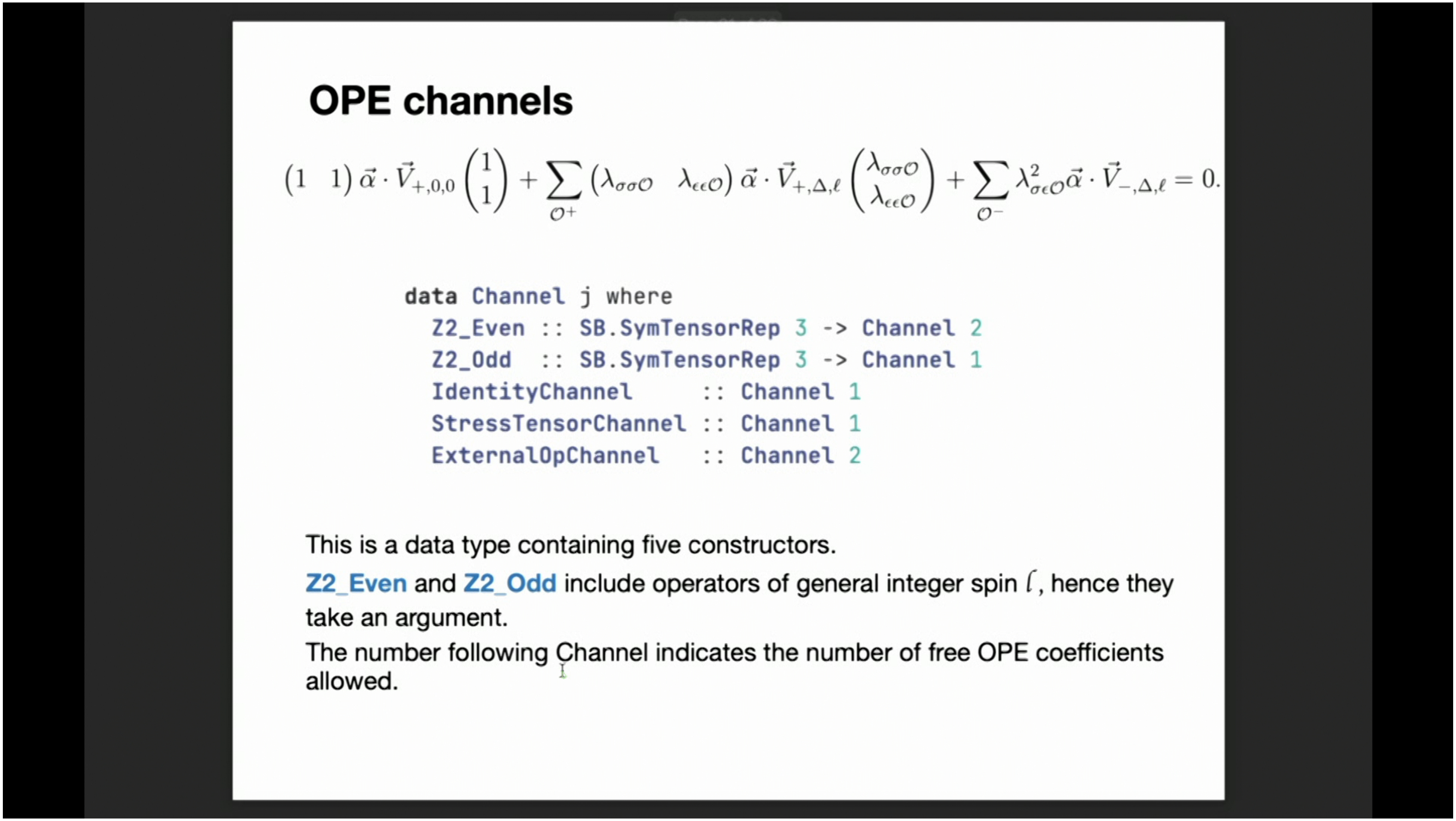


Data.Tagged

A **`Tagged s b`** value is a value **`b`** with an attached phantom type **`s`**.

While **reify** is used to pass type-level information to a function, **tag** is used to associate a type-level value with a runtime value. This can be useful when we want to track type-level information about a value, for example, to make sure that we are using values of the correct type in a function.

```
runTagged :: forall a r. a
          -> (forall (s :: Type) . Reifies s a => Tagged s r)
          -> r
runTagged a f = reify a (proxy f)

proxy :: Tagged s r -> proxy s -> r
reify :: forall a r. a
      -> (for-all (s :: *). Reifies s a => Proxy s -> r)
      -> r
reflect :: proxy s -> a

>> reify 6 (\p -> reflect p + reflect p)
>> 12
```



OPE channels

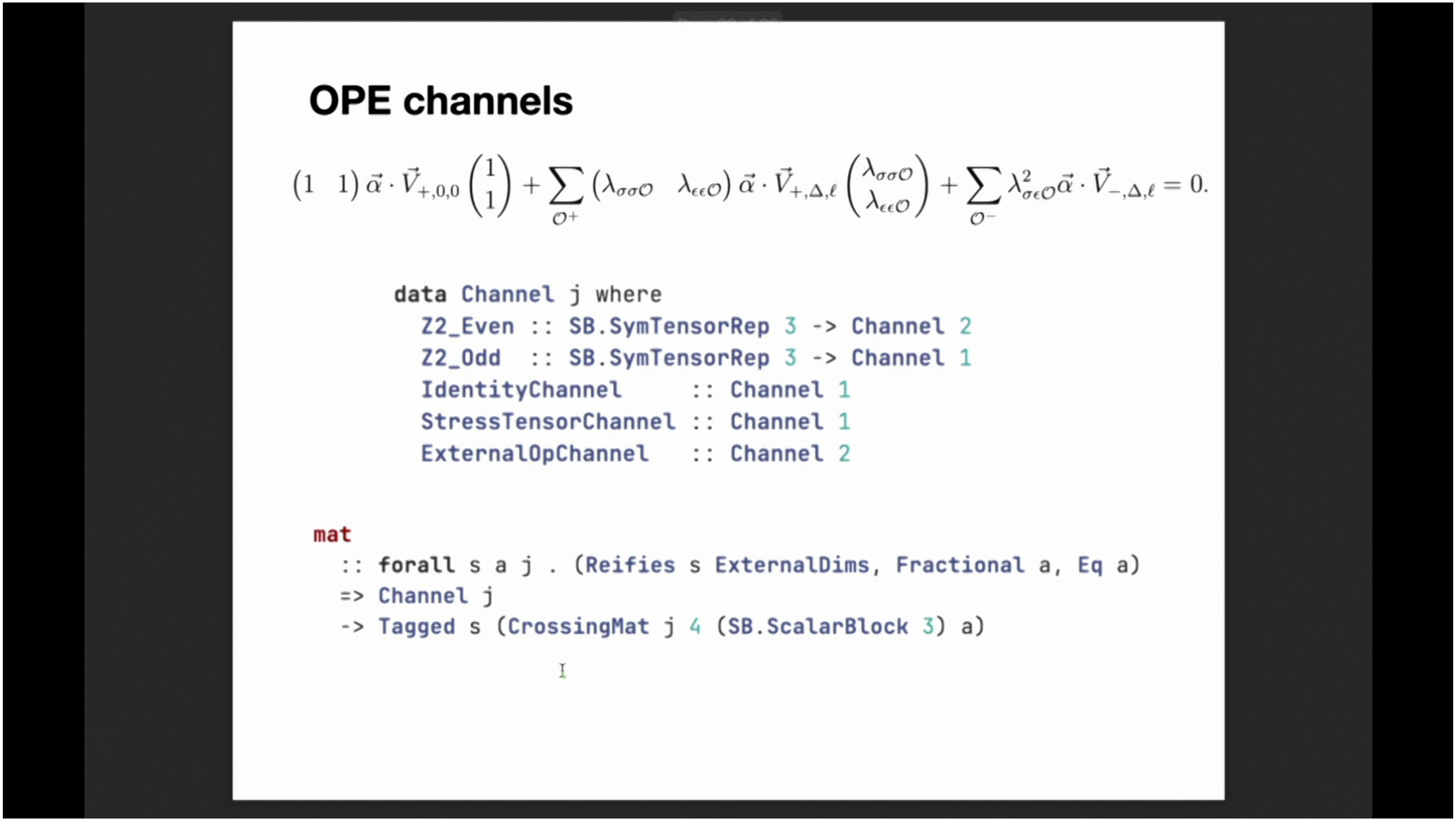
$$\begin{pmatrix}1 & 1\end{pmatrix} \vec{\alpha}\cdot\vec{V}_{+,0,0}\begin{pmatrix}1\\1\end{pmatrix} + \sum_{\mathcal{O}^+}\begin{pmatrix}\lambda_{\sigma\sigma\mathcal{O}} & \lambda_{\epsilon\epsilon\mathcal{O}}\end{pmatrix}\vec{\alpha}\cdot\vec{V}_{+,\Delta,\ell}\begin{pmatrix}\lambda_{\sigma\sigma\mathcal{O}}\\ \lambda_{\epsilon\epsilon\mathcal{O}}\end{pmatrix} + \sum_{\mathcal{O}^-}\lambda^2_{\sigma\epsilon\mathcal{O}}\vec{\alpha}\cdot\vec{V}_{-,\Delta,\ell} = 0.$$

```
data Channel j where
  Z2_Even :: SB.SymTensorRep 3 -> Channel 2
  Z2_Odd  :: SB.SymTensorRep 3 -> Channel 1
  IdentityChannel     :: Channel 1
  StressTensorChannel :: Channel 1
  ExternalOpChannel   :: Channel 2
```

This is a data type containing five constructors.

Z2_Even and **Z2_Odd** include operators of general integer spin ℓ, hence they take an argument.

The number following Channel indicates the number of free OPE coefficients allowed.

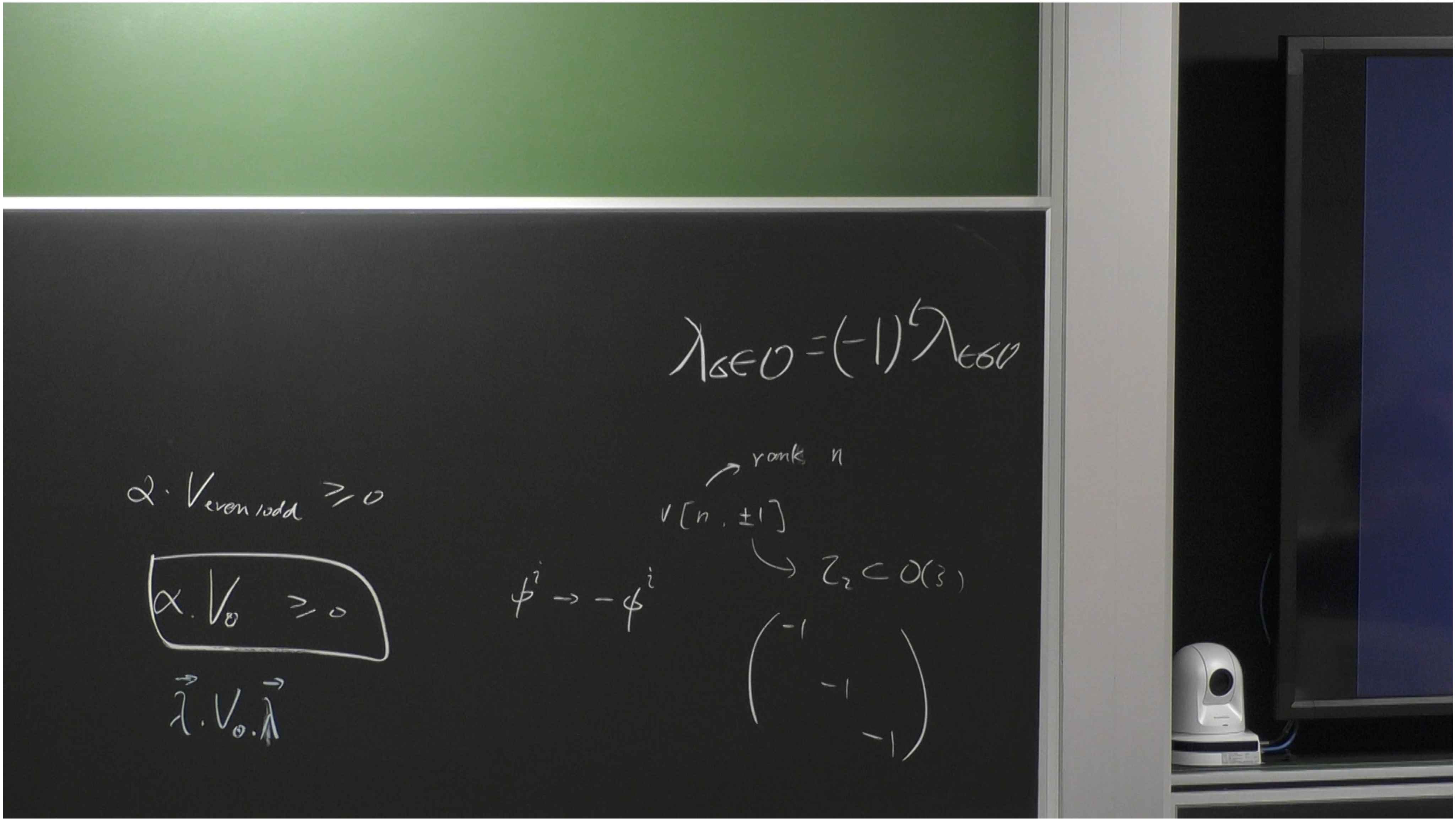


OPE channels

$$\begin{pmatrix}1 & 1\end{pmatrix} \vec{\alpha}\cdot\vec{V}_{+,0,0}\begin{pmatrix}1\\1\end{pmatrix} + \sum_{\mathcal{O}^+}\begin{pmatrix}\lambda_{\sigma\sigma\mathcal{O}} & \lambda_{\epsilon\epsilon\mathcal{O}}\end{pmatrix}\vec{\alpha}\cdot\vec{V}_{+,\Delta,\ell}\begin{pmatrix}\lambda_{\sigma\sigma\mathcal{O}}\\ \lambda_{\epsilon\epsilon\mathcal{O}}\end{pmatrix} + \sum_{\mathcal{O}^-}\lambda^2_{\sigma\epsilon\mathcal{O}}\vec{\alpha}\cdot\vec{V}_{-,\Delta,\ell} = 0.$$

```
data Channel j where
  Z2_Even :: SB.SymTensorRep 3 -> Channel 2
  Z2_Odd  :: SB.SymTensorRep 3 -> Channel 1
  IdentityChannel     :: Channel 1
  StressTensorChannel :: Channel 1
  ExternalOpChannel   :: Channel 2
```

```
mat
  :: forall s a j . (Reifies s ExternalDims, Fractional a, Eq a)
  => Channel j
  -> Tagged s (CrossingMat j 4 (SB.ScalarBlock 3) a)
```



$$\lambda_{\sigma\epsilon O} = (-1)^{\ell} \lambda_{\epsilon\sigma O}$$

$$\alpha \cdot V_{\text{even/odd}} \geqslant 0$$

$$\boxed{\alpha \cdot V_{8} \geqslant 0}$$

$$\vec{\lambda} \cdot V_{\theta} \cdot \vec{\lambda}$$

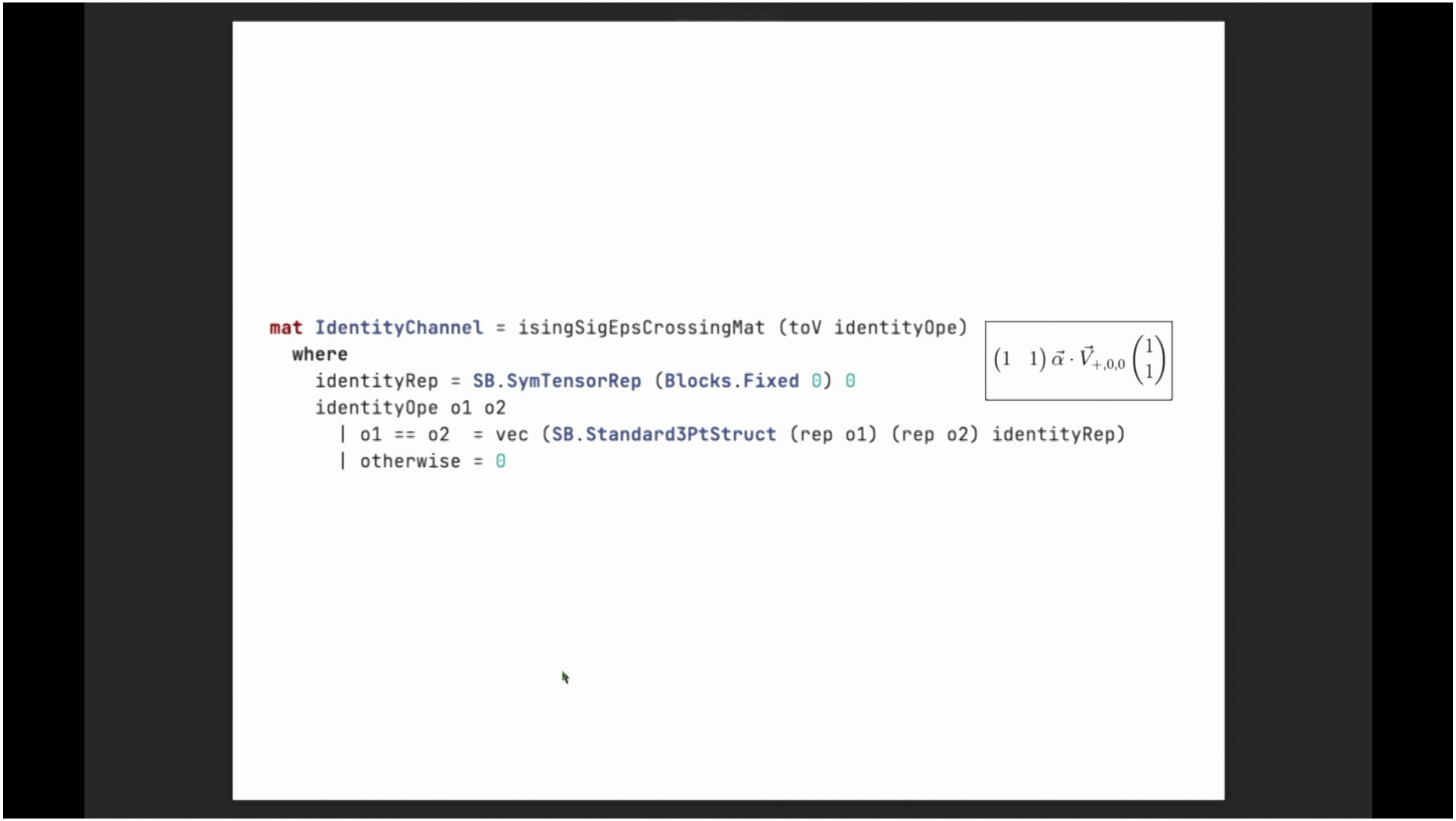
$$\phi^{i} \to -\phi^{i}$$

rank n

$$v[n, \pm 1]$$

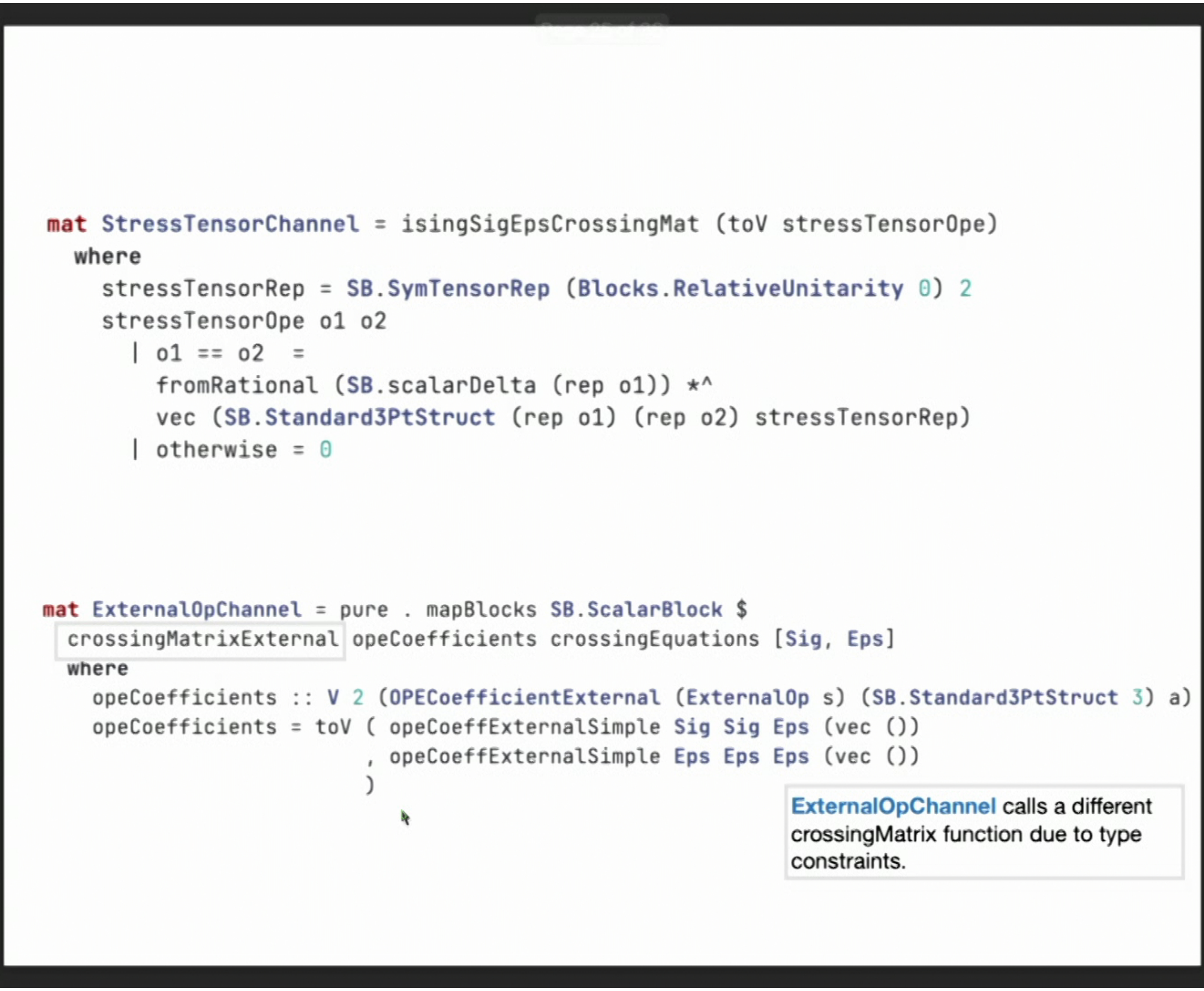
$$\to \mathbb{Z}_2 \subset O(3)$$

$$\begin{pmatrix} -1 & & \\ & -1 & \\ & & -1 \end{pmatrix}$$



```
mat IdentityChannel = isingSigEpsCrossingMat (toV identityOpe)
  where
    identityRep = SB.SymTensorRep (Blocks.Fixed 0) 0
    identityOpe o1 o2
      | o1 == o2  = vec (SB.Standard3PtStruct (rep o1) (rep o2) identityRep)
      | otherwise = 0
```

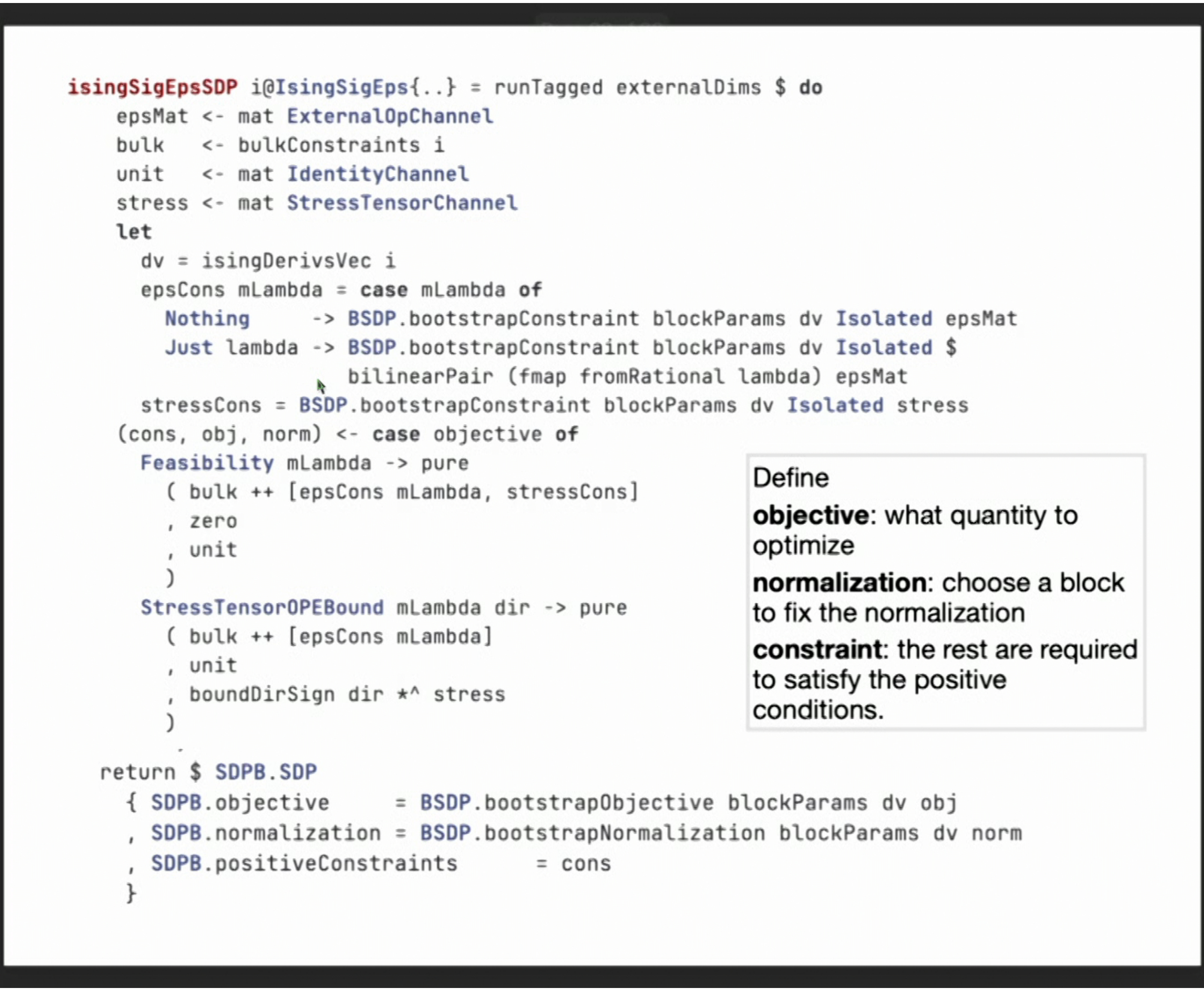
$$\boxed{\begin{pmatrix}1 & 1\end{pmatrix}\vec{\alpha}\cdot\vec{V}_{+,0,0}\begin{pmatrix}1\\1\end{pmatrix}}$$



```
mat StressTensorChannel = isingSigEpsCrossingMat (toV stressTensorOpe)
  where
    stressTensorRep = SB.SymTensorRep (Blocks.RelativeUnitarity 0) 2
    stressTensorOpe o1 o2
      | o1 == o2  =
        fromRational (SB.scalarDelta (rep o1)) *^
        vec (SB.Standard3PtStruct (rep o1) (rep o2) stressTensorRep)
      | otherwise = 0
```

```
mat ExternalOpChannel = pure . mapBlocks SB.ScalarBlock $
  crossingMatrixExternal opeCoefficients crossingEquations [Sig, Eps]
  where
    opeCoefficients :: V 2 (OPECoefficientExternal (ExternalOp s) (SB.Standard3PtStruct 3) a)
    opeCoefficients = toV ( opeCoeffExternalSimple Sig Sig Eps (vec ())
                          , opeCoeffExternalSimple Eps Eps Eps (vec ())
                          )
```

ExternalOpChannel calls a different crossingMatrix function due to type constraints.



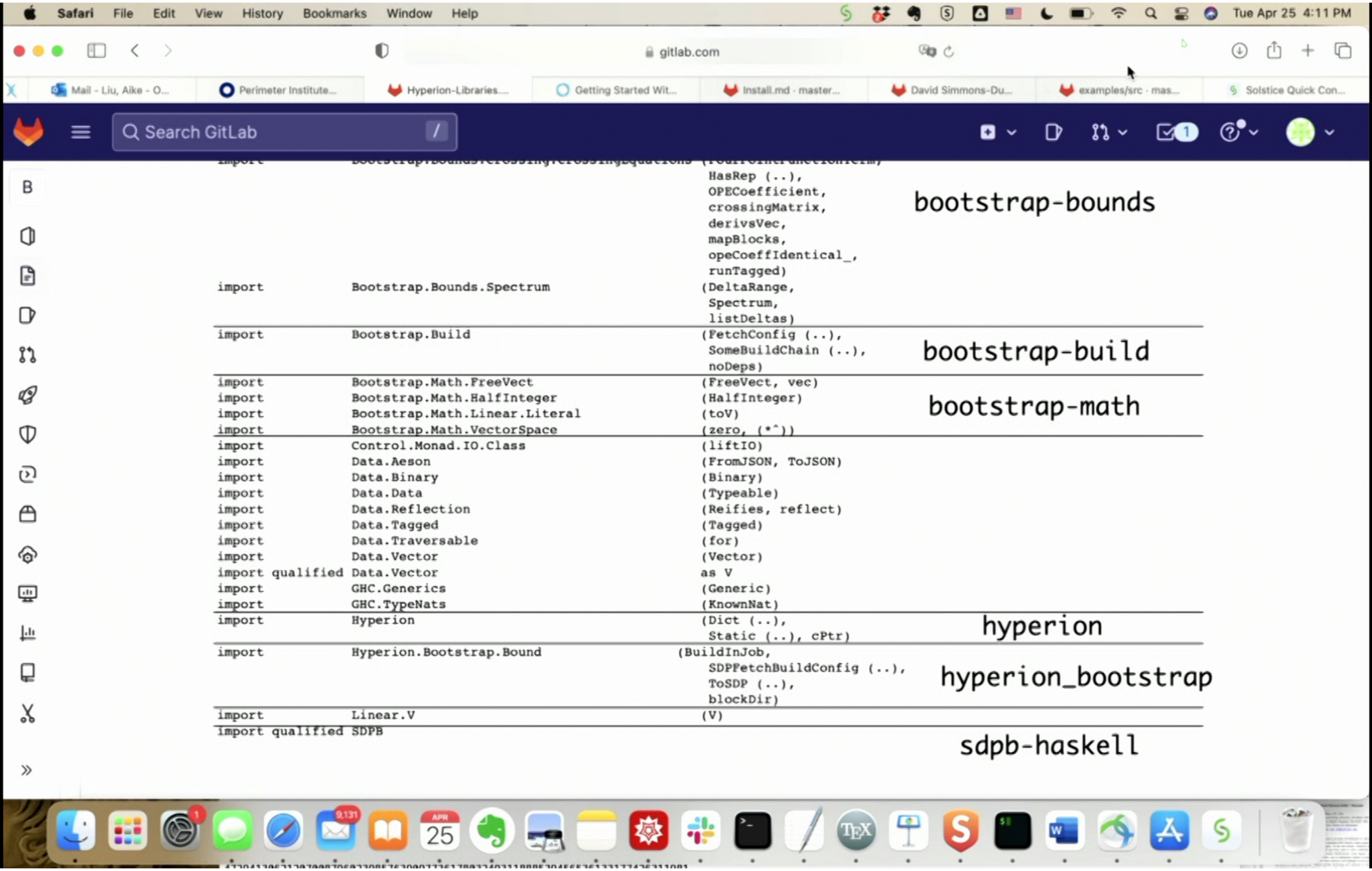
```
isingSigEpsSDP i@IsingSigEps{..} = runTagged externalDims $ do
    epsMat <- mat ExternalOpChannel
    bulk   <- bulkConstraints i
    unit   <- mat IdentityChannel
    stress <- mat StressTensorChannel
    let
      dv = isingDerivsVec i
      epsCons mLambda = case mLambda of
        Nothing     -> BSDP.bootstrapConstraint blockParams dv Isolated epsMat
        Just lambda -> BSDP.bootstrapConstraint blockParams dv Isolated $
                       bilinearPair (fmap fromRational lambda) epsMat
      stressCons = BSDP.bootstrapConstraint blockParams dv Isolated stress
    (cons, obj, norm) <- case objective of
      Feasibility mLambda -> pure
        ( bulk ++ [epsCons mLambda, stressCons]
        , zero
        , unit
        )
      StressTensorOPEBound mLambda dir -> pure
        ( bulk ++ [epsCons mLambda]
        , unit
        , boundDirSign dir *^ stress
        )
  return $ SDPB.SDP
    { SDPB.objective     = BSDP.bootstrapObjective blockParams dv obj
    , SDPB.normalization = BSDP.bootstrapNormalization blockParams dv norm
    , SDPB.positiveConstraints      = cons
    }
```

Define

objective: what quantity to optimize

normalization: choose a block to fix the normalization

constraint: the rest are required to satisfy the positive conditions.



```
                                                              HasRep (..),
                                                              OPECoefficient,
                                                              crossingMatrix,
                                                              derivsVec,
                                                              mapBlocks,
                                                              opeCoeffIdentical_,
                                                              runTagged)
import           Bootstrap.Bounds.Spectrum                   (DeltaRange,
                                                              Spectrum,
                                                              listDeltas)
import           Bootstrap.Build                             (FetchConfig (..),
                                                              SomeBuildChain (..),
                                                              noDeps)
import           Bootstrap.Math.FreeVect                     (FreeVect, vec)
import           Bootstrap.Math.HalfInteger                  (HalfInteger)
import           Bootstrap.Math.Linear.Literal               (toV)
import           Bootstrap.Math.VectorSpace                  (zero, (*^))
import           Control.Monad.IO.Class                      (liftIO)
import           Data.Aeson                                  (FromJSON, ToJSON)
import           Data.Binary                                 (Binary)
import           Data.Data                                   (Typeable)
import           Data.Reflection                             (Reifies, reflect)
import           Data.Tagged                                 (Tagged)
import           Data.Traversable                            (for)
import           Data.Vector                                 (Vector)
import qualified Data.Vector                                 as V
import           GHC.Generics                                (Generic)
import           GHC.TypeNats                                (KnownNat)
import           Hyperion                                    (Dict (..),
                                                              Static (..), cPtr)
import           Hyperion.Bootstrap.Bound                  (BuildInJob,
                                                              SDPFetchBuildConfig (..),
                                                              ToSDP (..),
                                                              blockDir)
import           Linear.V                                    (V)
import qualified SDPB
```

bootstrap-bounds

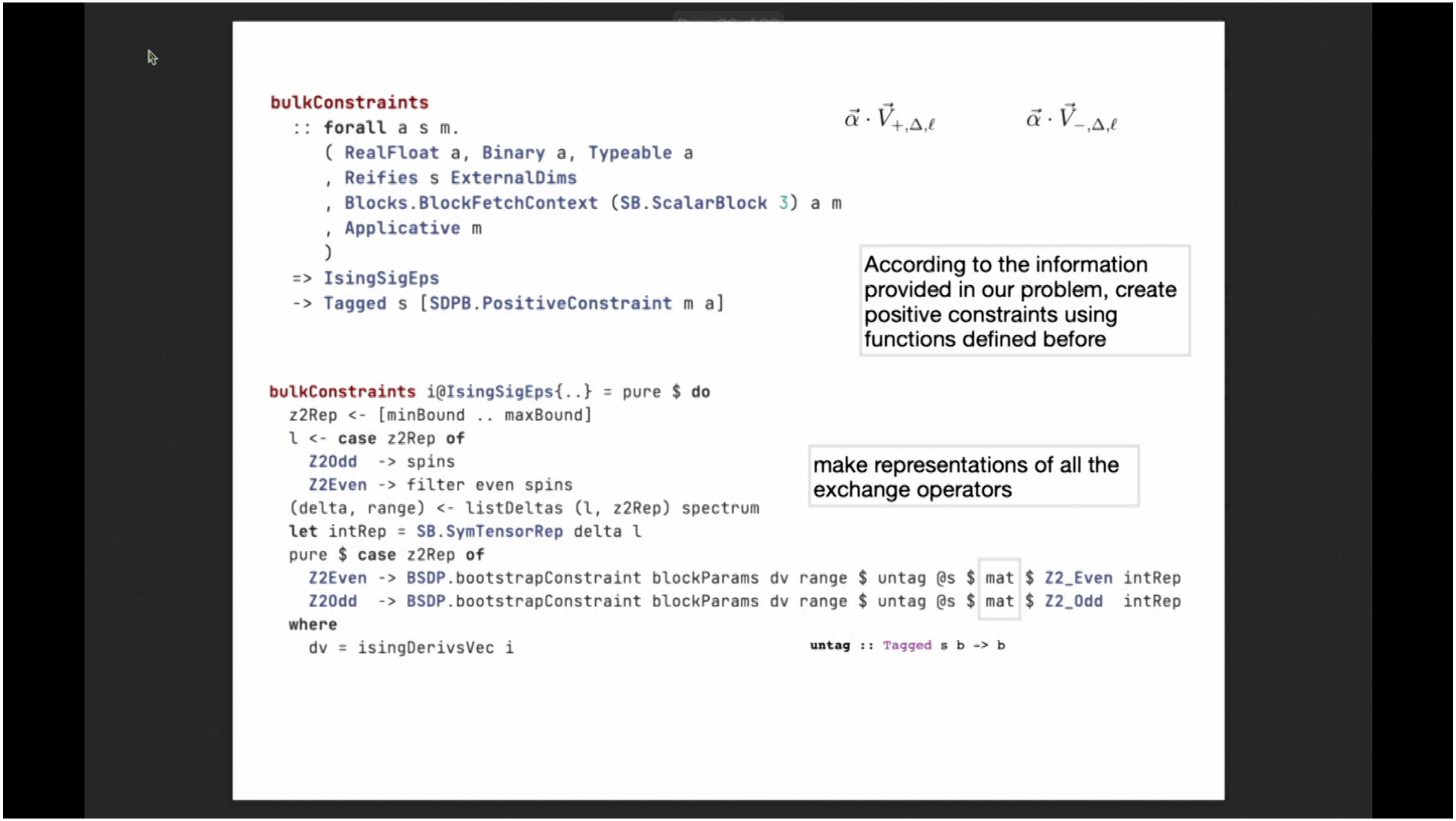
bootstrap-build

bootstrap-math

hyperion

hyperion_bootstrap

sdpb-haskell



```
bulkConstraints
  :: forall a s m.
     ( RealFloat a, Binary a, Typeable a
     , Reifies s ExternalDims
     , Blocks.BlockFetchContext (SB.ScalarBlock 3) a m
     , Applicative m
     )
  => IsingSigEps
  -> Tagged s [SDPB.PositiveConstraint m a]
```

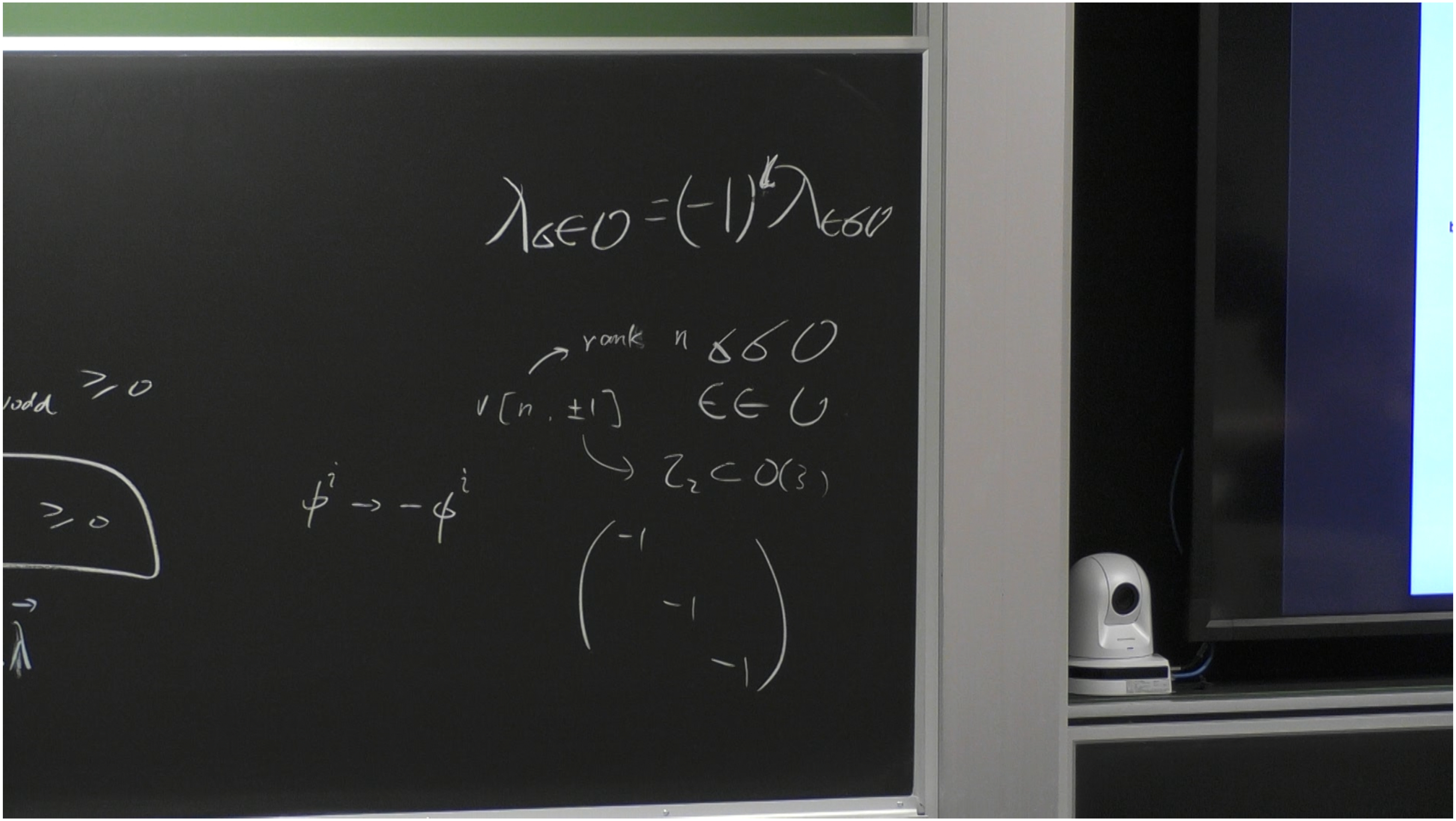
$\vec{\alpha}\cdot\vec{V}_{+,\Delta,\ell}$ $\vec{\alpha}\cdot\vec{V}_{-,\Delta,\ell}$

According to the information provided in our problem, create positive constraints using functions defined before

```
bulkConstraints i@IsingSigEps{..} = pure $ do
  z2Rep <- [minBound .. maxBound]
  l <- case z2Rep of
    Z2Odd  -> spins
    Z2Even -> filter even spins
  (delta, range) <- listDeltas (l, z2Rep) spectrum
  let intRep = SB.SymTensorRep delta l
  pure $ case z2Rep of
    Z2Even -> BSDP.bootstrapConstraint blockParams dv range $ untag @s $ mat $ Z2_Even intRep
    Z2Odd  -> BSDP.bootstrapConstraint blockParams dv range $ untag @s $ mat $ Z2_Odd  intRep
  where
    dv = isingDerivsVec i
```

make representations of all the exchange operators

```
untag :: Tagged s b -> b
```



$\text{odd} \geq 0$

$\boxed{\geq 0}$

$\vec{\lambda}$

$$\lambda_{\sigma\epsilon\mathcal{O}} = (-1)^{\ell}\lambda_{\epsilon\sigma\mathcal{O}}$$

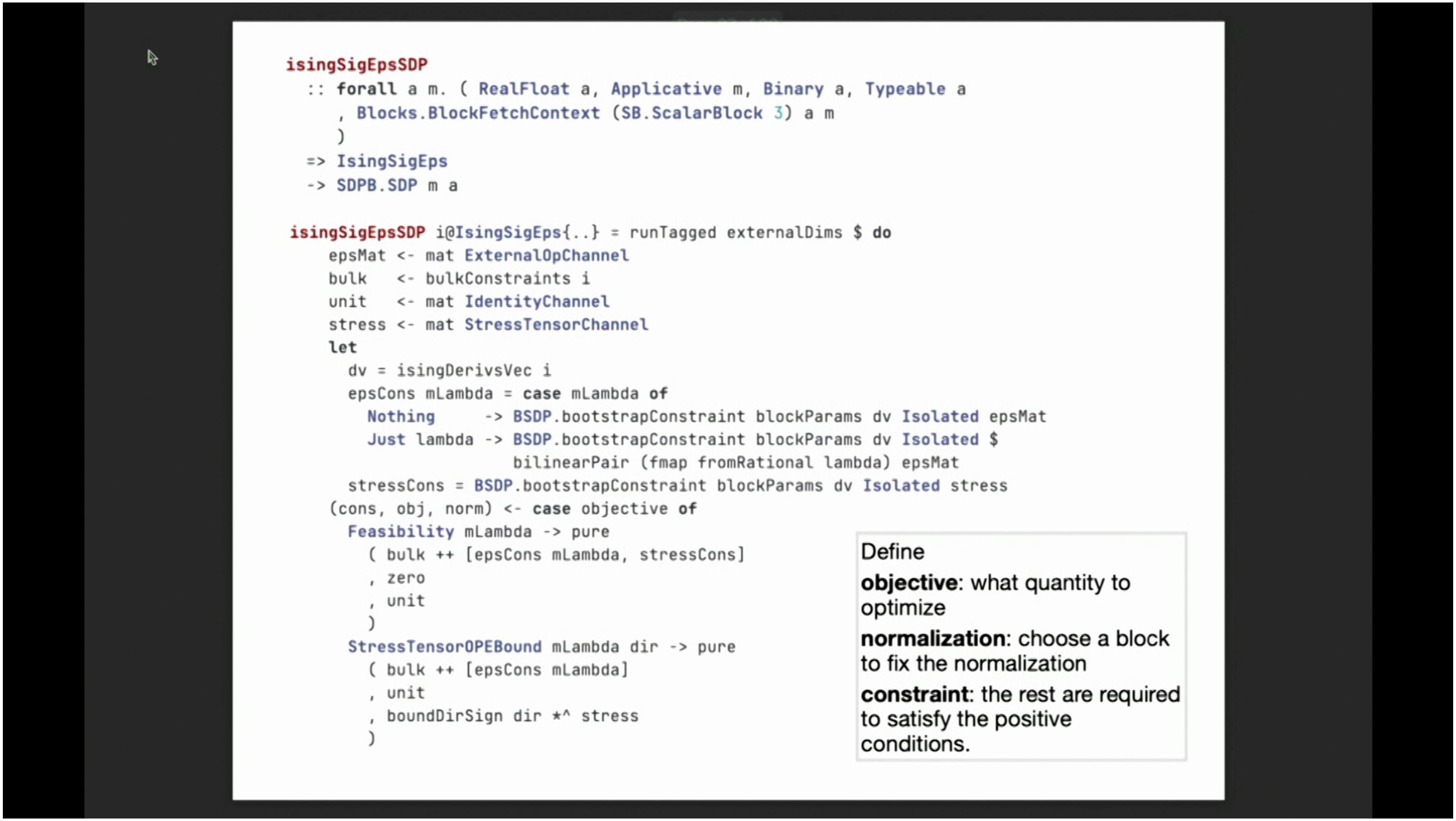
rank n $\quad \sigma\sigma\mathcal{O}$

$v[n, \pm 1] \quad \epsilon\epsilon\mathcal{O}$

$\to \mathbb{Z}_2 \subset O(3)$

$$\phi^i \to -\phi^i$$

$$\begin{pmatrix} -1 & & \\ & -1 & \\ & & -1 \end{pmatrix}$$



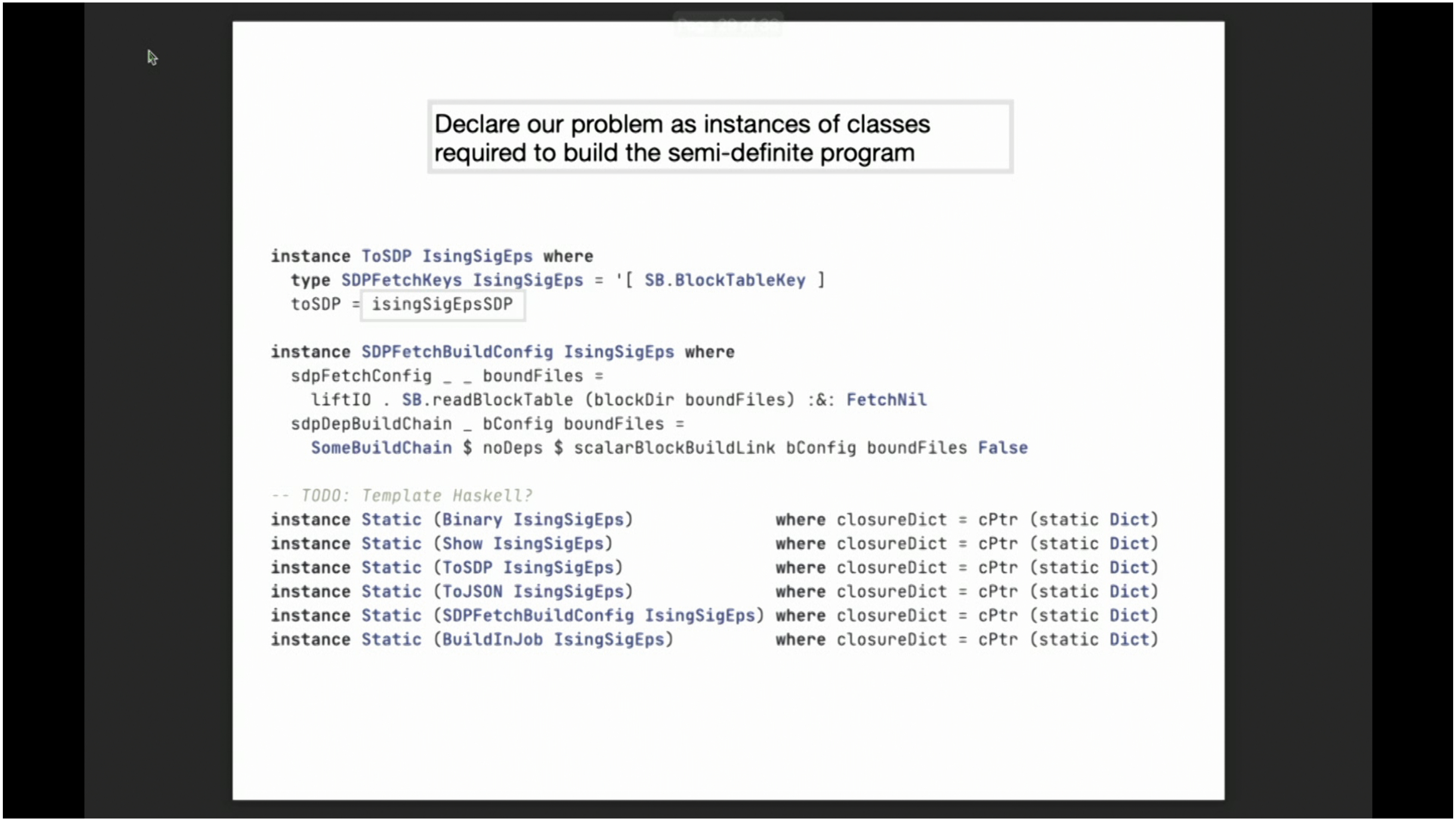
```
isingSigEpsSDP
  :: forall a m. ( RealFloat a, Applicative m, Binary a, Typeable a
     , Blocks.BlockFetchContext (SB.ScalarBlock 3) a m
     )
  => IsingSigEps
  -> SDPB.SDP m a

isingSigEpsSDP i@IsingSigEps{..} = runTagged externalDims $ do
    epsMat <- mat ExternalOpChannel
    bulk   <- bulkConstraints i
    unit   <- mat IdentityChannel
    stress <- mat StressTensorChannel
    let
      dv = isingDerivsVec i
      epsCons mLambda = case mLambda of
        Nothing     -> BSDP.bootstrapConstraint blockParams dv Isolated epsMat
        Just lambda -> BSDP.bootstrapConstraint blockParams dv Isolated $
                       bilinearPair (fmap fromRational lambda) epsMat
      stressCons = BSDP.bootstrapConstraint blockParams dv Isolated stress
    (cons, obj, norm) <- case objective of
      Feasibility mLambda -> pure
        ( bulk ++ [epsCons mLambda, stressCons]
        , zero
        , unit
        )
      StressTensorOPEBound mLambda dir -> pure
        ( bulk ++ [epsCons mLambda]
        , unit
        , boundDirSign dir *^ stress
        )
```

Define

objective: what quantity to optimize

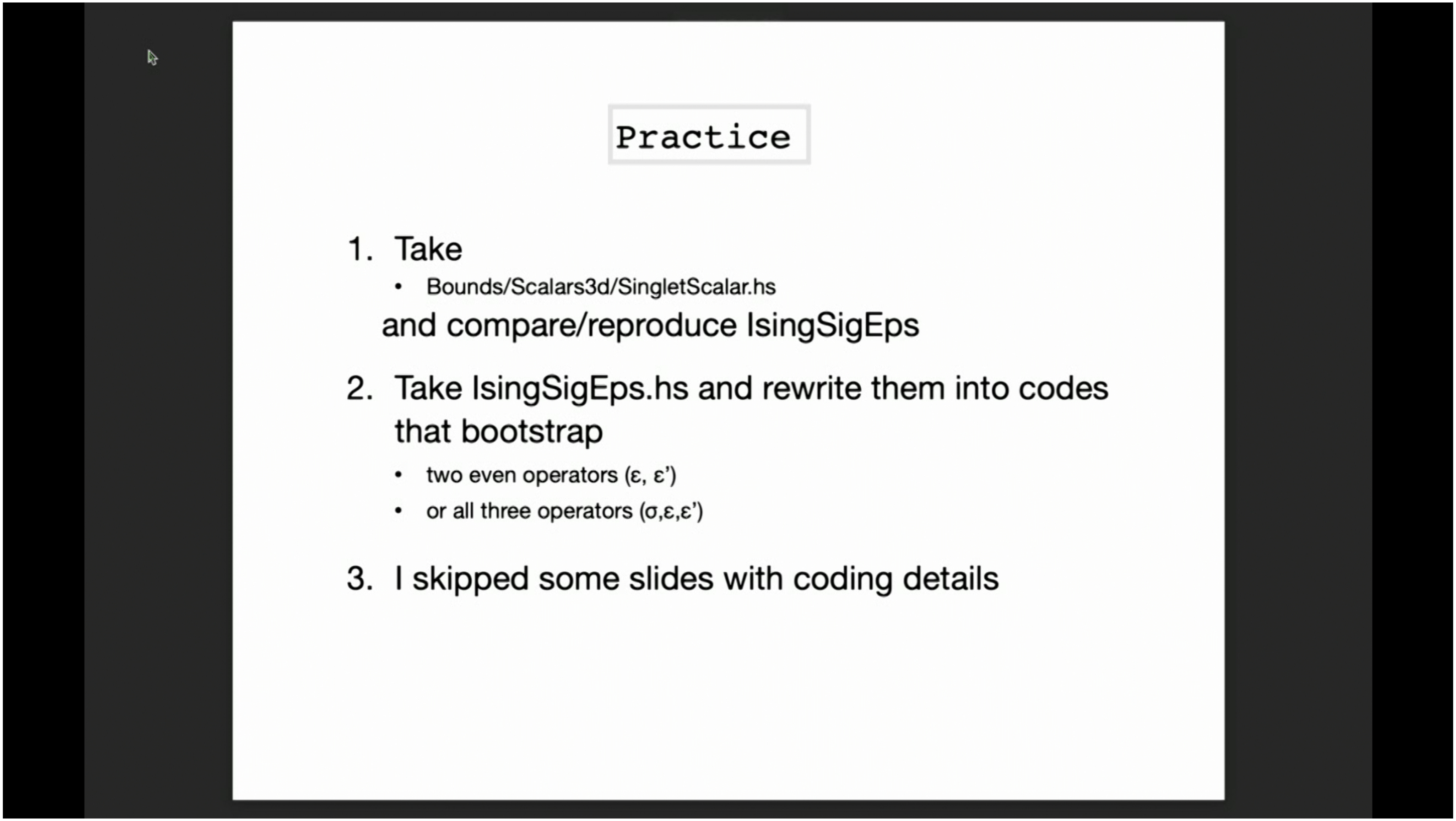
normalization: choose a block to fix the normalization

constraint: the rest are required to satisfy the positive conditions.



Declare our problem as instances of classes required to build the semi-definite program

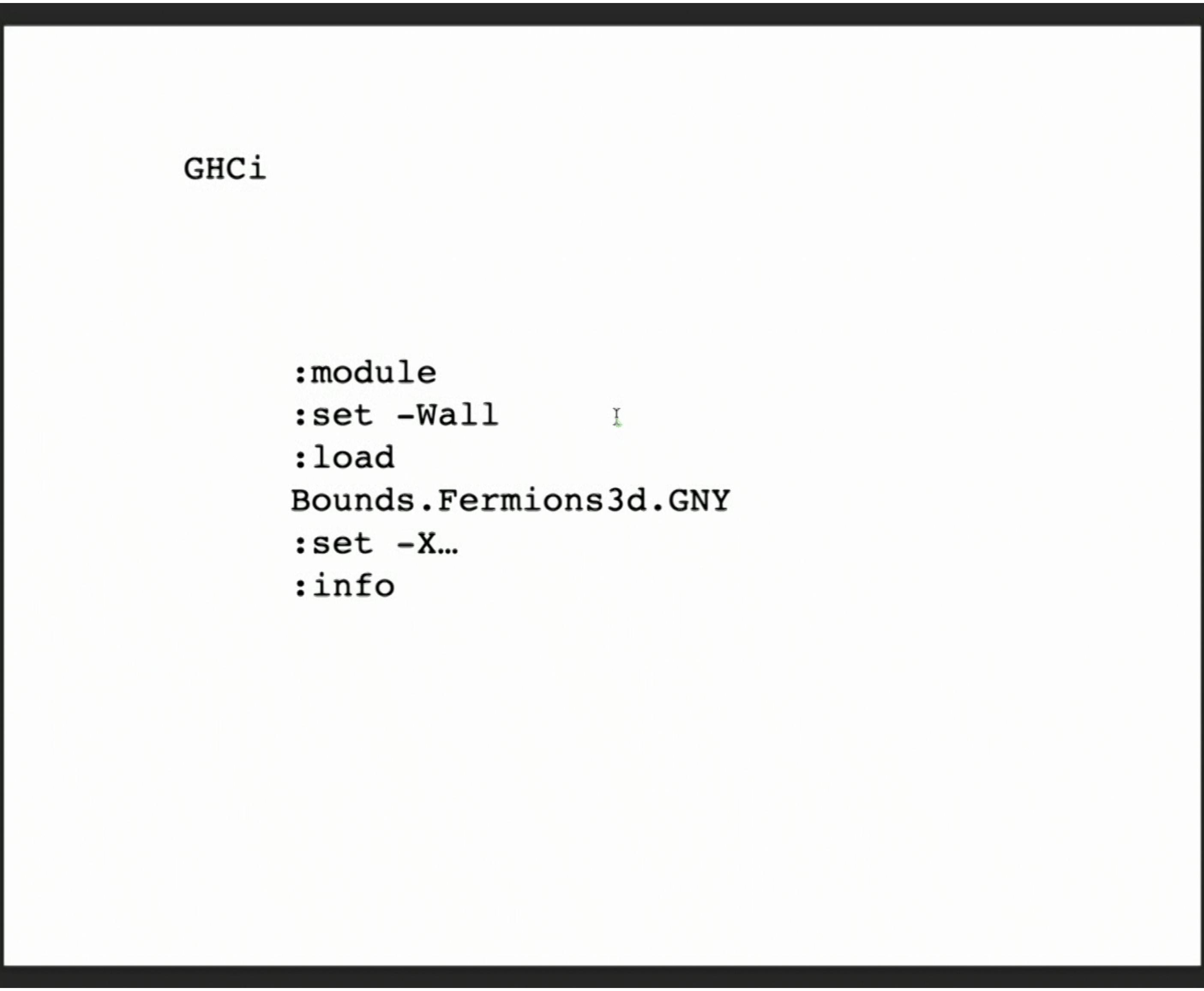
```
instance ToSDP IsingSigEps where
  type SDPFetchKeys IsingSigEps = '[ SB.BlockTableKey ]
  toSDP = isingSigEpsSDP

instance SDPFetchBuildConfig IsingSigEps where
  sdpFetchConfig _ _ boundFiles =
    liftIO . SB.readBlockTable (blockDir boundFiles) :&: FetchNil
  sdpDepBuildChain _ bConfig boundFiles =
    SomeBuildChain $ noDeps $ scalarBlockBuildLink bConfig boundFiles False

-- TODO: Template Haskell?
instance Static (Binary IsingSigEps)              where closureDict = cPtr (static Dict)
instance Static (Show IsingSigEps)                where closureDict = cPtr (static Dict)
instance Static (ToSDP IsingSigEps)               where closureDict = cPtr (static Dict)
instance Static (ToJSON IsingSigEps)              where closureDict = cPtr (static Dict)
instance Static (SDPFetchBuildConfig IsingSigEps) where closureDict = cPtr (static Dict)
instance Static (BuildInJob IsingSigEps)          where closureDict = cPtr (static Dict)
```



Practice

1. Take
   - Bounds/Scalars3d/SingletScalar.hs

   and compare/reproduce IsingSigEps
2. Take IsingSigEps.hs and rewrite them into codes that bootstrap
   - two even operators (ε, ε')
   - or all three operators (σ,ε,ε')
3. I skipped some slides with coding details



GHCi

```
:module
:set -Wall
:load
Bounds.Fermions3d.GNY
:set -X…
:info
```