Practical Normalization by Evaluation for EDSLs

NACHIAPPAN VALLIAPPAN

ALEJANDRO RUSSO

SAM LINDLEY

CHALMERS UNIVERSITY OF TECHNOLOGY

OCTOPI

THE UNIVERSITY OF EDINBURGH

```haskell
data Exp a where
  Lift :: Int -> Exp Int
  Add  :: Exp Int -> Exp Int -> Exp Int
  Mul  :: Exp Int -> Exp Int -> Exp Int
  Var  :: String -> Exp a
```

Extending Syntax with Arrays – Deep Embedding

```
data Exp a where …
  NewArr :: Exp Int -> (Exp Int -> Exp a) -> Exp (Arr a)
  LenArr :: Exp (Arr a) -> Exp Int
  IxArr  :: Exp (Arr a) -> Exp Int -> Exp a
```

Mapping over Arrays – Deep Embedding

```
mapArr_D :: (Exp a -> Exp b) -> Exp (Arr a) -> Exp (Arr b)
mapArr_D f arr = NewArr (LenArr arr) (f ∘ IxArr arr)
```

However, `mapArr_D` lacks map-map fusion:

```
mapArr_D f (mapArr_D g) arr /= mapArr_D (f ∘ g) arr
```

Using Semantic Arrays – Shallow Embedding

```
data Arr a where
  Arr :: Exp Int -> (Exp Int -> a) -> Arr a

newArrS :: Exp Int -> (Exp Int -> Exp a) -> Arr (Exp a)
newArrS = Arr

lenArrS :: Arr (Exp a) -> Exp Int
lenArrS (Arr n _) = n

ixArrS  :: Arr (Exp a) -> Exp Int -> Exp a
ixArrS (Arr _ ixf) = ixf
```

Mapping over Arrays – Shallow Embedding

```
mapArrS :: (Exp a -> Exp b) -> Arr (Exp a) -> Arr (Exp b)
mapArrS f arr = newArrS (lenArrS arr) (f ∘ ixArrS arr)
```

`mapArrS` obeys map-map fusion!

```
mapArrS f (mapArrS g) arr == mapArrS (f ∘ g) arr
```

Combining Deep and Shallow Embedding

```
toExp_D    :: Arr (Exp a) -> Exp (Arr a)
fromExp_S :: Exp (Arr a) -> Arr (Exp a)
  -- fromExp_S ∘ toExp_D == id
```

```
mapArr_DS :: (Exp a -> Exp b) -> Exp (Arr a) -> Exp (Arr b)
mapArr_DS f arr = toExp_D (mapArr_S f (fromExp_S arr))
```

Recovers syntax

Enables fusion in semantics

Converts to semantics

```
mapArr_DS f (mapArr_DS g) arr == mapArr_DS (f ∘ g) arr
```

Fusion on syntax

Problems Combining Deep and Shallow Embedding

`Arr (Exp a)`	⟷	`Exp (Arr a)`
`(Exp a, Exp b)`	⟷	`Exp (a,b)`
`Exp a -> Exp b`	⟷	`Exp (a -> b)`

`Int`	→ ~~←~~	`Exp Int`
`Either (Exp a) (Exp b)`	→ ~~←~~	`Exp (Either a b)`
`Err (Exp a)`	→ ~~←~~	`Exp (Err a)`

Unknowns (Var :: String -> Exp a) disrupt this harmony for ***types with multiple introduction forms***

Refined Shallow Embedding?

`Either (Exp Int) Int` ⟷ `Exp Int`

Expressions with unknowns

- Shallow embedding is no longer “natural”, cannot reuse host language features
- Difficult to distinguish between object (syntax) and host (semantics) languages
- Semantic domain isn’t precise, making it difficult to reason about correctness

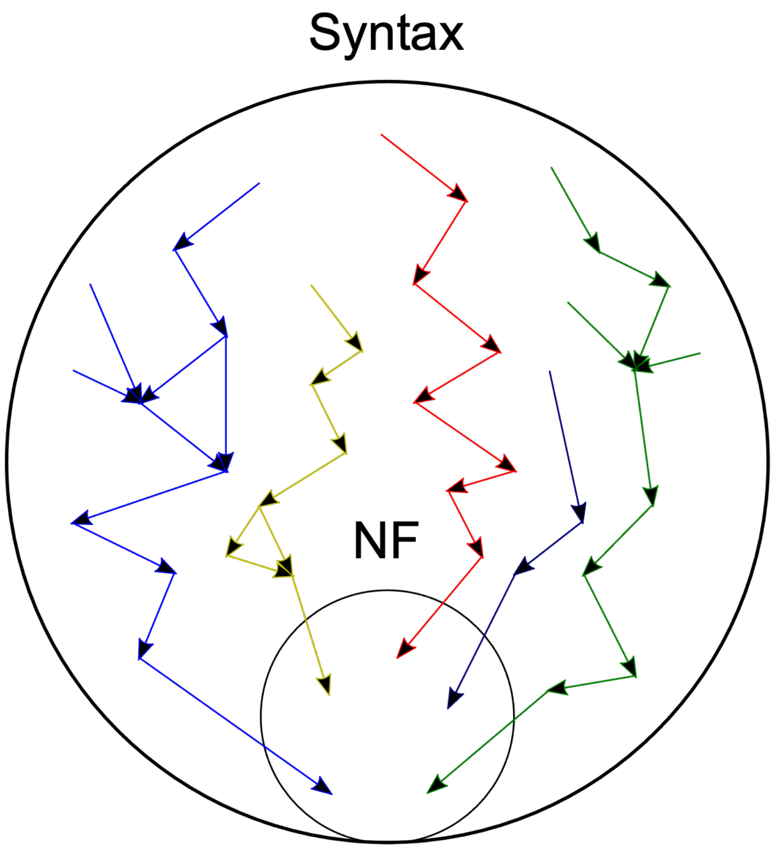
Main Idea

Normalization by Evaluation (NbE) provides a principled account of specializing expression *syntax* (deep embedding) by leveraging their *semantics* (shallow embedding)

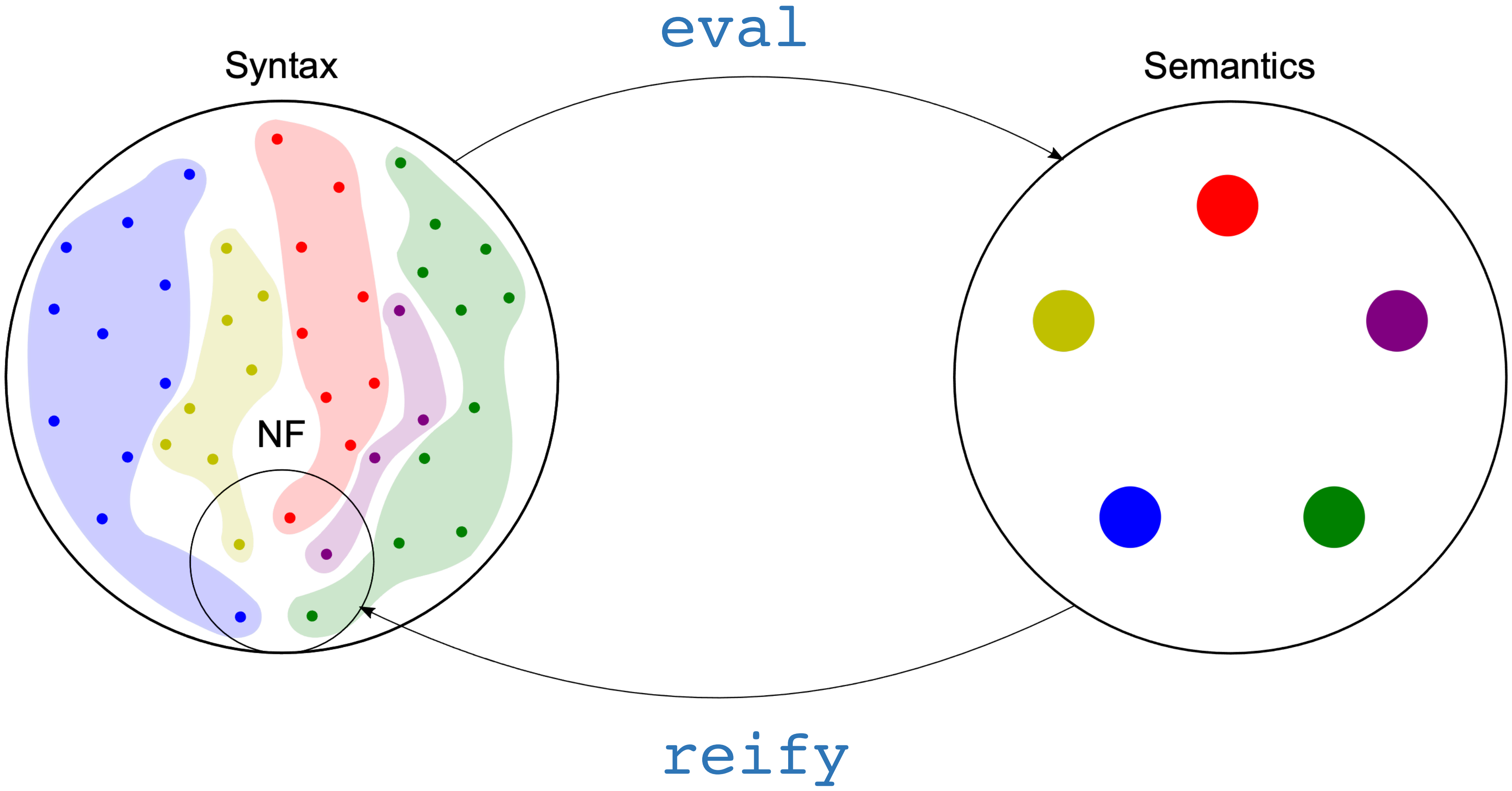
Main Idea

Normalization by Evaluation (NbE) provides a principled account of specializing expression *syntax* (deep embedding) by leveraging their *semantics* (shallow embedding)

Normalization by Rewriting



Normalization by Evaluation



The NbE Toolkit

```
class Rf a where
  type Sem a :: *
  reify   :: Sem a -> Nf a
  reflect :: Ne a -> Sem a
```

Semantic domain for a type

Specialized expressions

Expressions “stuck” at unknowns

```
eval :: Rf a => Exp a -> Sem a

norm :: Rf a => Exp a -> Nf a
norm = reify ∘ eval
```

NbE for Integers: Equations

```
Add (Lift x) (Lift y) ≈ Lift (x + y)
Mul (Lift x) (Lift y) ≈ Lift (x * y)
```

NbE for Integers: Neutrals

```
data Ne a where
  NVar  :: String -> Ne a
  NAdd₁ :: Ne Int -> Int -> Ne Int
  NAdd₂ :: Int -> Ne Int -> Ne Int
  NAdd  :: Ne Int -> Ne Int -> Ne Int
     -- similarly NMul₁, NMul₂ and NMul
```

NbE for Integers: Normal forms

```
data Nf a where
  NUp    :: Ne Int -> Nf Int
  NLift :: Int      -> Nf Int
```

NbE for Integers: Semantic Domain

```
instance Rf Int where
  type Sem Int = Either (Ne Int) Int
  reify (Left n)  = NUp n
  reify (Right x) = NLift x
  reflect n       = Left n
```

NbE for Integers: Evaluation

```
eval (Add e e’) = add (eval e) (eval e’)

add :: Sem Int -> Sem Int -> Sem Int
add (Right x) (Right y) = Right (x + y)
add (Left n)  (Right y) = reflect (NAdd₁ n y)
add (Right x) (Left n)  = reflect (NAdd₂ x n)
add …         …         = …

-- similarly evaluate Mul using mul
```

NbE for Arrays: Equations

```
arr :: Exp (Arr a) ≈ NewArr (LenArr arr) (IxArr arr)
LenArr (NewArr n ixf) ≈ n
IxArr (NewArr n ixf)  ≈ ixf
```

NbE for Arrays: Semantic Domain

```
data SArr a where
  SArr :: Sem Int -> (Exp Int -> a) -> SArr a

newArr :: Sem Int -> (Exp Int -> Sem a) -> SArr (Sem a)
newArr = SArr

lenArr :: SArr (Sem a) -> Sem Int
lenArr (SArr n _) = n

ixArr  :: SArr (Sem a) -> Exp Int -> Sem a
ixArr (SArr _ ixf) = ixf
```

NbE for Arrays: Semantic Domain

```
instance Rf a => Rf (Arr a) where
  type Sem (Arr a) = SArr (Sem a)
  reify (SArr n ixf) = NewArr … …
  reflect ne         = SArr … …

eval (NewArr n ixf) = newArr (eval n) (eval ∘ ixf)
-- similarly using lenArr and ixArr
```

Extension To Other Types

```
-- Pure Types
instance (Rf a, Rf b) => Rf (a -> b) where …
instance (Rf a, Rf b) => Rf (a,b) where …
instance (Rf a, Rf b) => Rf (Either a b) where …

-- Computational Effects
instance Rf a => Rf (Err a) where …
instance (Rf s, Rf a) => Rf (State s a) where
```

Extension To Other Types

```
-- Pure Types
instance (Rf a, Rf b) => Rf (a -> b) where …
instance (Rf a, Rf b) => Rf (a,b) where …
instance (Rf a, Rf b) => Rf (Either a b) where …

-- Computational Effects
instance Rf a => Rf (Err a) where …
instance (Rf s, Rf a) => Rf (State s a) where …
```

Map-map Fusion in a Branching Program

```
prgBr :: Exp (Either Int Int -> Arr Int -> Arr Int)
prgBr = lam2D $ \ scr arr -> mapArrD (addD 1) $
  caseD scr
    (lamD $ \x -> mapArrD (lamD (addD x)) arr)
    (lamD $ \_ -> arr)
```

```
> norm prgBr
\scr. \arr. NewArr (LenArr arr) (\i. Case scr of
  inl -> (\x. (IxArr arr i) + x + 1)
  inr -> (\_. (IxArr arr i) + 1))
```

Put-Put and Put-Get Fusion

```
prgSt :: Exp (Arr Int -> State (Arr Int) Int)
prgSt = lamD $ \ arr ->
  putD (mapArrD (lamD (addD 2)) arr)
  >>D putD (mapArrD (lamD (addD 1)) arr)
  >>D getD
  >>=D (lamD $ \ arr' -> returnD (ixArrD arr' 0))

> norm prgSt
\arr. get
  >>= (\s. put (NewArr (LenArr arr) (\i. (IxArr arr i) + 1))
  >>= (\_. return (IxArr arr 0) + 1))
```

What is so “practical”?

Practical NbE: Taming η-expansion

```
> norm (Var “arr” :: Exp (Arr Int))
NewArr (LenArr arr) (\i. IxArr arr i)
```

```
instance Rf a => Rf (Arr a) where
  type Sem (Arr a) = (SArr (Sem a), Nf (Arr a))
  reify    (_,nf)   = nf
  …
```

There’s more!

- Controlling duplication using sharing and a “save” combinator
- Optimizing case expressions
- Richer arithmetic equations:
  - 0 + x = x
  - x * (y + z) = (x * y) + (x * z)
  - …, etc.
- Adding uninterpreted constants, e.g., “fix”

In a nutshell

NbE provides a principled alternative to techniques that combine deep and shallow embedding for building and specializing eDSLs, while retaining modularity and fine-grained control over the extent of specialization

github.com/nachivpn/nbe-edsl