Embedding By Normalisation

Shayan Najd
Laboratory for Foundations of Computer Science,
The University of Edinburgh

Cambridge, August 2016

CamBridge = Bridge over Cam

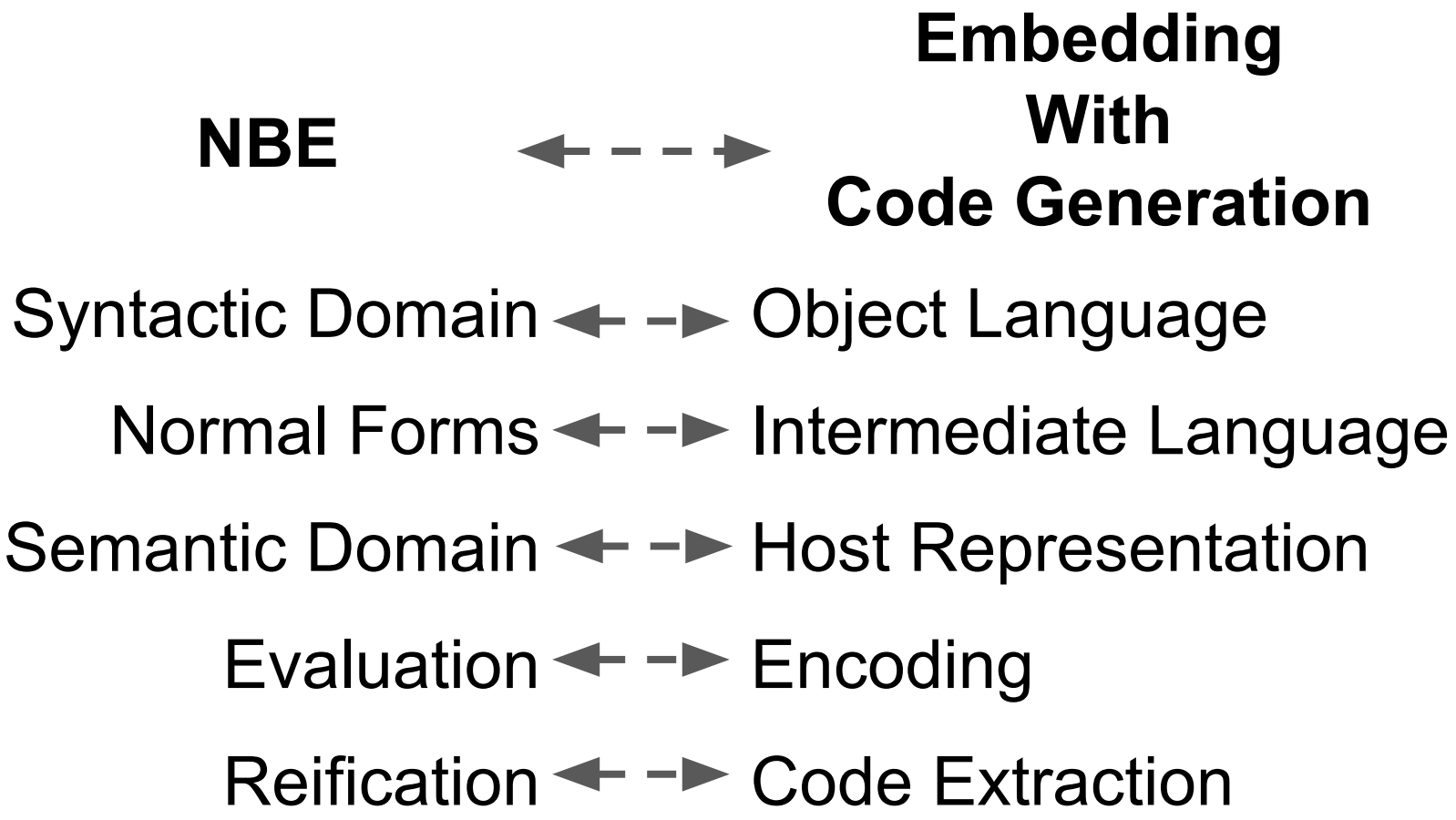
Theory

Practice

Theory  Practice

Normalisation
By
Evaluation
(NBE)

Embedding DSLs
With
Code Generation


NBE
Embedding
With
Code Generation
Syntactic Domain
Object Language
Normal Forms
Intermediate Language
Semantic Domain
Host Representation
Evaluation
Encoding
Reification
Code Extraction

Normalisation
By
Evaluation
(NBE)

Embedding
By
Normalisation
(EBN)

Normalisation
By
Evaluation
(NBE)

Embedding
By
Normalisation
(EBN)

&

Svenningsson Axelsson

Feldspar's Shallow-Deep Embedding

Gill

Shallow EDSL's Code Extraction

Domain-specific languages and code synthesis using Haskell
Queue, *12*(4), 30

Gill

Control flow is problematic and cannot be used directly [in this setting]

Gill

Domain-specific languages and code synthesis using Haskell
Queue, *12*(4), 30

Svenningsson

&

Axelsson

Combining deep and shallow embedding for EDSL,
TFP 2012

[...] one may be tempted to make a Syntactic instance for *Maybe*. Unfortunately, there is no way to make this work [in this setting]

Svenningsson

&

Axelsson

Combining deep and shallow embedding for EDSL, TFP 2012

Embedding DSLs With Code Generation

```
A,B ::= Exp α
      | A → B
      | A x B
      | A + B
      | ι
```

&

Dybjer Filinski

NBE with Interpreted Constants

Danvy

Type-Directed Partial Evaluation

Danvy

Type-Directed Partial Evaluation
POPL 1996

[to support disjoint sums] we thus need to abstract and relocate this context. Context abstraction is achieved with a control operator.

Danvy

Type-Directed Partial Evaluation
POPL 1996

&

Dybjer Filinski

Normalization and Partial Evaluation
APPSEM 2000

Dybjer

&

Filinski

To recover a simple notion of equivalence [w.r.t. interpreted constants], we need to introduce an explicit notion of binding times in the programs.

Normalization and Partial Evaluation
APPSEM 2000

Normalisation By Evaluation (NBE)

```
A,B ::= Exp α
      | A → B
      | A x B
      | A + B
      | ι
```

Normalisation By Evaluation (NBE)

Embedding DSLs With Code Generation

```
A,B ::= Exp α
      | A → B
      | A x B
      | A + B
      | ι
```

Normalisation By Evaluation (NBE)

Embedding By Normalisation (EBN)

```
A,B ::= Exp α
      | A → B
      | A x B
      | A + B
      | ι
```

Embedding DSLs With Code Generation

LINQ

Generates

SQL

Lava

Generates

VHDL

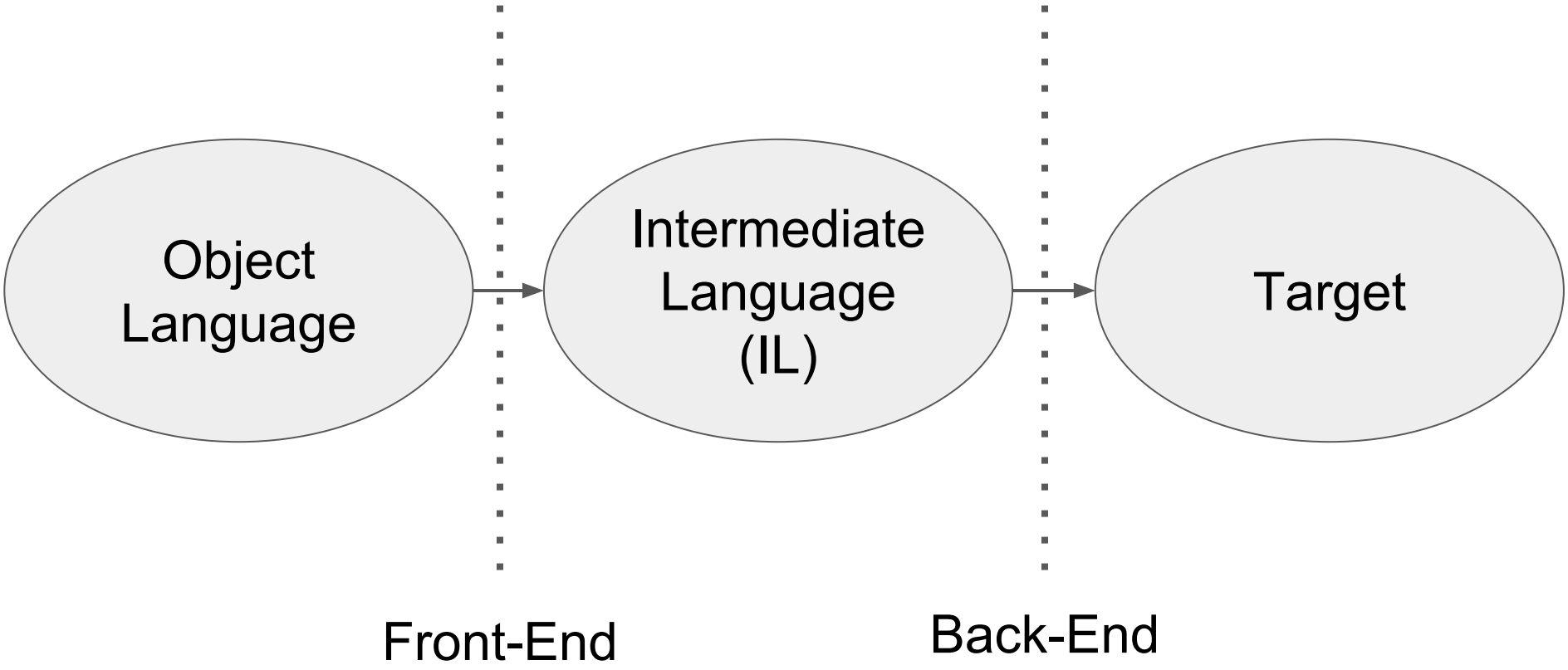
Feldspar

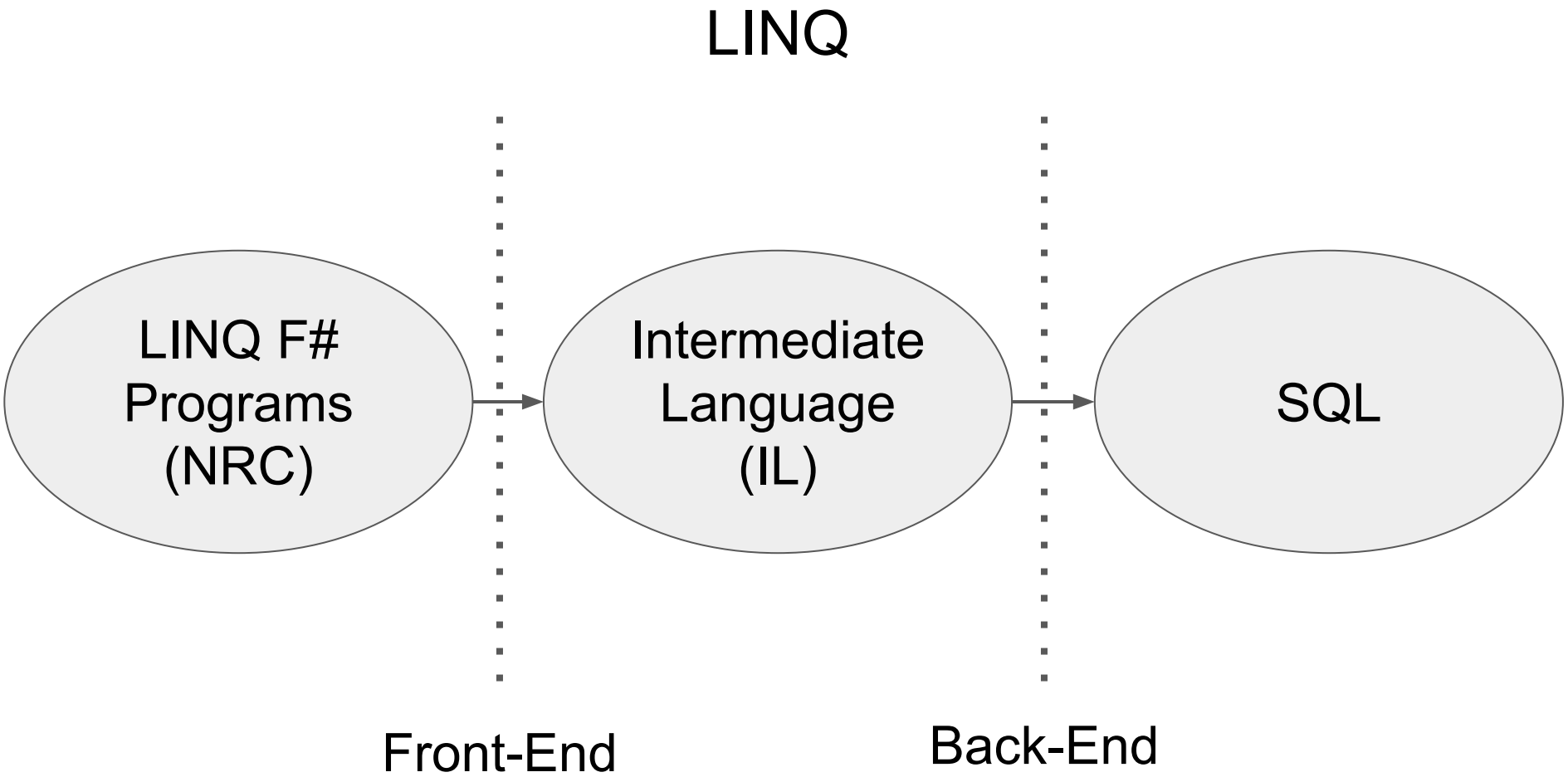
Generates

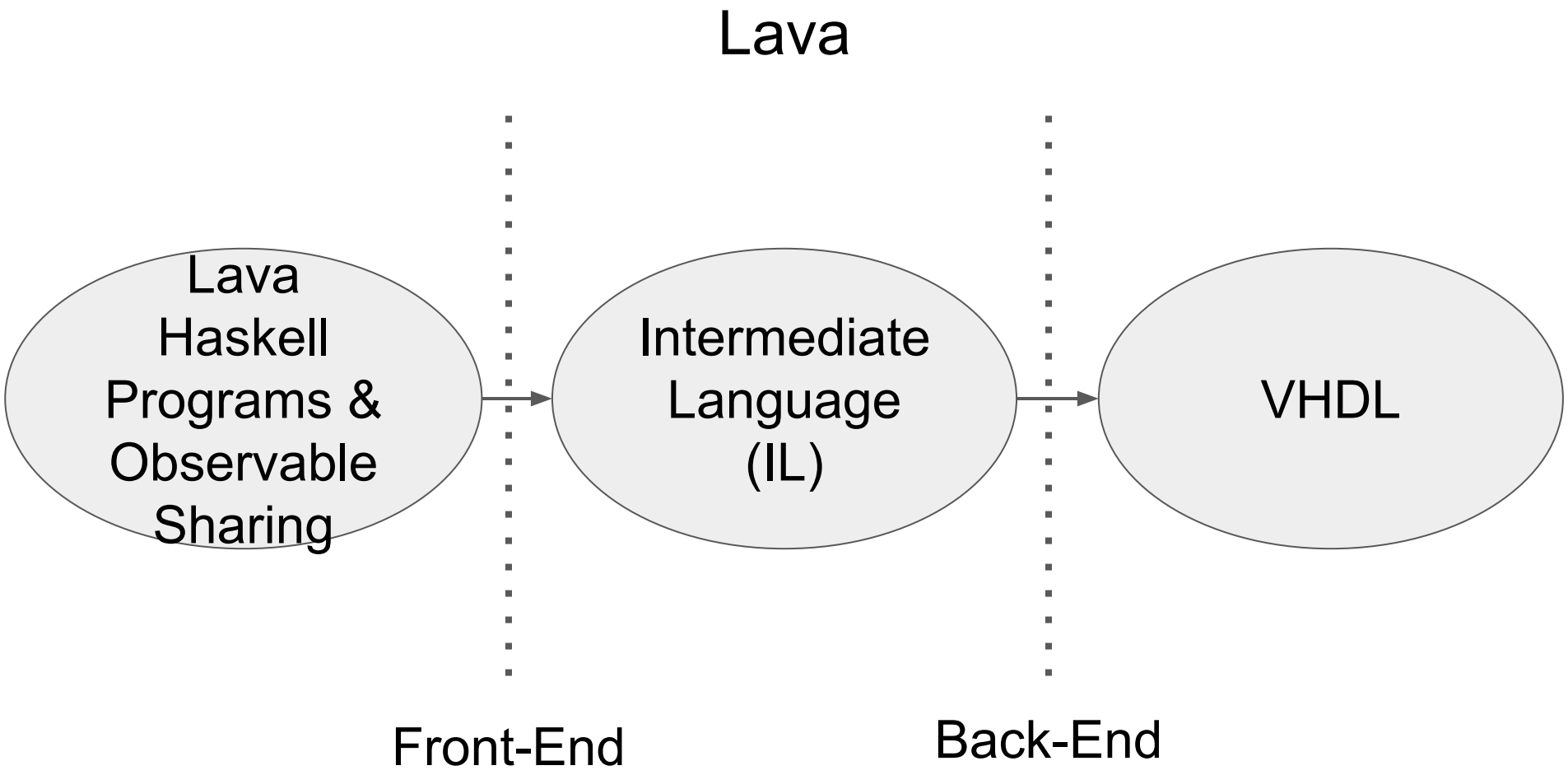
Cx

```
{  . . . ;
   . . . ;
   . . . ;  }
```

Embedding DSLs With Code Generation




LINQ
LINQ F#
Programs
(NRC)
Intermediate
Language
(IL)
SQL
Front-End
Back-End

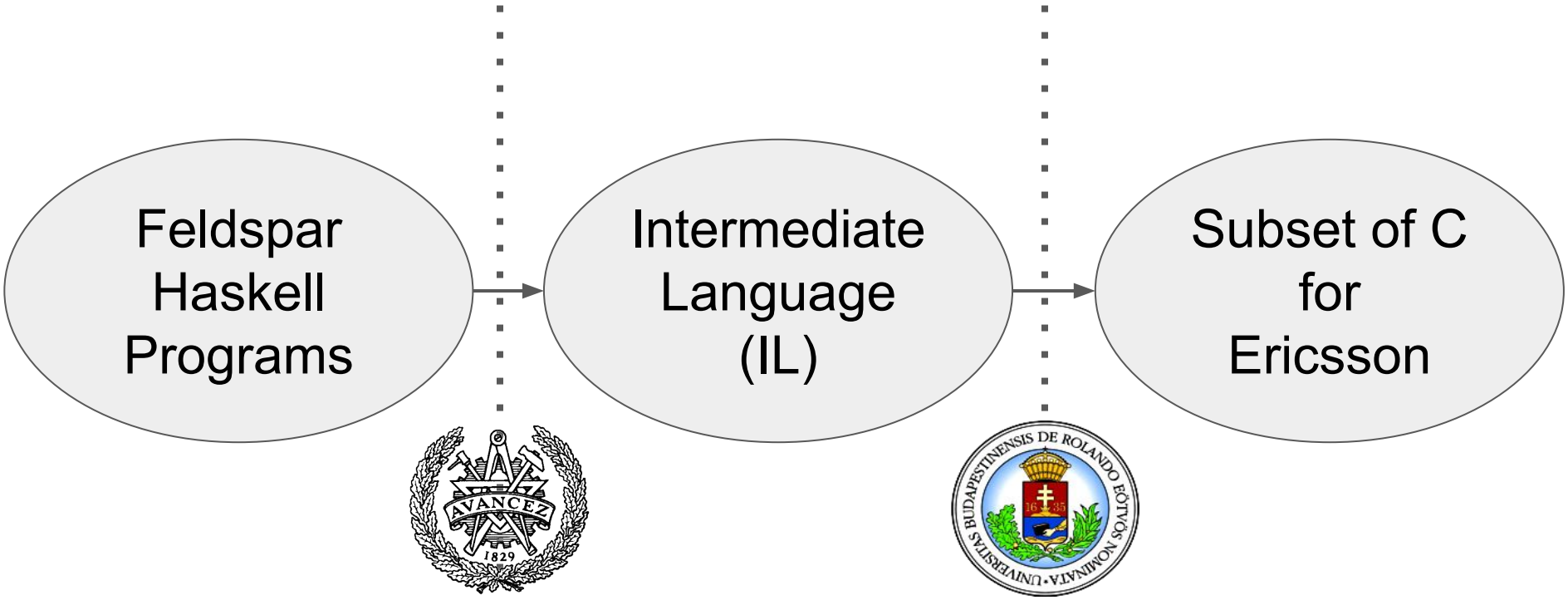

Lava
Lava
Haskell
Programs &
Observable
Sharing
Intermediate
Language
(IL)
VHDL
Front-End
Back-End

Feldspar
(EDSL in Haskell)

Chalmers
University of Technology

Eötvös Loránd University

Feldspar


Feldspar Haskell Programs
Intermediate Language (IL)
Subset of C for Ericsson
AVANCEZ
1829
UNIVERSITAS BUDAPESTINENSIS DE ROLANDO EÖTVÖS NOMINATA
feldspar-language package
feldspar-compiler package

Feldspar

Demo

> git clone https://github.com/shayan-najd/NanoFeldspar

NanoFeldspar

NanoFeldspar

Expressions

```
 x ∈ variables
 w ∈ word32 integers
 b     ::= False | True
 L,M,N ::= x
         | w | M + N | M - N | M * N | M / N | M % N | √ m
         | M < N | M ≡ N
         | ( M , N ) | fst L | snd L
         | < N | x ↤ 0 to M > | L[M] | len L
         | b | if L then M else N
```

Programs

```
 P ::= M | λ x. P
```

Expression Types

```
 A,B ::= Word32 | Bool | (A , B) | < A >
```

Program Types

```
 C   ::= A | A → C
```

```
data Exp a where
  Var   :: String -> Exp a
  LitI  :: Word32 -> Exp Word32
  Add   :: Exp Word32 -> Exp Word32 -> Exp Word32
  Sub   :: Exp Word32 -> Exp Word32 -> Exp Word32
  Mul   :: Exp Word32 -> Exp Word32 -> Exp Word32
  Div   :: Exp Word32 -> Exp Word32 -> Exp Word32
  Mod   :: Exp Word32 -> Exp Word32 -> Exp Word32
  Sqrt  :: Exp Word32 -> Exp Word32
  Eql   :: Exp Word32 -> Exp Word32 -> Exp Bool
  Ltd   :: Exp Word32 -> Exp Word32 -> Exp Bool
  Tup   :: (Type a, Type b) => Exp a -> Exp b -> Exp (a , b)
  Fst   :: Type (a , b) => Exp (a , b) -> Exp a
  Snd   :: Type (a , b) => Exp (a , b) -> Exp b
  Ary   :: Type a => Exp Word32 -> (Exp Word32 -> Exp a) -> Exp (Array Word32 a)
  Len   :: Type a => Exp (Array Word32 a) -> Exp Word32
  Ind   :: Exp (Array Word32 a) -> Exp Word32 -> Exp a
  LitB  :: Bool -> Exp Bool
  Cnd   :: Exp Bool -> Exp a -> Exp a -> Exp a

data Prg a where
  Exp :: Type a => Exp a -> Prg a
  Fun :: Type a => (Exp a -> Prg b) -> Prg (a -> b)

data STyp a where
  SWrd :: STyp Word32
  SBol :: STyp Bool
  STup :: (Type a,Type b) => STyp a -> STyp b -> STyp (a , b)
  SAry :: Type a => STyp a -> STyp (Array Word32 a)

class Type a where {typ :: STyp a}
```

NanoFeldspar

```haskell
module Examples.Live where

import Data.Word
import IL
import BackEnd.Compiler

inc :: Prg (Word32 -> Word32)
inc = Fun (\ x -> Exp (Add x (LitI 1)))
```

```
*Live> :t compile
compile :: Prg a -> String

*Live> compile inc
// after C macro expansion
unsigned int func (unsigned int v0)
{
  return (v0 + 1u);
}
```

```haskell
module Examples.Live where

import Data.Word
import IL
import BackEnd.Evaluator

inc :: Prg (Word32 -> Word32)
inc = Fun (\ x -> Exp (Add x (LitI 1)))
```

```
*Live> :t evaluate
evaluate :: Prg a -> a

*Live> evaluate inc 41
42
```

NanoFeldspar

Expressions

```
 x ∈ variables
 w ∈ word32 integers
 b      ::= False | True
 L,M,N ::= x
         | w | M + N | M - N | M * N | M / N | M % N | √ m
         | M < N | M ≡ N
         | ( M , N ) | fst L | snd L
         | < N | x ↤ 0 to M > | L[M] | len L
         | b | if L then M else N
```

Programs

```
 P ::= M | λ x. P
```

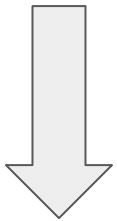
Expression Types

```
 A,B ::= Word32 | Bool | (A , B) | < A >
```

Program Types

```
 C   ::= A | A → C
```

```
inc :: Prg (Word32 -> Word32)
inc = Fun (\ x -> Exp (Add x (LitI 1)))
```



```
inc :: Prg (Word32 -> Word32)
inc = Fun (\ x -> Exp (x + 1))
```

```
instance Num (Exp Word32) where
  fromInteger = LitI . fromInteger
  (+)         = Add
  (-)         = Sub
  (*)         = Mul

divE :: Exp Word32 -> Exp Word32 -> Exp Word32
divE = Div

modE :: Exp Word32 -> Exp Word32 -> Exp Word32
modE = Mod

sqrtE :: Exp Word32 -> Exp Word32
sqrtE = Sqrt

infix 4 ==.
(==.) :: Exp Word32 -> Exp Word32 -> Exp Bool
(==.) = Eql

infix 4 <.
(<.) :: Exp Word32 -> Exp Word32 -> Exp Bool
(<.) = Ltd
```

Expressions

```
 x ∈ variables
 w ∈ word32 integers
 b      ::= False | True
 L,M,N ::= x
         | w | M + N | M - N | M * N | M / N | M % N | √ m
         | M < N | M ≡ N
         | ( M , N ) | fst L | snd L
         | < N | x ↤ 0 to M > | L[M] | len L
         | b | if L then M else N
```

Programs

```
 P ::= M | λ x. P
```

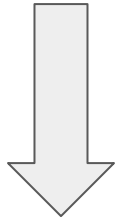
Expression Types

```
 A,B ::= Word32 | Bool | (A , B) | < A >
```

Program Types

```
 C   ::= A | A → C
```

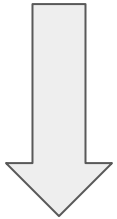
```
inc :: Prg (Word32 -> Word32)
inc = Fun (\ x -> Exp (x + 1))
```



```
inc :: Prg (Word32 -> Word32)
inc = fromFun1 (\ x -> x + 1)
```

```
fromFun1 :: (Exp Word32 -> Exp Word32) ->
            Prg (Word32 -> Word32)
fromFun1 f = Fun (\ x -> Exp (f x))
```

```
avg :: Prg (Word32 -> Word32 -> Word32)
avg = Fun (\ x -> Fun (\ y -> Exp ((x + y) `divE` 2)))
```



```
avg :: Prg (Word32 -> Word32 -> Word32)
avg = fromFun2 (\ x y -> (x + y) `divE` 2)
```

```
fromFun1 :: (Exp Word32 -> Exp Word32) ->
            Prg (Word32 -> Word32)
fromFun1 f = Fun (\ x -> Exp (f x))

fromFun2 :: (Exp Word32 -> Exp Word32 -> Exp Word32) ->
            Prg (Word32 -> Word32 -> Word32)
fromFun2 f = Fun (\ x -> fromFun1 (f x))
```

```
compile :: PrgLike a => a -> String
compile = BackEnd.compile . toPrg

evaluate :: PrgLike a => a -> (InT a)
evaluate = BackEnd.evaluate . toPrg

type family InT a
class PrgLike a where
  toPrg  :: a -> Prg (InT a)
  frmPrg :: Prg (InT a) -> a
class Type (InT a) => ExpLike a where
  toExp  :: a -> Exp (InT a)
  frmExp :: Exp (InT a) -> a
```

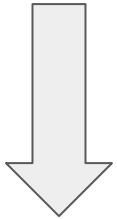
Svenningsson

&

Axelsson

```haskell
type instance InT (Prg a) = a
instance PrgLike (Prg a) where
  toPrg  = id
  frmPrg = id

type instance InT (Exp a) = a
instance Type a => ExpLike (Exp a) where
  toExp  = id
  frmExp = id
instance Type a => PrgLike (Exp a) where
  toPrg = Exp
  frmPrg (Exp m) = m
  frmPrg (Fun _) = case typ :: STyp a of {}
                   -- impossible

type instance InT (a -> b) = InT a -> InT b
instance (ExpLike a, PrgLike b) => PrgLike (a -> b) where
  toPrg  f       = Fun (toPrg . f . frmExp)
  frmPrg (Exp m) = case typOf m of {}
                   -- impossible
  frmPrg (Fun f) = frmPrg . f . toExp
```

```
avg :: Prg (Word32 -> Word32 -> Word32)
avg = fromFun2 (\ x y -> (x + y) `divE` 2)

c :: String
c = BackEnd.compile avg
```

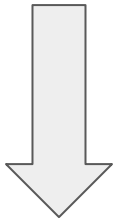


```
avg :: Exp Word32 -> Exp Word32 -> Exp Word32
avg x y = (x + y) `divE` 2

c :: String
c = FrontEnd.compile avg
```

```
Expressions
 x ∈ variables
 w ∈ word32 integers
 b      ::= False | True
 L,M,N ::= x
         | w | M + N | M - N | M * N | M / N | M % N | √ m
         | M < N | M ≡ N
         | ( M , N ) | fst L | snd L
         | < N | x ↤ 0 to M > | L[M] | len L
         | b | if L then M else N

Programs
 P ::= M | λ x. P

Expression Types
 A,B ::= Word32 | Bool | (A , B) | < A >
Program Types
 C    ::= A | A → C
```

```haskell
type instance InT (a , b) = (InT a , InT b)
instance (ExpLike a , ExpLike b) => PrgLike (a , b) where
  toPrg  u       = Exp (toExp u)
  frmPrg (Exp l) = frmExp l
instance (ExpLike a , ExpLike b) => ExpLike (a , b) where
  toExp  u = Tup (toExp (fst u)) (toExp (snd u))
  frmExp l = (frmExp (Fst l) , frmExp (Snd l))
```

```
type Point = Exp (Word32 , Word32)

distance :: Point -> Point -> Exp Word32
distance p p'
  = let x  = Fst p  ; y  = Snd p
        x' = Fst p' ; y' = Snd p'
        dx = x' - x ; dy = y' - y
    in sqrtE ((dx * dx) + (dy * dy))
```

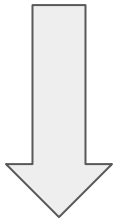


```
type Point = (Exp Word32 , Exp Word32)

distance :: Point -> Point -> Exp Word32
distance (x,y) (x',y')
  = let dx = x' - x ; dy = y' - y
    in  sqrtE ((dx * dx) + (dy * dy))
```

```
Expressions
 x ∈ variables
 w ∈ word32 integers
 b      ::= False | True
 L,M,N ::= x
         | w | M + N | M - N | M * N | M / N | M % N | √ m
         | M < N | M ≡ N
         | ( M , N ) | fst L | snd L
         | < N | x ↤ 0 to M > | L[M] | len L
         | b | if L then M else N

Programs
 P ::= M | λ x. P

Expression Types
 A,B ::= Word32 | Bool | (A , B) | < A >
Program Types
 C    ::= A | A → C
```

```
data Vec a
  = Vec { len :: Exp Word32,
          ind :: Exp Word32 -> a}
    deriving Functor

type instance InT (Vec a) = Array Word32 (InT a)
instance (ExpLike a) => PrgLike (Vec a) where
  toPrg  u       = Exp (toExp u)
  frmPrg (Exp l) = frmExp l
instance ExpLike a => ExpLike (Vec a) where
  toExp  u = Ary (len u) (\ x -> toExp  (ind u x))
  frmExp l = Vec (Len l) (\ x -> frmExp (Ind l x))
```

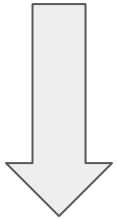
```
reverseE :: Exp (Ary Word32) -> Exp (Ary Word32)
reverseE l = Ary (Len l) (\ x -> Ind l (Len l - 1 - x))

revRev :: Exp (Ary Word32) -> Exp (Ary Word32)
revRev = reverseE . reverseE
```



```
reverseE :: Vec (Exp Word32) -> Vec (Exp Word32)
reverseE u = Vec (len u) (\ x -> ind u (len u - 1 - x))

revRev :: Vec (Exp Word32) -> Vec (Exp Word32)
revRev = reverseE . reverseE
```

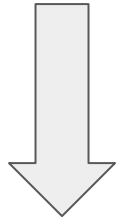
GHCi> Compile reverseE

```
AryWrd func (AryWrd v0)
{
  Wrd v3;
  AryWrd v2;
  Wrd v1;
  v1 = len (v0);
  v2 = newAry (Wrd, v1);
  v3 = 0u;
  while (ltd (v3, v1))
  {
    v2 = setAry (v2, v3, ind (v0, sub (sub (len (v0), 1u),
v3)));
    v3 = add (v3, 1u);
  }
  return v2;
}
```



GHCi> Compile revRev

```
AryWrd func (AryWrd v0)
{
  [...]
  while (ltd (v6, v4))
  { [...] }
  [...]
  while (ltd (v3, v1))
  { [...]
    while (ltd (v9, v7))
    { [...] }
    [...]
    while (ltd (v12, v10))
    { [...] }
    [...]
  }
  return v2;
}
```



GHCi> Compile reverseE

```
AryWrd func (AryWrd v0)
{
  Wrd v3;
  AryWrd v2;
  Wrd v1;
  v1 = len (v0);
  v2 = newAry (Wrd, v1);
  v3 = 0u;
  while (ltd (v3, v1))
  {
    v2 = setAry (v2, v3, ind (v0, sub (sub (len (v0), 1u),
v3)));
    v3 = add (v3, 1u);
  }
  return v2;
}
```

GHCi> Compile revRev

```
AryWrd func (AryWrd v0)
{
  Wrd v3;
  AryWrd v2;
  Wrd v1;
  v1 = len (v0);
  v2 = newAry (Wrd, v1);
  v3 = 0u;
  while (ltd (v3, v1))
  {
    v2 = setAry (v2, v3, ind (v0, sub (sub (len (v0), 1u),
sub (sub (len (v0), 1u), v3))));
    v3 = add (v3, 1u);
  }
  return v2;
}
```

```
 reverseE . reverseE
=
 \ l -> let l' = reverseE l
        in  reverseE l'
=
 \ l ->
 let l' = Ary (Len l) (\ x -> Ind l (Len l - 1 - x))
 in  Ary (Len l') (\ x -> Ind l (Len l' - 1 - x)) .
```

```
 reverseE . reverseE
=
 \ u -> let u' = reverseE u
        in  reverseE u'
=
 \ u ->
 let u' = Vec (len u) (\ x -> ind u (len u - 1 - x))
 In  Vec (len u') (\ x -> ind u' (len u' - 1 - x))
= {- len (Vec M N) --> M -}
 \ u ->
 let u' = Vec (len u) (\ x -> ind u (len u - 1 - x))
 In  Vec (len u) (\ x -> ind u' (len u - 1 - x))
= {- ind (Vec M N) --> N -}
 \ u ->
 let f = (\ x -> ind u (len u - 1 - x))
 In  Vec (len u) (\ x -> f (len u - 1 - x))
=
 Vec (len u) (\ x -> ind u (len u - 1 - (len u - 1 - x)))
```

Whenever two such functions are composed, the intermediate vector is guaranteed to be eliminated. This guarantee by far exceeds guarantees given by conventional optimizing compilers.

Svenningsson

&

Axelsson

Combining deep and shallow embedding for EDSL,
TFP 2012

Expressions

```
 x ∈ variables
 w ∈ word32 integers
 b      ::= False | True
 L,M,N ::= x
         | w | M + N | M - N | M * N | M / N | M % N | √ m
         | M < N | M ≡ N
         | ( M , N ) | fst L | snd L
         | < N | x ↤ 0 to M > | L[M] | len L
         | b | if L then M else N
```

Programs

```
 P ::= M | λ x. P
```

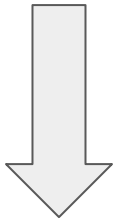
Expression Types

```
 A,B ::= Word32 | Bool | (A , B) | < A >
```

Program Types

```
 C   ::= A | A → C
```

```
not :: Exp Bool -> Exp Bool
not x = Cnd x (LitB False) (LitB True)
```



Or

```
not :: Bool -> Bool
not True  = False
not False = True
```

Or

```
not :: Bool -> Bool
not x = if x
        then False
        else True
```

```
not :: Bool -> Bool
not x = case x
          True  -> False
          False -> True
```

```
type instance InT Bool = Bool
instance PrgLike Bool where
  toPrg  u        = Exp (toExp u)
  frmPrg (Exp l) = frmExp l
instance ExpLike Bool where
  toExp  u = if u then LitB True else LitB False
  frmExp l = Cnd l ? ?
```

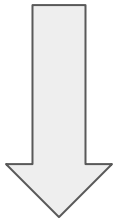
Control flow is problematic and cannot be used directly [in this setting]

Gill

Domain-specific languages and code synthesis using Haskell
Queue, *12*(4), 30

```
pattern TrueE :: () => a ~ Bool => Exp a
pattern TrueE = LitB True

pattern FalseE :: () => a ~ Bool => Exp a
pattern FalseE = LitB False

(?) :: ExpLike a => Exp Bool -> (a , a) -> a
l ? (v , w) = frmExp (Cnd l (toExp v) (toExp w))
```

```
not :: Exp Bool -> Exp Bool
not x = Cnd x (LitB False) (LitB True)
```



```
not :: Exp Bool -> Exp Bool
not x = x ? (FalseE , TrueE)
```

[...] one may be tempted to make a Syntactic instance for *Maybe*. Unfortunately, there is no way to make this work [in this setting]

Svenningsson

&

Axelsson

Combining deep and shallow embedding for EDSL, TFP 2012

Embedding DSLs With Code Generation

$$
\begin{aligned}
A,B \;::=\; & \mathbf{Exp}\ \alpha \\
\mid\; & A \rightarrow B \\
\mid\; & A \times B
\end{aligned}
$$

Embedding DSLs With Code Generation

$$
\begin{aligned}
A,B \;::=\; & \mathbf{Exp}\ \alpha \\
\mid\; & A \rightarrow B \\
\mid\; & A \times B \\
\mid\; & A + B
\end{aligned}
$$

```
Bool → Exp C
```

⇒

```
Exp (Bool → C)
```

```
λ x →  if x then M else
N
```

⇒

```
Abs "x"
  (Cnd (Var "x") ⌞M⌟ ⌞N⌟)
```

```
frmExp :: Exp Bool → Bool
frmExp l = ???
```

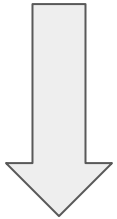
```
Either (Exp A)(Exp B) → Exp
                  C
                  ⇒
  Exp (Either A B → C)
λ z → case z of
          {Left  x → M;
           Right y → N}
                  ⇒
Abs "z"
(Cas (Var "z")
   [ Abs "x" ⌜M⌝ ,
     Abs "y" ⌜N⌝ ])
```

```
frmExp :: Exp (Either A B) →
          Either (Exp A)(Exp B)
frmExp l = ???
```

```
frmExp :: Exp (Either A B) →
          Either (Exp A)(Exp B)
frmExp l = ???

evalEither :: Exp (Either A B) →
              Either A B
```

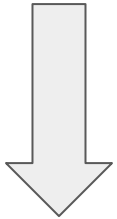
Solution

```
frmExp :: Exp Bool → Bool
frmExp l = ???
```



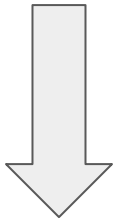
```
frmExp :: Exp Bool → Cont Exp Bool
frmExp l =
  shift (λ k → Cnd l (k True) (k False))
```

```
frmExp :: Exp Bool → Bool
frmExp l = ???
```



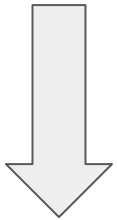
```
frmExp :: Exp Bool → Cont Exp Bool
frmExp l =
  shift (λ k → Cnd l (k True) (k False))
```

```
frmExp :: Exp (Either A B) →
          Either (Exp A)(Exp B)
frmExp l = ???
```



```
frmExp :: Exp (Either A B) →
          Cont Exp (Either (Exp A)(Exp B))
frmExp l = shift (λ k →
 Cas l
  (Abs (λ x → reset ((k ∘ Left)  <$> reflect x)))
  (Abs (λ y → reset ((k ∘ Right) <$> reflect y))))
```

```
frmExp :: Exp (Either A B) →
          Either (Exp A)(Exp B)
frmExp l = ???
```



```
frmExp :: Exp (Either A B) →
          Cont Exp (Either (Exp A)(Exp B))
frmExp l = shift (λ k →
 Cas l
  (Abs (λ x → reset ((k ∘ Left)  <$> reflect x)))
  (Abs (λ y → reset ((k ∘ Right) <$> reflect y))))
```

Where is this solution coming from?

Where is this ~~solution~~ dark magic coming from?

Embedding By Normalisation (EBN)

EBN is based on the correspondence to NBE

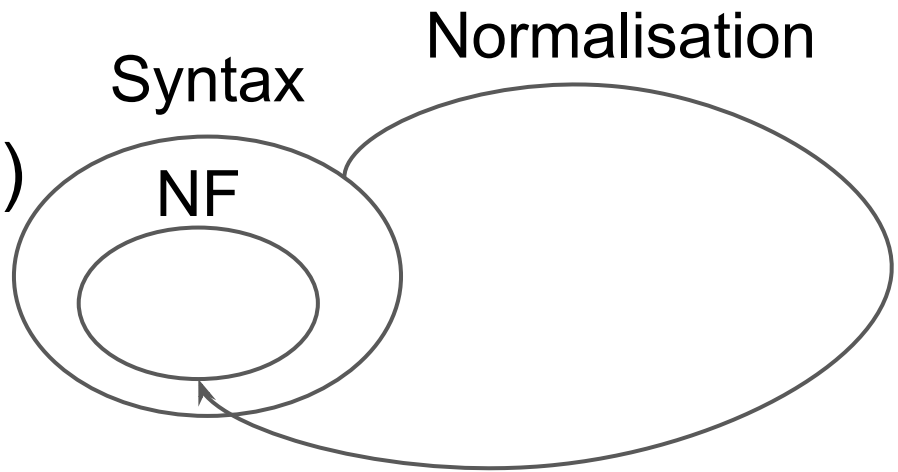
Normalisation
By
Evaluation
(NBE)

Embedding
By
Normalisation
(EBN)

Normalisation By Evaluation (NBE)

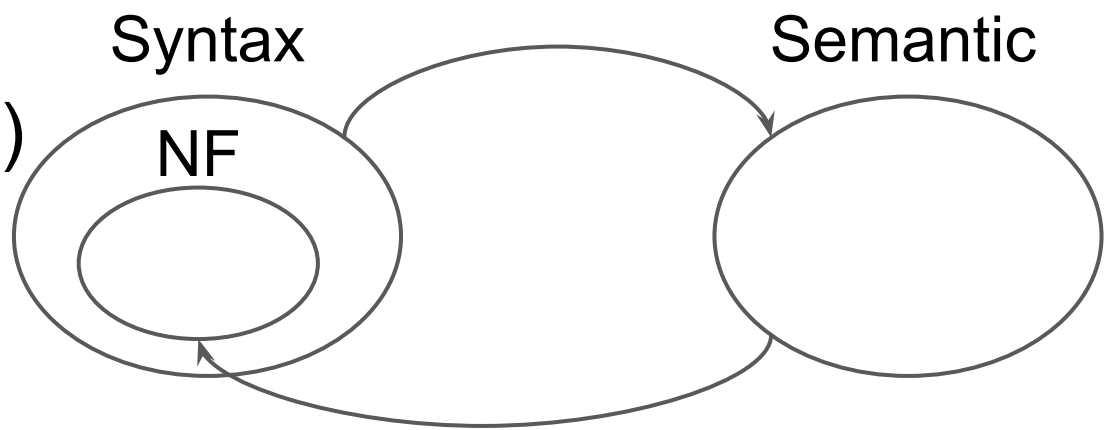
Normalisation By Evaluation (NBE)

- Syntactic Domain
- Normal Forms (NF)



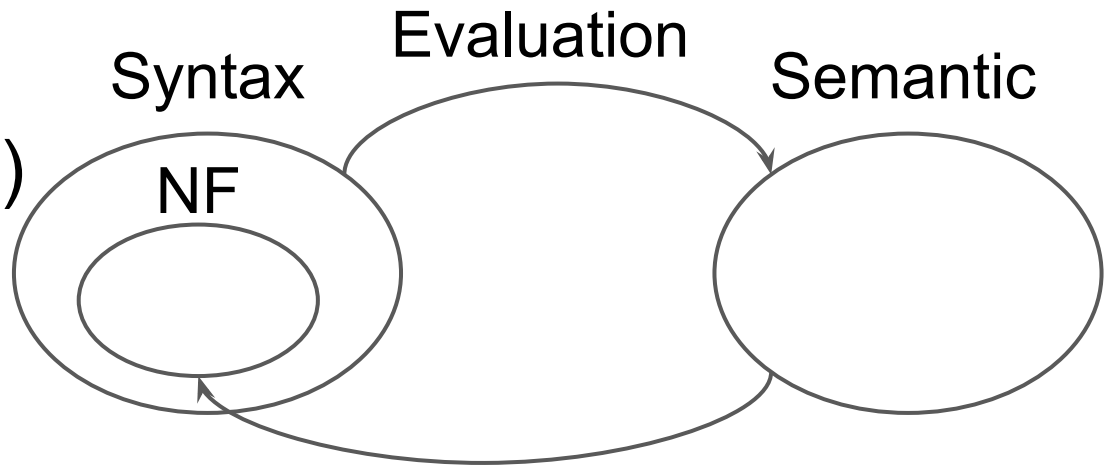
Normalisation By Evaluation (NBE)

- Syntactic Domain
- Normal Forms (NF)
- Semantic Domain



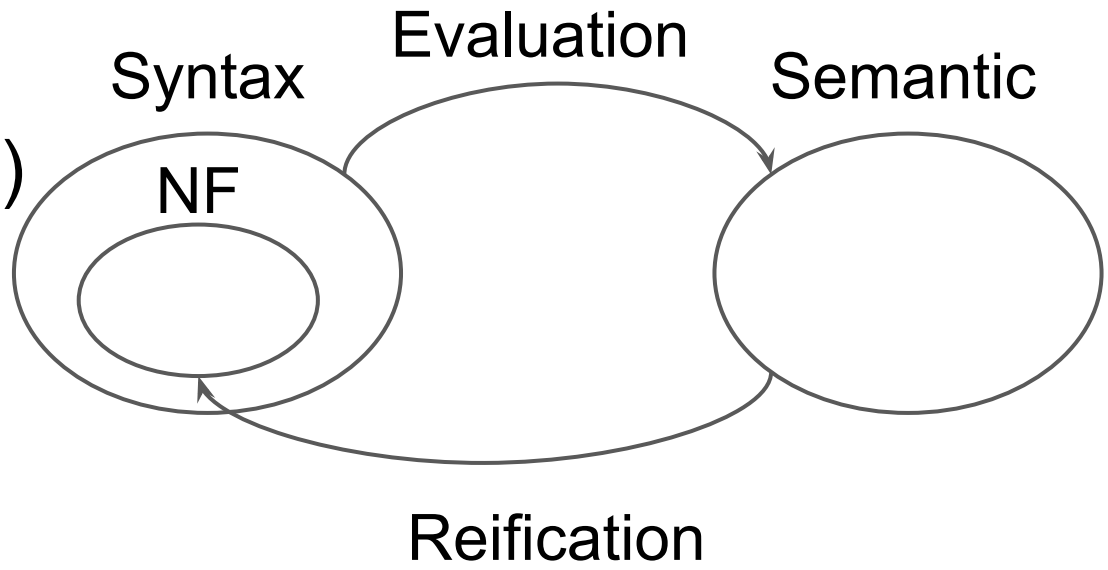
Normalisation By Evaluation (NBE)

- Syntactic Domain
- Normal Forms (NF)
- Semantic Domain
- Evaluation (⟦_⟧)



Normalisation By Evaluation (NBE)

- Syntactic Domain
- Normal Forms (NF)
- Semantic Domain
- Evaluation (⟦_⟧)
- Reification (↓_)

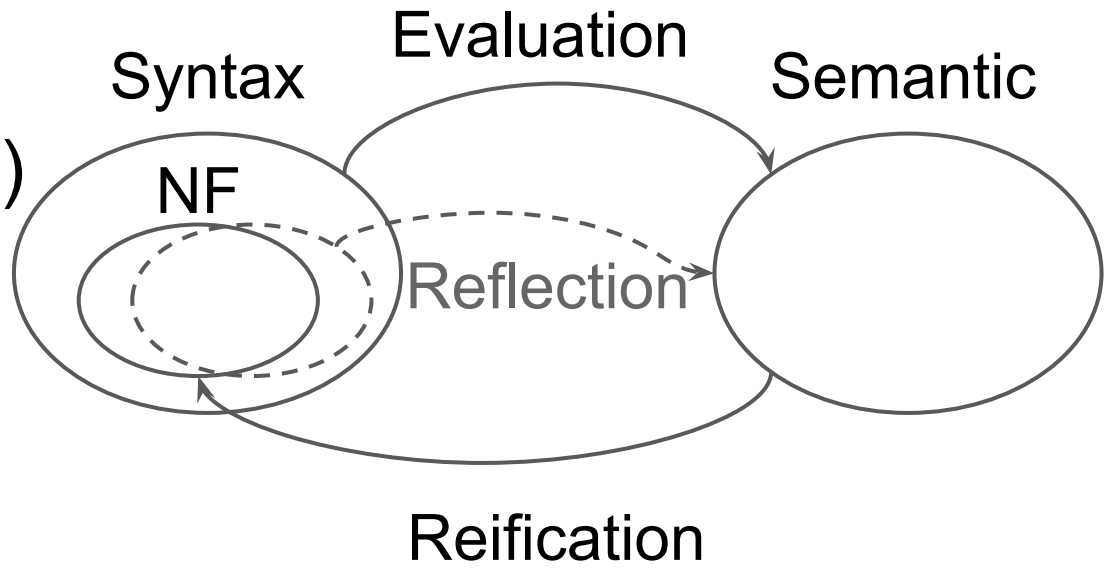


Normalisation By Evaluation (NBE)

- Syntactic Domain
- Normal Forms (NF)
- Semantic Domain
- Evaluation (⟦_⟧)
- Reification (↓_)

+

Reflection (↑_)



Expressions

x ∈ variables

i ∈ integers

L,M,N ∈ Exp ::= x

| i | M + N

| λ x. N | L M

| (M , N) | fst L | snd L

Expression Types

A,B,C ::= Int | A → B | A * B

```haskell
type a :*: b = (a , b)

data Exp a where
  Lit :: Int -> Exp Int
  Add :: Exp Int -> Exp Int -> Exp Int
  Abs :: (Exp a -> Exp b) -> Exp (a -> b)
  App :: Exp (a -> b) -> Exp a -> Exp b
  Tup :: Exp a -> Exp b -> Exp (a :*: b)
  Fst :: Exp (a :*: b) -> Exp a
  Snd :: Exp (a :*: b) -> Exp b
  Val :: a -> Exp a
  Var :: String -> Exp a
```

```
⟦ Int ⟧     =  Exp Int
⟦ A → B ⟧ =  ⟦ A ⟧ → ⟦ B ⟧
⟦ A * B ⟧ =  ⟦ A ⟧ * ⟦ B ⟧

↓ : (α : Type) →  ⟦ α ⟧ → Exp α
↓ Int       U = U
↓ (A → B) U = Abs (λ x. ↓ B (U (↑ A x)))
↓ (A * B) U = Tup (↓ A (fst U)) (↓ B (snd U))

↑ : (α : *) → Exp α →  ⟦ α ⟧
↑ Int       L = L
↑ (A → B) L = λ x. ↑ B (App L (↓ A x))
↑ (A * B) L = (↑ A (Fst L) , ↑ B (Snd L))
```

```haskell
type family InT a
type instance InT (Exp a)       = a
type instance InT (a -> b)      = InT a -> InT b
type instance InT (a , b)       = (InT a , InT b)

class Reify a where
  reify :: a -> Exp (InT a)
instance Reify (Exp a) where
  reify u = u
instance (Reflect a , Reify b) => Reify (a -> b) where
  reify u = Abs (\ x -> reify (u (reflect x)))
instance (Reify a , Reify b) => Reify (a , b) where
  reify u = Tup (reify (fst u)) (reify (snd u))

class Reflect a where
  reflect :: Exp (InT a) -> a
instance Reflect (Exp a) where
  reflect l = l
instance (Reify a , Reflect b) => Reflect (a -> b) where
  reflect l = \ x -> reflect (App l (reify x))
instance (Reflect a , Reflect b) => Reflect (a , b) where
  reflect l = (reflect (Fst l) , reflect (Snd l))
```

Looks Familiar?

```
type family InT a
class PrgLike a where
  toPrg  :: a -> Prg (InT a)
  frmPrg :: Prg (InT a) -> a
class Type (InT a) => ExpLike a where
  toExp  :: a -> Exp (InT a)
  frmExp :: Exp (InT a) -> a
```

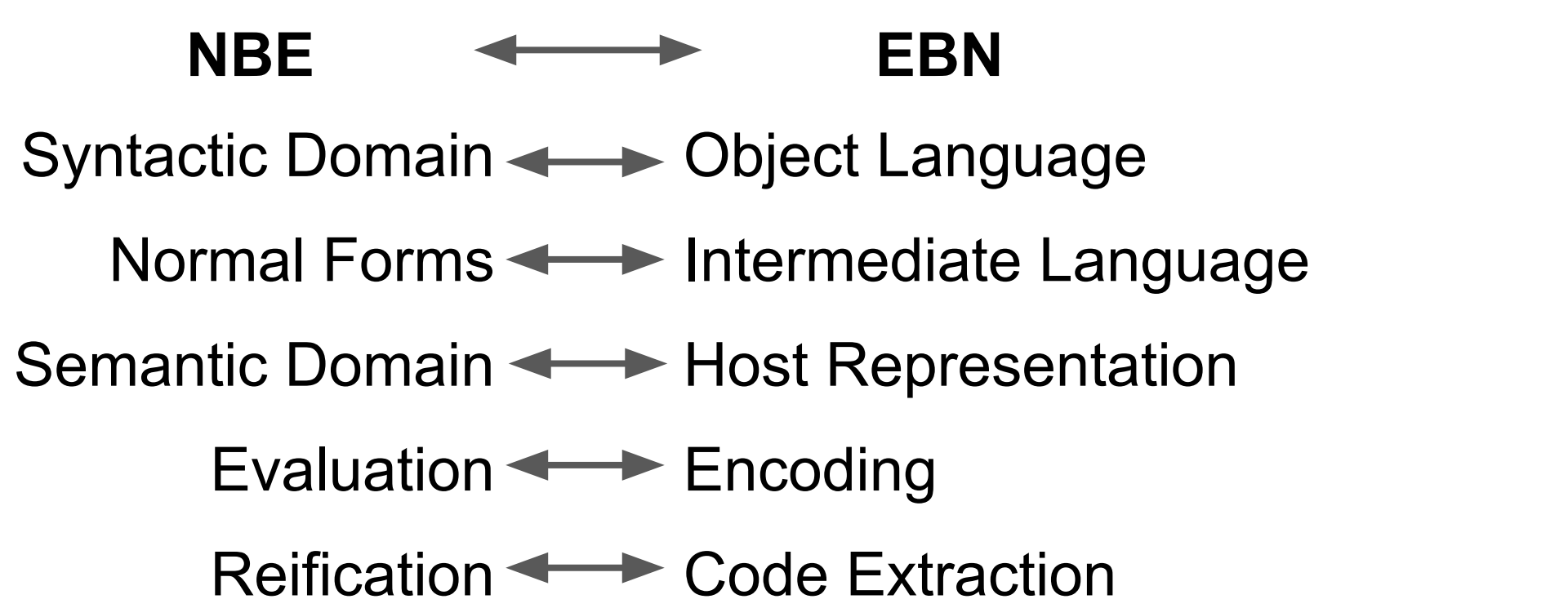
```haskell
type instance InT (Prg a) = a
instance PrgLike (Prg a) where
  toPrg  = id
  frmPrg = id

type instance InT (Exp a) = a
instance Type a => ExpLike (Exp a) where
  toExp  = id
  frmExp = id
instance Type a => PrgLike (Exp a) where
  toPrg = Exp
  frmPrg (Exp m) = m
  frmPrg (Fun _) = case typ :: STyp a of {}
                   -- impossible

type instance InT (a -> b) = InT a -> InT b
instance (ExpLike a, PrgLike b) => PrgLike (a -> b) where
  toPrg  f       = Fun (toPrg . f . frmExp)
  frmPrg (Exp m) = case typOf m of {}
                   -- impossible
  frmPrg (Fun f) = frmPrg . f . toExp
```

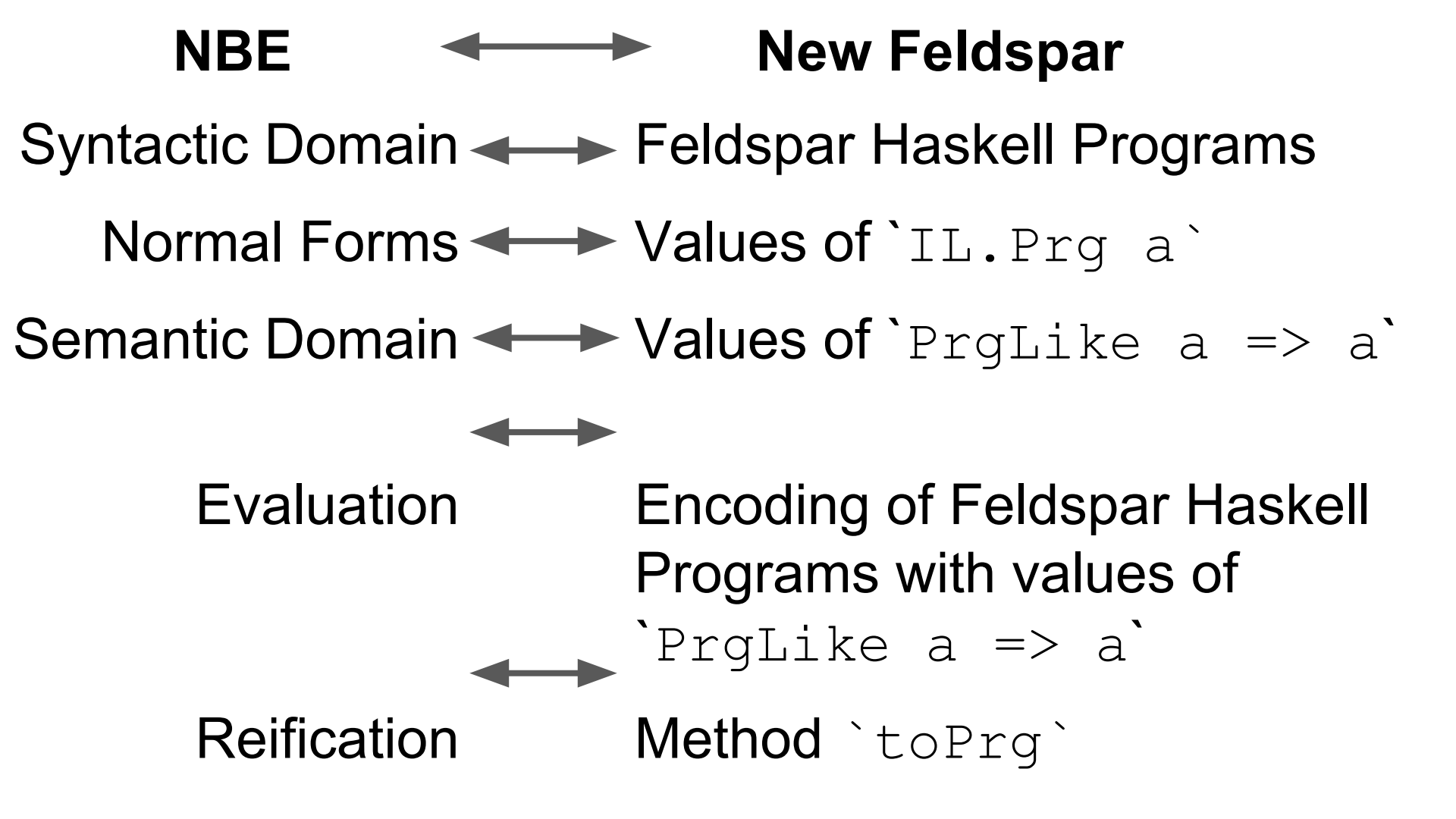
```haskell
type instance InT (a , b) = (InT a , InT b)
instance (ExpLike a , ExpLike b) => PrgLike (a , b) where
  toPrg  u       = Exp (toExp u)
  frmPrg (Exp l) = frmExp l
instance (ExpLike a , ExpLike b) => ExpLike (a , b) where
  toExp  u = Tup (toExp (fst u)) (toExp (snd u))
  frmExp l = (frmExp (Fst l) , frmExp (Snd l))
```

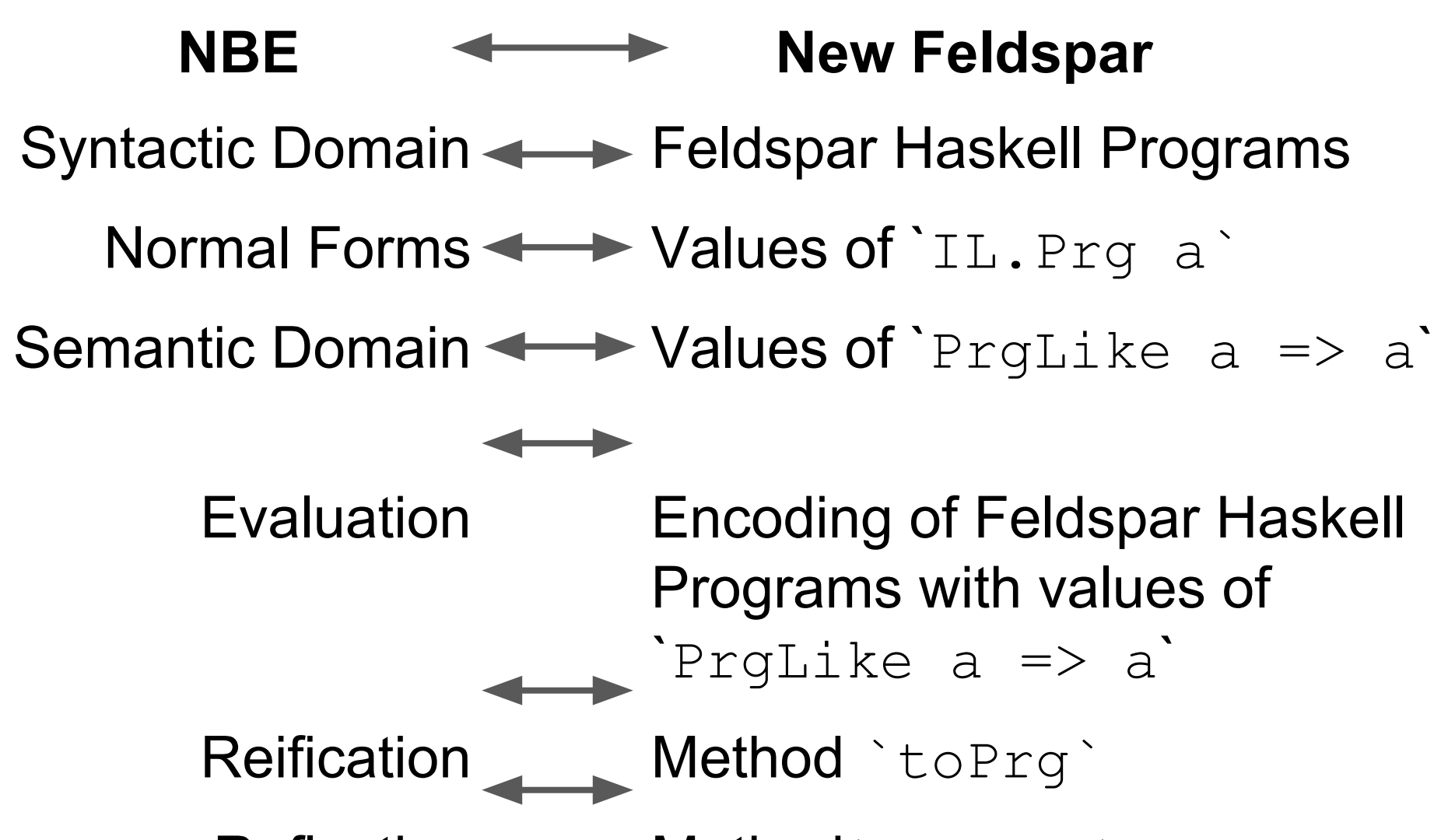
EBN is based on the correspondence to NBE

Normalisation
By
Evaluation
(NBE)

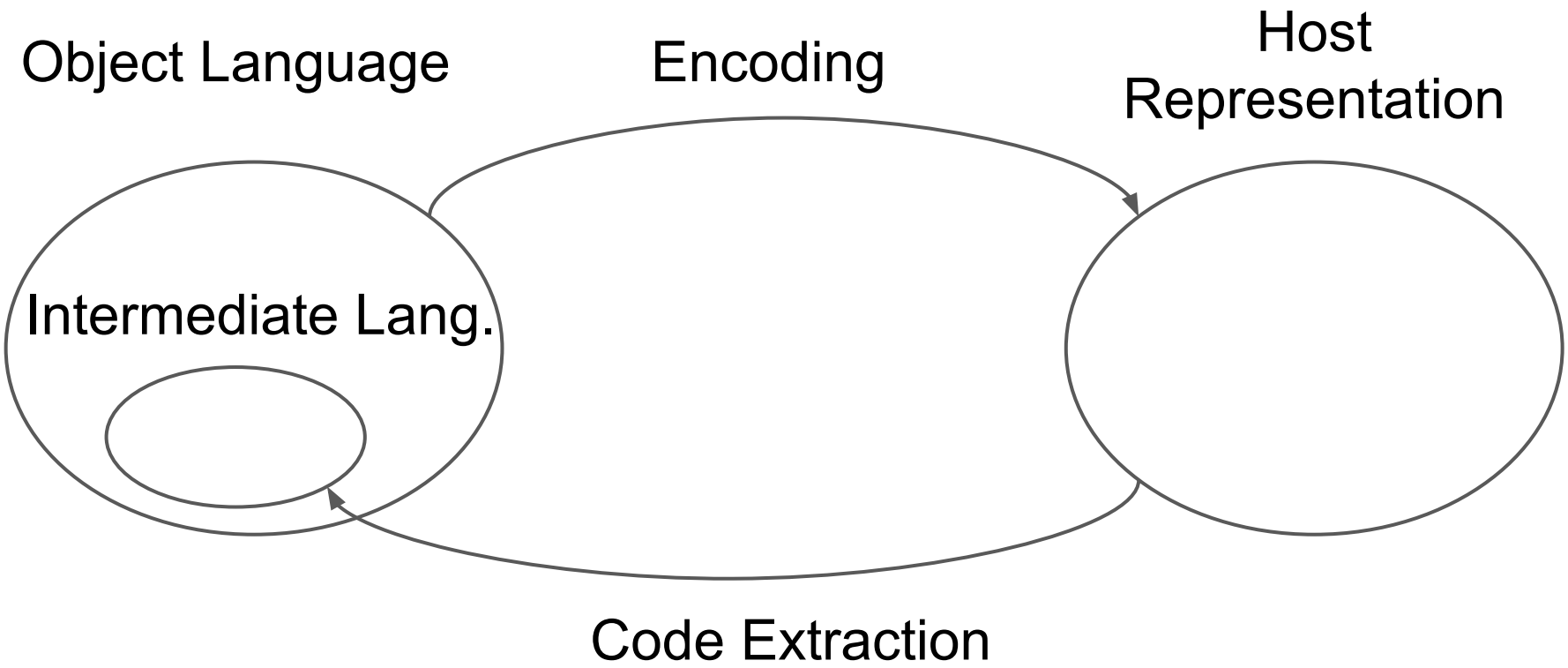
Embedding
By
Normalisation
(EBN)


NBE
EBN
Syntactic Domain
Object Language
Normal Forms
Intermediate Language
Semantic Domain
Host Representation
Evaluation
Encoding
Reification
Code Extraction

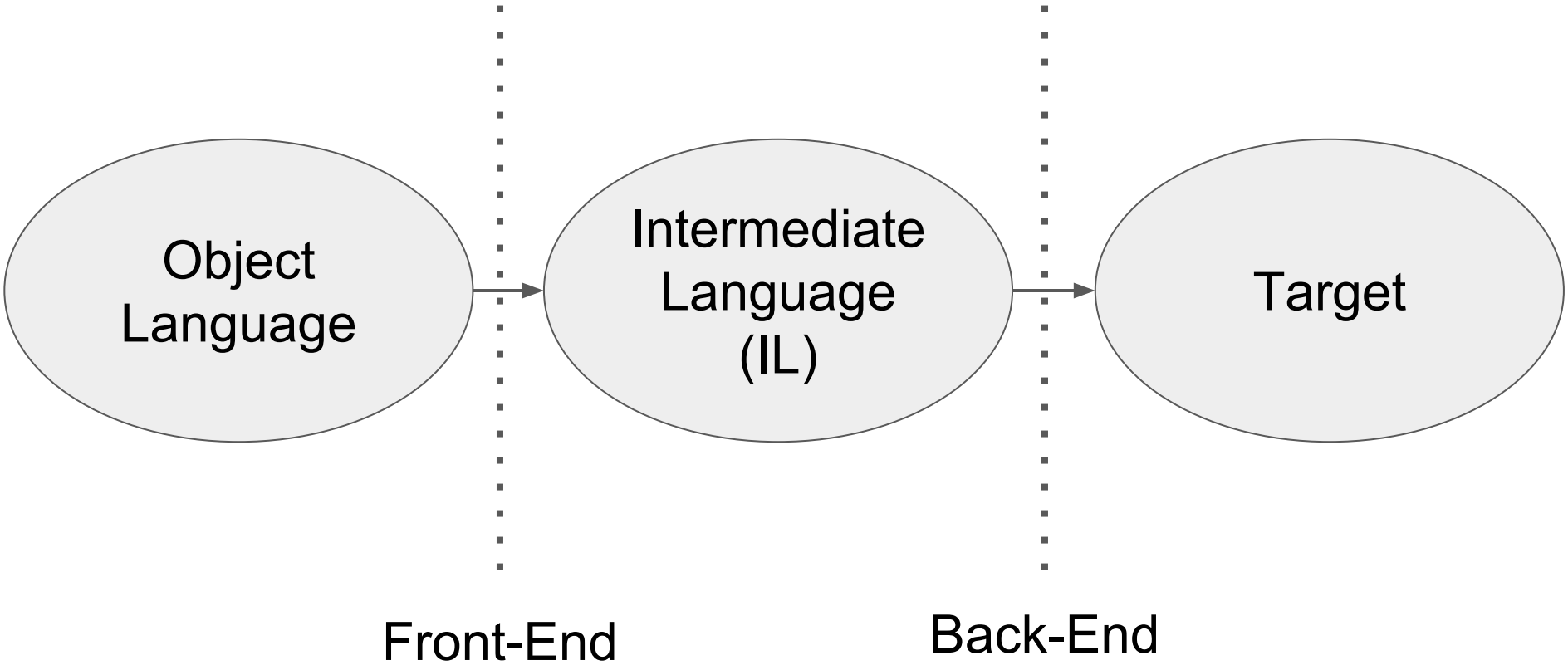

NBE
New Feldspar
Syntactic Domain
Feldspar Haskell Programs
Normal Forms
Values of `IL.Prg a`
Semantic Domain
Values of `PrgLike a => a`
Evaluation
Encoding of Feldspar Haskell Programs with values of `PrgLike a => a`
Reification
Method `toPrg`


NBE
New Feldspar
Syntactic Domain
Feldspar Haskell Programs
Normal Forms
Values of `IL.Prg a`
Semantic Domain
Values of `PrgLike a => a`
Evaluation
Encoding of Feldspar Haskell Programs with values of `PrgLike a => a`
Reification
Method `toPrg`

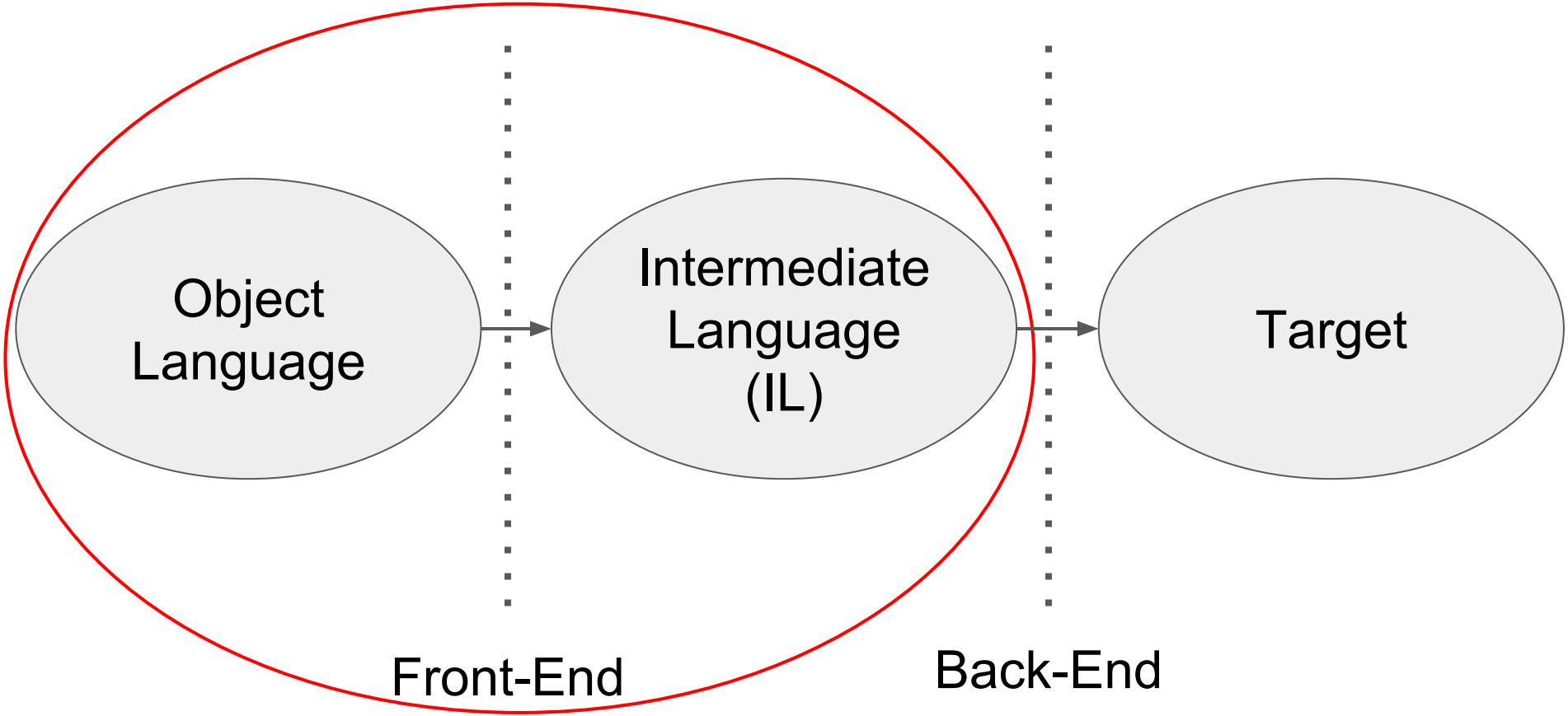
Embedding By Normalisation



Embedding DSLs With Code Generation

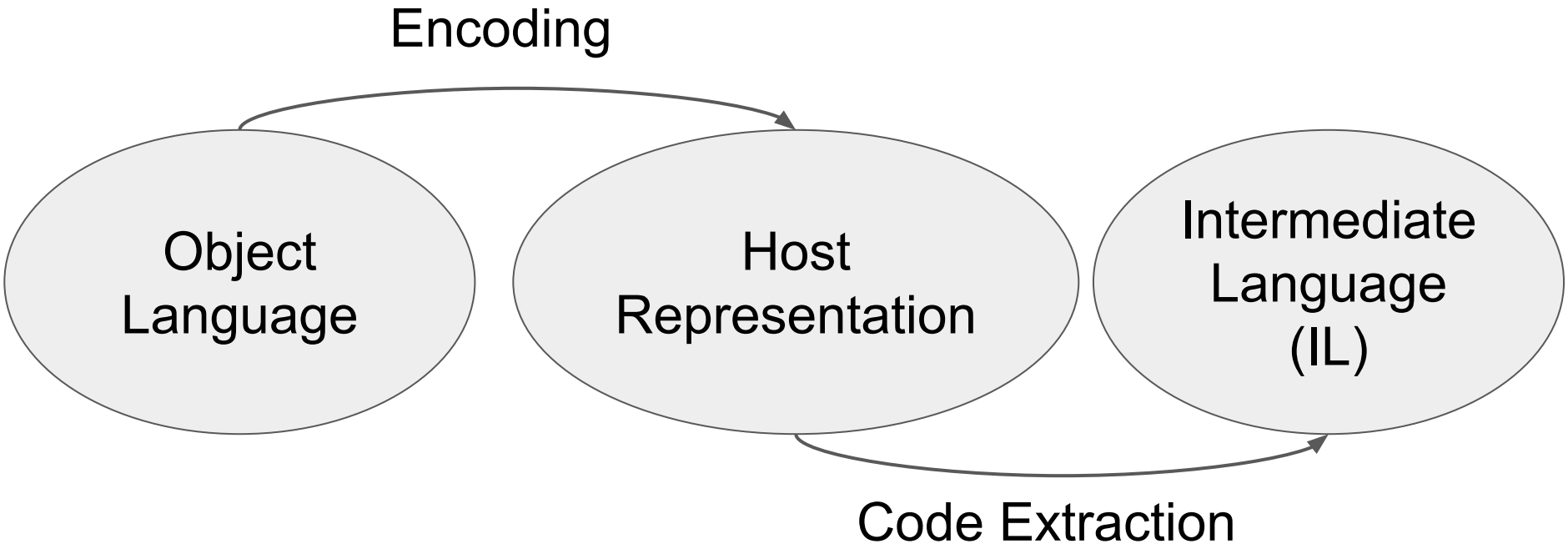


Embedding DSLs With Code Generation

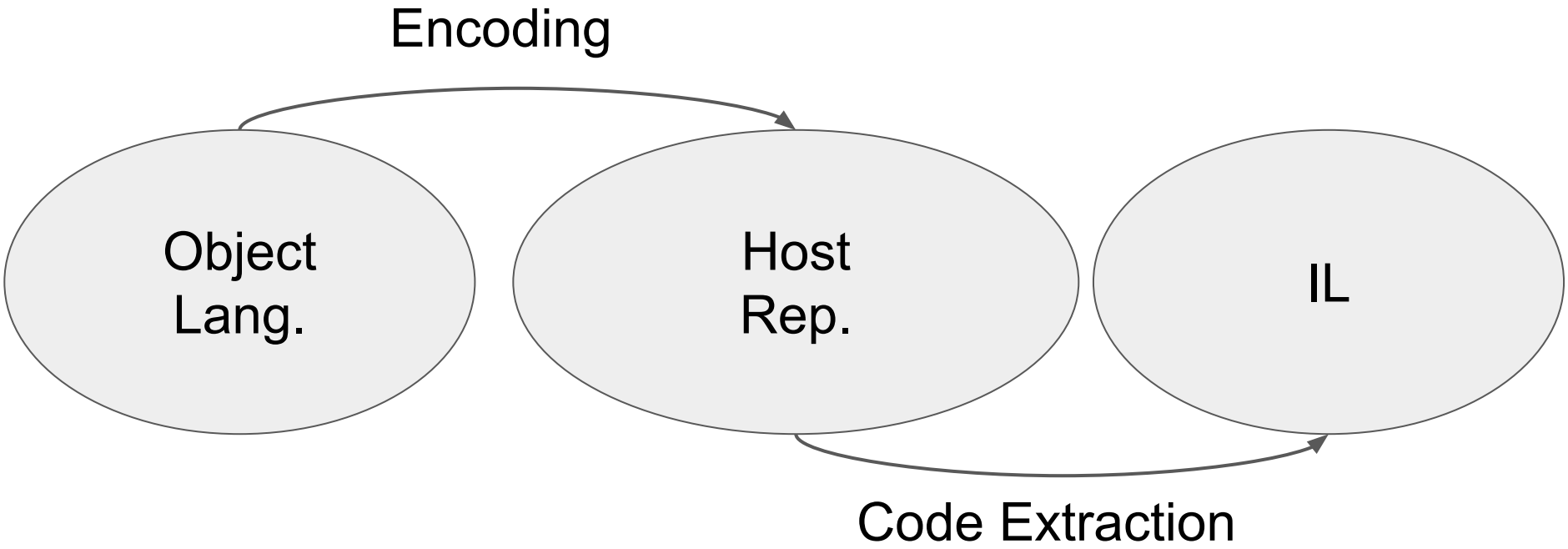


Embedding By Normalisation

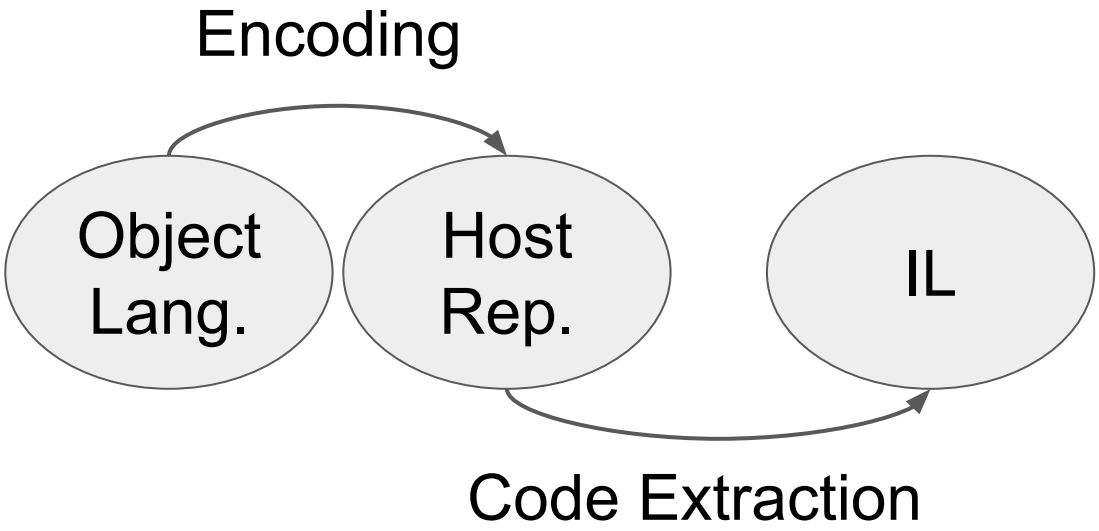
Embedding By Normalisation



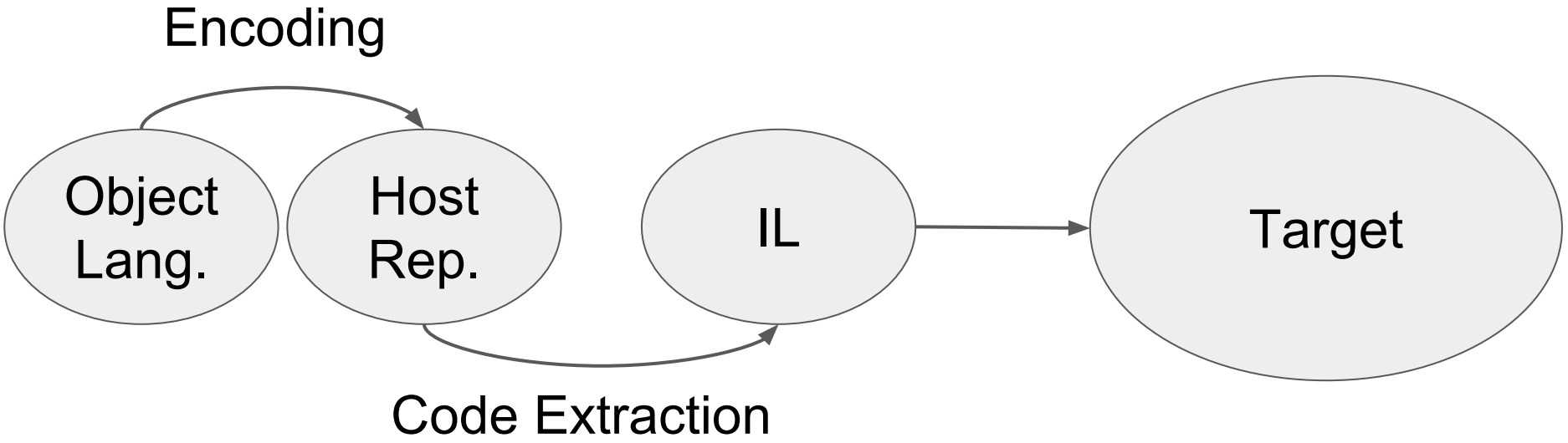
Embedding By Normalisation



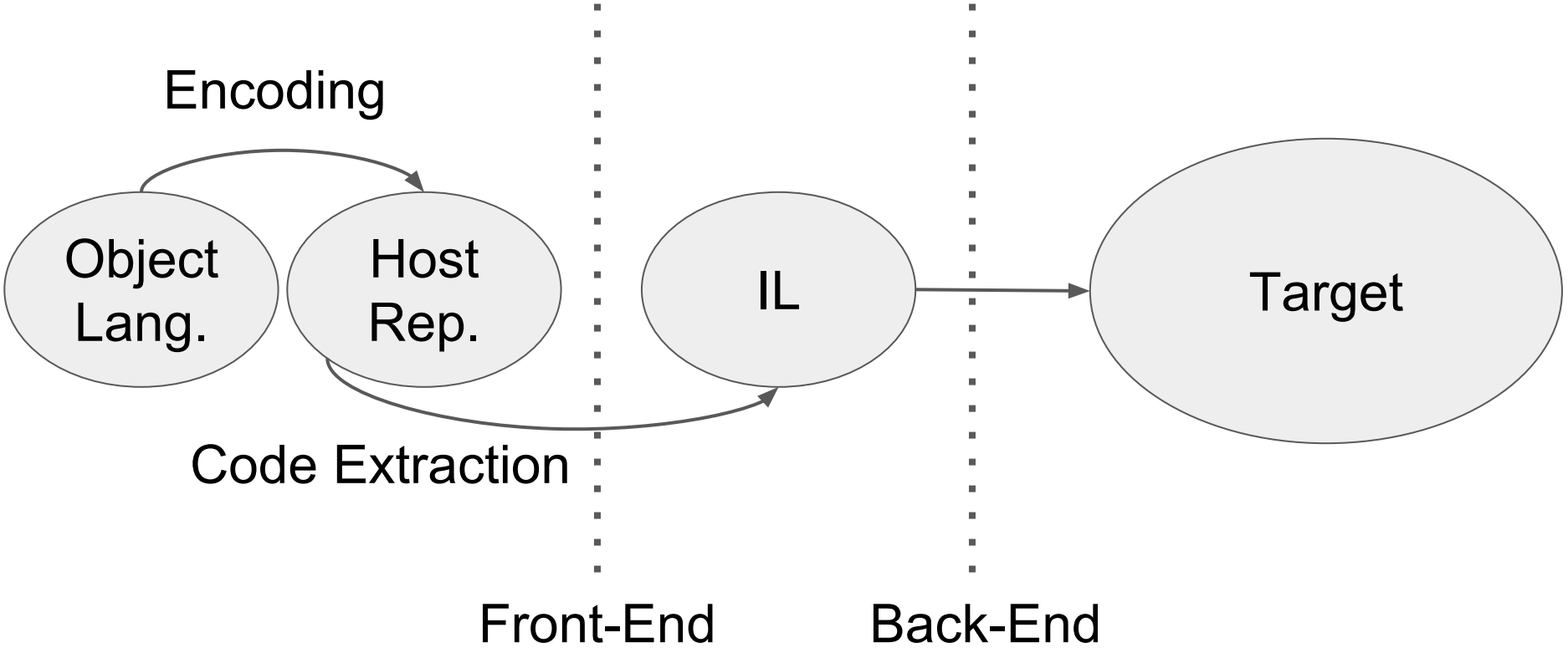
Embedding By Normalisation



Embedding By Normalisation



Embedding By Normalisation



Expressions

$x \in$ variables

$i \in$ integers

$L,M,N \in$ Exp ::= x
| i | $M + N$
| $\lambda\, x.\ N$ | $L\ M$
| $(\ M\ ,\ N\ )$ | fst L | snd L
| True | False | if L then M else N
| InL M | InR N | case L M N

Expression Types

A,B,C ::= Int | $A \to B$ | $A * B$ | $A + B$

```
type family InT a
type instance InT (Exp a)       = a
type instance InT (a -> b)      = InT a -> InT b
type instance InT (a :*: b)     = InT a :*: InT b
type instance InT Bool          = Bool
type instance InT (a :+: b)     = InT a :+: InT b
type instance InT (Cont Exp a) = InT a

class Reify a where
  reify :: a -> Exp (InT a)
instance Reify (Exp a) where
  reify = id
instance (Reflect a , Reify b) => Reify (a -> b) where
  reify u = Abs (\ x -> reset (reify <$> (u <$> (reflect x))))
instance (Reify a , Reify b) => Reify (a :*: b) where
  reify u = Tup (reify (fst u)) (reify (snd u))
instance Reify Bool where
  reify u = if u then Tru else Fal
instance (Reify a, Reify b) => Reify (a :+: b) where
  reify u = case u of
              Left  x -> InL (reify x)
              Right y -> InR (reify y)
instance Reify a => Reify (Cont Exp a) where
  reify u = reset (fmap reify u)
```

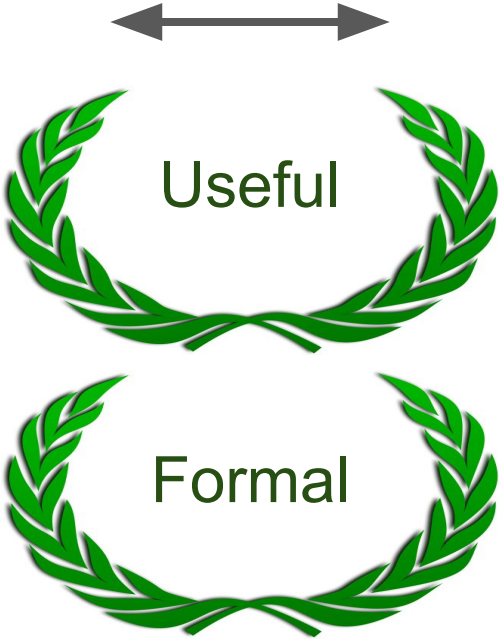
```haskell
class Reflect a where
  reflect :: Exp (InT a) -> Cont Exp a
instance Reflect (Exp a) where
  reflect   = return
instance (Reify a , Reflect b) => Reflect (a -> Cont Exp b) where
  reflect l = return (\ x -> reflect (App l (reify x)))
instance (Reflect a , Reflect b) => Reflect (a :*: b) where
  reflect l = ((,)) <$> reflect (Fst l) <*> reflect (Snd l)
instance Reflect Bool where
  reflect l = shift (\ k -> Cnd l (k True) (k False))
instance (Reflect a , Reflect b) => Reflect (a :+: b) where
  reflect l = shift (\ k ->
                Cas l
                (Abs (\ x -> reset ((k . Left)  <$> reflect x)))
                (Abs (\ y -> reset ((k . Right) <$> reflect y))))
```

```
*Live> reify (id :: Bool -> Bool)
Abs (\ x0 -> (Cnd (x0) (Tru) (Fal)))

*Live> reify (not :: Bool -> Bool)
Abs (\ x0 -> (Cnd (x0) (Fal) (Tru)))
```

Normalisation
By
Evaluation
(NBE)

Embedding
By
Normalisation
(EBN)

Normalisation By Evaluation (NBE)



Embedding By Normalisation (EBN)

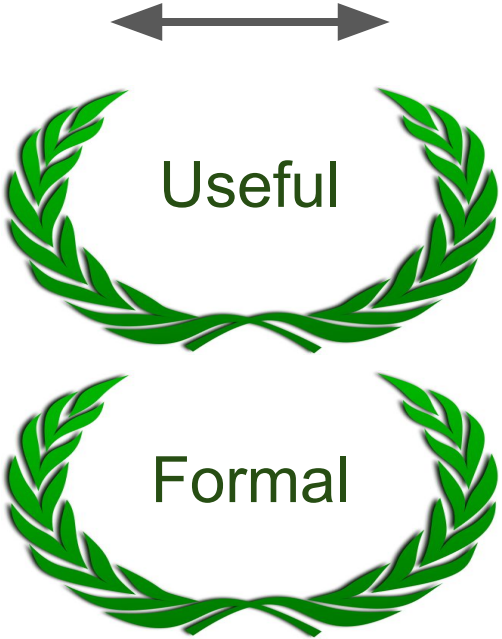
Theorem (Soundness)

If $\Gamma \vdash M \simeq N : A$, *then* $[\![M]\!] = [\![N]\!]$.

Theorem (Completeness)

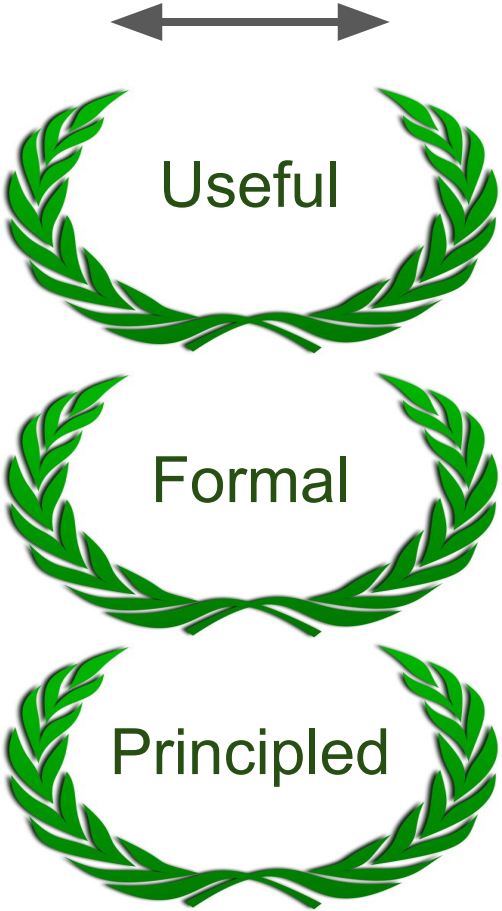
If $[\![M]\!] = [\![N]\!]$, *then* $\Gamma \vdash M \simeq N : A$.

Normalisation By Evaluation (NBE)



Embedding By Normalisation (EBN)

Normalisation By Evaluation (NBE)



Embedding By Normalisation (EBN)

1998 APPSEM Workshop on Normalization by Evaluation,

NBE '98,

(rg, Sweden, May

Peter Dybjer

Olivier Danvy

Same Pattern, Different Contexts and Different People

*Inverse of Evaluation Functio*n
Proof Theory
(Berger, Eberl, Schwichtenberg)

Type-Directed Partial Evaluation
Partial Evaluation
(Danvy)

Normalisation By Intuitionistic Model Construction
Type Theory
(Dybjer)

Normalisation By
Evaluation
Category
Theory
(Reynolds,Streicher)

Reduction-Free Normalisation
Logic
(Altenkirch)

Same Pattern, Different Contexts and Different People

Inverse of Evaluation Function
Proof Theory
(Berger, Eberl, Schwichtenberg)

Type-Directed Partial Evaluation
Partial Evaluation
(Danvy)

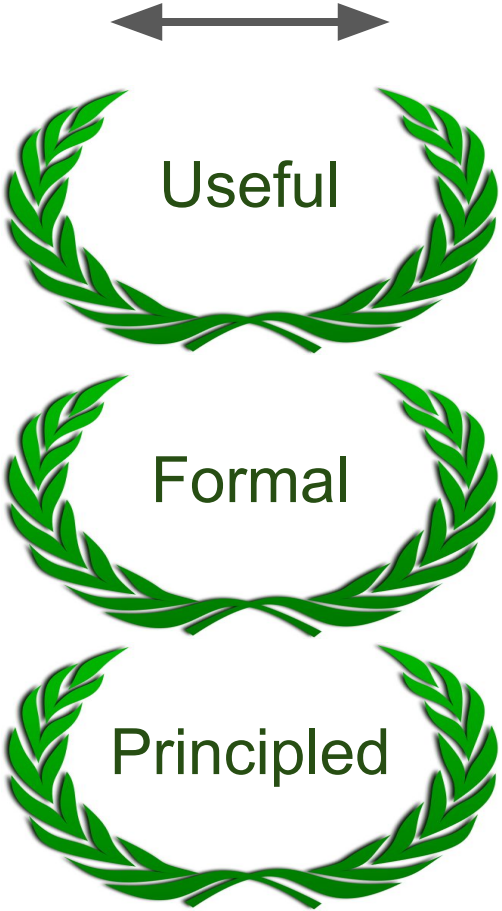
Normalisation By Intuitionistic Model Construction
Type Theory
(Dybjer)

Normalisation By Evaluation
Category Theory
(Reynolds Streicher)

Embedding By Normalisation
Embedded DSLs
(Najd et al)

Reduction-Free Normalisation
Logic
(Altenkirch)

Normalisation By Evaluation (NBE)



Embedding By Normalisation (EBN)

Conclusion

Embedding By Normalisation (EBN)

=

Supervisor:
Philip Wadler

Co-Supervisor:
Sam Lindley

Co-Supervisor:
Josef Svenningsson

Thank you!