Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Indexed fixed points

Andres Löh

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
Web pages: `http://www.cs.uu.nl/wiki/Center`

GP meeting, 21 November 2008

Fixed points

$$\begin{aligned}
\mathsf{Fix}_1 &: (* \to *) \to \\
&\quad * \\
\mathsf{Fix}_2 &: (* \to * \to *) \to \\
&\quad (* \to * \to *) \to \\
&\quad * \\
\mathsf{Fix}_3 &: (* \to * \to * \to *) \to \\
&\quad (* \to * \to * \to *) \to \\
&\quad (* \to * \to * \to *) \to \\
&\quad * \\
\mathsf{Fix}_n &: \mathbf{?}
\end{aligned}$$

How to generalize to arbitrary arities?

Fixed points

$$
\begin{aligned}
\mathsf{Fix}_1 &: (* \to *) \to \\
&\quad * \\
\mathsf{Fix}_2 &: (* \times * \to *) \times \\
&\quad (* \times * \to *) \to \\
&\quad * \\
\mathsf{Fix}_3 &: (* \times * \times * \to *) \times \\
&\quad (* \times * \times * \to *) \times \\
&\quad (* \times * \times * \to *) \to \\
&\quad * \\
\mathsf{Fix}_n &: \textbf{?}
\end{aligned}
$$

How to generalize to arbitrary arities?

① Uncurry

Fixed points

$$\mathsf{Fix}_1 : (*^1 \to *)^1 \to *$$

$$\mathsf{Fix}_2 : (*^2 \to *)^2 \to *$$

$$\mathsf{Fix}_3 : (*^3 \to *)^3 \to *$$

$$\mathsf{Fix}_n : \textbf{?}$$

How to generalize to arbitrary arities?

① Uncurry — ② Simplify

Fixed points

$$\text{Fix}_1 : (*^1 \rightarrow *)^1 \rightarrow *$$

$$\text{Fix}_2 : (*^2 \rightarrow *)^2 \rightarrow *$$

$$\text{Fix}_3 : (*^3 \rightarrow *)^3 \rightarrow *$$

$$\text{Fix}_n : (*^n \rightarrow *)^n \rightarrow *$$

How to generalize to arbitrary arities?

① Uncurry — ② Simplify

Fixed points

$$\mathsf{Fix}_1 : (\mathbf{1} \to ((\mathbf{1} \to *) \to *)) \to *$$

$$\mathsf{Fix}_2 : (\mathbf{2} \to ((\mathbf{2} \to *) \to *)) \to *$$

$$\mathsf{Fix}_3 : (\mathbf{3} \to ((\mathbf{3} \to *) \to *)) \to *$$

$$\mathsf{Fix}_n : (\mathbf{n} \to ((\mathbf{n} \to *) \to *)) \to *$$

How to generalize to arbitrary arities?

① Uncurry — ② Simplify — ③ Index

Agda vs. Haskell

$\mathsf{Fix}_n : (\mathbf{n} \to ((\mathbf{n} \to *) \to *)) \to *$

Agda:

$\mathsf{Fix} : (n : \mathbb{N}) \to (\mathsf{Fin}\ n \to ((\mathsf{Fin}\ n \to *) \to *)) \to *$

Haskell:

$\mathsf{Fix} :: (* \to ((* \to *) \to *)) \to *$

where $*$ is instantiated by a suitable index type.

Adding projection

Describes the complete family:

$$\mathsf{Fix}_n : (\mathbf{n} \rightarrow ((\mathbf{n} \rightarrow *) \rightarrow *)) \rightarrow *$$

Projecting out one type:

$$\mathsf{Fix}_n : (\mathbf{n} \rightarrow ((\mathbf{n} \rightarrow *) \rightarrow *)) \rightarrow \mathbf{n} \rightarrow *$$

Adding projection

Describes the complete family:

$$\mathsf{Fix}_{\mathsf{n}} \; : \; (\mathbf{n} \to ((\mathbf{n} \to *) \to *)) \to *$$

Projecting out one type:

$$\mathsf{Fix}_{\mathsf{n}} \; : \; (\mathbf{n} \to ((\mathbf{n} \to *) \to *)) \to \mathbf{n} \to *$$

Agda:

$$\mathsf{Fix} \; : \; (\mathsf{n} \; : \; \mathbb{N}) \to (\mathsf{Fin}\ \mathsf{n} \to ((\mathsf{Fin}\ \mathsf{n} \to *) \to *)) \to \mathsf{Fin}\ \mathsf{n} \to *$$

Haskell:

$$\mathsf{Fix} :: (* \to ((* \to *) \to *)) \to * \to *$$

The universe

```
module Base (n : ℕ) where
  Ix = Fin n
```

The universe

```
module Base (n : ℕ) where
  Ix = Fin n

  data Code : Set₁ where
     I   :                 Ix  → Code
     K   :                 Set → Code
    _+_  : Code → Code → Code
    _×_  : Code → Code → Code
    _▷_  :        Code → Ix → Code
```

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

The universe

```
module Base (n : ℕ) where
  Ix = Fin n

  data Code : Set₁ where
    I   :                 Ix   → Code
    K   :                 Set  → Code
    _+_ : Code → Code → Code
    _×_ : Code → Code → Code
    _▷_ :        Code → Ix   → Code

  ⟦_⟧ : Code → (Ix → Set) → Ix → Set
  ⟦ I x     ⟧ ⟨_⟩ _ = ⟨ x ⟩
  ⟦ K A     ⟧  _  _ = A
  ⟦ C₁ + C₂ ⟧ ⟨_⟩ y = ⟦ C₁ ⟧ ⟨_⟩ y ⊎ ⟦ C₂ ⟧ ⟨_⟩ y
  ⟦ C₁ × C₂ ⟧ ⟨_⟩ y = ⟦ C₁ ⟧ ⟨_⟩ y × ⟦ C₂ ⟧ ⟨_⟩ y
  ⟦ C ▷ x   ⟧ ⟨_⟩ y = x ≡ y          × ⟦ C  ⟧ ⟨_⟩ y
```

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Generic map

$$
\begin{array}{l}
[\![\_]\!] : \mathsf{Code} \to (\mathsf{Ix} \to \mathsf{Set}) \to \mathsf{Ix} \to \mathsf{Set} \\
[\![\ \mathbf{I}\ x\ ]\!]\ \langle\_\rangle\ \_ = \langle\ x\ \rangle \\
[\![\ \mathbf{K}\ A\ ]\!]\ \_\ \_ = A \\
[\![\ C_1 + C_2\ ]\!]\ \langle\_\rangle\ y = [\![\ C_1\ ]\!]\ \langle\_\rangle\ y \uplus [\![\ C_2\ ]\!]\ \langle\_\rangle\ y \\
[\![\ C_1 \times C_2\ ]\!]\ \langle\_\rangle\ y = [\![\ C_1\ ]\!]\ \langle\_\rangle\ y \times [\![\ C_2\ ]\!]\ \langle\_\rangle\ y \\
[\![\ C \rhd x\ ]\!]\ \langle\_\rangle\ y = x \equiv y \times [\![\ C\ ]\!]\ \langle\_\rangle\ y
\end{array}
$$

$$
\begin{array}{l}
\mathsf{map} : \{F\ G : \mathsf{Ix} \to \mathsf{Set}\}\ \{y : \mathsf{Ix}\} \to (C : \mathsf{Code}) \to \\
\quad (\{x : \mathsf{Ix}\} \to F\ x \to G\ x) \to [\![\ C\ ]\!]\ F\ y \to [\![\ C\ ]\!]\ G\ y \\
\mathsf{map}\ (\ \mathbf{I}\ x\ )\ f\ y = f\ y \\
\mathsf{map}\ (\ \mathbf{K}\ \_\ )\ f\ y = y \\
\mathsf{map}\ (C_1 + C_2)\ f\ (\mathsf{inj}_1\ y_1) = \mathsf{inj}_1\ (\mathsf{map}\ C_1\ f\ y_1) \\
\mathsf{map}\ (C_1 + C_2)\ f\ (\mathsf{inj}_2\ y_2) = \mathsf{inj}_2\ (\mathsf{map}\ C_2\ f\ y_2) \\
\mathsf{map}\ (C_1 \times C_2)\ f\ (y_1, y_2) = \mathsf{map}\ C_1\ f\ y_1, \mathsf{map}\ C_2\ f\ y_2 \\
\mathsf{map}\ (\ C \rhd x\ )\ f\ (\equiv\text{-refl}, y) = \equiv\text{-refl}, \mathsf{map}\ C\ f\ y
\end{array}
$$

Embedding-projection

Lets group everything required to allow generic programming for a family of types in a record:

```
record Fam : Set₁ where
  field
    FC   : Code
    ⟨_⟩  : Ix → Set
    from : {x : Ix} → ⟨ x ⟩ → ⟦ FC ⟧ ⟨_⟩ x
    to   : {x : Ix} → ⟦ FC ⟧ ⟨_⟩ x → ⟨ x ⟩
```

Embedding-projection

Lets group everything required to allow generic programming for a family of types in a record:

```
record Fam : Set₁ where
  field
    FC    : Code
    ⟨_⟩   : Ix → Set
    from  : {x : Ix} → ⟨ x ⟩ → ⟦ FC ⟧ ⟨_⟩ x
    to    : {x : Ix} → ⟦ FC ⟧ ⟨_⟩ x → ⟨ x ⟩
```

Boilerplate to write:

1. The Code for the family.
2. The interpretation function ⟨_⟩.
3. Conversion functions from and to.

ASTs

```
module AST where
  Var : Set
  Var = String
  mutual
    data Expr : Set where
      econst :               ℕ → Expr
      eadd   : Expr → Expr → Expr
      emul   : Expr → Expr → Expr
      evar   :             Var → Expr
      elet   : Decl → Expr → Expr
    data Decl : Set where
      _:=_   :  Var → Expr → Decl
      _•_    : Decl → Decl → Decl
```

Instantiating Fam — preliminaries

```
open Base 3
```

A view is not necessary, but nice to have:

```
expr = zero
decl = suc zero
var  = suc (suc zero)

data ViewAST : Ix → Set where
  vexpr : ViewAST expr
  vdecl : ViewAST decl
  vvar  : ViewAST var

viewAST : (n : Ix) → ViewAST n
viewAST zero                   = vexpr
viewAST (suc zero)             = vdecl
viewAST (suc (suc zero))       = vvar
viewAST (suc (suc (suc ())))
```

Instantiating Fam — codes

```
ExprC : Code
ExprC = K ℕ                    -- econst
      + I expr × I expr        -- eadd
      + I expr × I expr        -- emul
      + I var                  -- evar
      + I decl × I expr        -- elet
DeclC : Code
DeclC = I var × I expr         -- _:=_
      + I decl × I decl        -- _•_
DeclC : Code
DeclC = K String
ASTC : Code
ASTC = ExprC ▷ expr
     + DeclC ▷ decl
     + DeclC ▷ var
```

Instantiating Fam – interpretation

```
AST⟨_⟩ : Ix → Set
AST⟨ x ⟩ with viewAST x
AST⟨ ._ ⟩ | vexpr = Expr
AST⟨ ._ ⟩ | vdecl = Decl
AST⟨ ._ ⟩ | vvar  = Var
```

Instantiating Fam – conversions

```
fromExpr : Expr → ⟦ ExprC ⟧ AST⟨_⟩ expr
fromExpr (econst i    ) = inj₁                    i
fromExpr (eadd e₁ e₂) = inj₂ (inj₁                (e₁ , e₂))
fromExpr (emul e₁ e₂) = inj₂ (inj₂ (inj₁          (e₁ , e₂)))
fromExpr (evar  v     ) = inj₂ (inj₂ (inj₂ (inj₁ v         )))
fromExpr (elet  d  e  ) = inj₂ (inj₂ (inj₂ (inj₂ (d  , e  ))))

fromDecl : Decl → ⟦ DeclC ⟧ AST⟨_⟩ decl
fromDecl (v   := e  ) = inj₁ (v, e)
fromDecl (d₁  •  d₂) = inj₂ (d₁ , d₂)

fromVar : Var → ⟦ DeclC ⟧ AST⟨_⟩ var
fromVar v = v

fromAST : {x : Ix} → AST⟨ x ⟩ → ⟦ ASTC ⟧ AST⟨_⟩ x
fromAST {x} v with viewAST x
fromAST e | vexpr = inj₁        (≡-refl, fromExpr e)
fromAST d | vdecl = inj₂ (inj₁ (≡-refl, fromDecl d))
fromAST v | vvar  = inj₂ (inj₂ (≡-refl, fromVar  v))
```

Instantiating Fam – conversions

```
toExpr  : ⟦ ExprC ⟧ AST⟨_⟩ expr  →  Expr
toExpr (inj₁                   i        )     =  econst i
toExpr (inj₂ (inj₁             (e₁ , e₂)))    =  eadd e₁ e₂
toExpr (inj₂ (inj₂ (inj₁       (e₁ , e₂))))   =  emul e₁ e₂
toExpr (inj₂ (inj₂ (inj₂ (inj₁ v        ))))  =  evar v
toExpr (inj₂ (inj₂ (inj₂ (inj₂ (d  , e  ))))) =  elet d e

toDecl  : ⟦ DeclC ⟧ AST⟨_⟩ decl  →  Decl
toDecl (inj₁ (v  , e  ))  =  v   := e
toDecl (inj₂ (d₁ , d₂))   =  d₁  •  d₂

toVar  : ⟦ DeclC ⟧ AST⟨_⟩ var  →  Var
toVar v  =  v

toAST  : {x : Ix} → ⟦ ASTC ⟧ AST⟨_⟩ x  →  AST⟨ x ⟩
toAST (inj₁        (≡-refl , e))    =  toExpr e
toAST (inj₂ (inj₁  (≡-refl , d)))   =  toDecl d
toAST (inj₂ (inj₂  (≡-refl , v)))   =  toVar v
```

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Instantiating Fam – finalization

```
AST : Fam
AST = record
  {FC     = ASTC
  ; ⟨_⟩   = AST⟨_⟩
  ; from  = fromAST
  ; to    = toAST
  }
```

Generic fold – algebras

```
module Fold {n : ℕ} (F : Base.Fam n) where
  open Base n public
  open Fam F

  RawAlg : Code → (F G : Ix → Set) → Ix → Set
  RawAlg C F G y = ⟦ C ⟧ F y → G y

  Alg : Code → (F G : Ix → Set) → Ix → Set
  Alg (    I x    ) F G y = F x → G y
  Alg (    K A    ) F G y = A → G y
  Alg (C₁  +  C₂) F G y = Alg C₁ F G y × Alg C₂ F G y
  Alg (C₁  ×  C₂) F G y = Alg C₁ F (Alg C₂ F G) y
  Alg (  C  ▷  x  ) F G y = Alg C F G x

  RawAlgebra : (Ix → Set) → Set
  RawAlgebra F = (x : Ix) → RawAlg FC F F x

  Algebra      : (Ix → Set) → Set
  Algebra      F = (x : Ix) → Alg      FC F F x
```

Generic fold – relating algebras

```
RawAlg : Code → (F G : Ix → Set) → Ix → Set
RawAlg C F G y = ⟦ C ⟧ F y → G y

Alg : Code → (F G : Ix → Set) → Ix → Set
Alg (    I x    ) F G y = F x → G y
Alg (    K A    ) F G y = A → G y
Alg (C₁  +  C₂) F G y = Alg C₁ F G y × Alg C₂ F G y
Alg (C₁  ×  C₂) F G y = Alg C₁ F (Alg C₂ F G) y
Alg ( C  ▷  x ) F G y = Alg C F G x
```

```
apply : (C : Code) {F G : Ix → Set} {y : Ix} →
        Alg C F G y → RawAlg C F G y
apply (    I x    ) a         y          = a y
apply (    K _    ) a         y          = a y
apply (C₁  +  C₂) (a₁, a₂) (inj₁ y₁)  = apply C₁ a₁ y₁
apply (C₁  +  C₂) (a₁, a₂) (inj₂ y₂)  = apply C₂ a₂ y₂
apply (C₁  ×  C₂) a         (y₁, y₂)   = apply C₂ (apply C₁ a y₁) y₂
apply ( C  ▷  x ) a         (≡-refl, y) = apply C a y
```

Generic fold – the function

This unfortunately doesn't termination-check:

```
foldRaw : {y : Ix} {F : Ix → Set} →
            RawAlgebra F → ⟨ y ⟩ → F y
foldRaw alg x = alg _ (map FC (foldRaw alg) (from x))
```

Generic fold – the function

This unfortunately doesn't termination-check:

```
foldRaw : {y : Ix} {F : Ix → Set} →
            RawAlgebra F → ⟨ y ⟩ → F y
foldRaw alg x = alg _ (map FC (foldRaw alg) (from x))
```

```
apply : (C : Code) {F G : Ix → Set} {y : Ix} →
        Alg C F G y → RawAlg C F G y
```

```
fold    : {y : Ix} {F : Ix → Set} →
            Algebra    F → ⟨ y ⟩ → F y
fold alg = foldRaw (λ x → apply FC (alg x))
```

Generic fold – application

Instantiating the Fold module to AST:

```
module FoldExample where
  open AST
  open Fold AST
  open Fam AST
```

Generic fold – application

```
Value : Ix → Set
Value x with viewAST x
Value ._ | vexpr = Env → ℕ
Value ._ | vdecl = Env → Env
Value ._ | vvar  = Var

evalAlgebra : Algebra Value
evalAlgebra _ =
  ((λ i     env → i                      ) ,   -- econst
   (λ r₁ r₂ env → r₁ env + r₂ env        ) ,   -- eadd
   (λ r₁ r₂ env → r₁ env ∗ r₂ env        ) ,   -- emul
   (λ v     env → env v                  ) ,   -- evar
   (λ f  r  env → r (f env)              )),   -- elet
  ((λ v  r  env → insert v (r env) env) ,      -- _:=_
   (λ f₁ f₂ env → f₂ (f₁ env)            )),   -- _•_
  ((λ v         → v                      ))
```

Generic fold – call

```
eval : Expr → Value expr
eval = fold {expr} {Value} evalAlgebra

example : Expr
example = elet ("x" := emul (econst 6) (econst 9) •
                "y" := eadd (evar "x") (econst 2))
               (eadd (evar "y") (evar "x"))

testFold : eval example empty ≡ 110
testFold = ≡-refl
```

One-hole contexts

```
module Zipper {n : ℕ} (F : Base.Fam n) where
  open Base n public
  open Fam F
  Ctx : Code → Ix → Ix → Set
  Ctx (   I x    ) y z = x ≡ y
  Ctx (   K _    ) y z = ⊥
  Ctx (C₁ ⊕ C₂) y z = Ctx C₁ y z  ⊎ Ctx C₂ y z
  Ctx (C₁ × C₂) y z = Ctx C₁ y z  × ⟦ C₂ ⟧ ⟨_⟩ z
                    ⊎ ⟦ C₁ ⟧ ⟨_⟩ z × Ctx C₂ y z
  Ctx ( C ▷ x ) y z = x ≡ z × Ctx C y z
```

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

From contexts to locations

A path of contexts is the reflexive transitive closure of the Ctx relation:

$$\mathsf{Ctx} : \mathsf{Code} \to \mathsf{Ix} \to \mathsf{Ix} \to \mathsf{Set}$$

$$\mathsf{Rel} : \mathsf{Set} \to \mathsf{Set}_1$$
$$\mathsf{Rel}\ A = A \to A \to \mathsf{Set}$$

From contexts to locations

A path of contexts is the reflexive transitive closure of the Ctx relation:

$$\mathsf{Ctx} \;:\; \mathsf{Code} \;\rightarrow\; \mathsf{Rel}\ \mathsf{Ix}$$

$$\mathsf{Rel} \;:\; \mathsf{Set} \;\rightarrow\; \mathsf{Set}_1$$
$$\mathsf{Rel}\ \mathsf{A} \;=\; \mathsf{A} \;\rightarrow\; \mathsf{A} \;\rightarrow\; \mathsf{Set}$$

From contexts to locations

A path of contexts is the reflexive transitive closure of the Ctx relation:

$$\mathsf{Ctx} : \mathsf{Code} \to \mathsf{Rel}\ \mathsf{Ix}$$

$$\mathsf{Rel} : \mathsf{Set} \to \mathsf{Set}_1$$
$$\mathsf{Rel}\ A = A \to A \to \mathsf{Set}$$

$$\mathsf{Ctxs} : \mathsf{Rel}\ \mathsf{Ix}$$
$$\mathsf{Ctxs} = \mathsf{Star}\ (\mathsf{Ctx}\ \mathsf{FC})$$

From contexts to locations

A path of contexts is the reflexive transitive closure of the Ctx relation:

```
Ctx : Code → Rel Ix

Rel : Set → Set₁
Rel A = A → A → Set
```

```
Ctxs : Rel Ix
Ctxs = Star (Ctx FC)
```

A location is a pair of a value and a path:

```
data Loc : Ix → Set where
  _,_ : {x y : Ix} → ⟨ x ⟩ → Ctxs x y → Loc y
```

Filling a hole

```
Ctx : Code → Ix → Ix → Set
Ctx (   I x   ) y z = x ≡ y
Ctx (   K _   ) y z = ⊥
Ctx (C₁ + C₂) y z = Ctx C₁ y z ⊎ Ctx C₂ y z
Ctx (C₁ × C₂) y z = Ctx C₁ y z × ⟦ C₂ ⟧ ⟨_⟩ z
                  ⊎ ⟦ C₁ ⟧ ⟨_⟩ z × Ctx C₂ y z
Ctx ( C ▷ x ) y z = x ≡ z × Ctx C y z
```

```
fill : (C : Code) {x y : Ix} →
       Ctx C x y → ⟨ x ⟩ → ⟦ C ⟧ ⟨_⟩ y
fill (   I _   ) ≡-refl             y  = y
fill (   K _   ) ()                 _
fill (C₁ + C₂) (inj₁ cy₁)          y₁ = inj₁ (fill C₁ cy₁ y₁)
fill (C₁ + C₂) (inj₂ cy₂)          y₂ = inj₂ (fill C₂ cy₂ y₂)
fill (C₁ × C₂) (inj₁ (cy₁ , y₂))   y₁ = fill C₁ cy₁ y₁ , y₂
fill (C₁ × C₂) (inj₂ (y₁ , cy₂))   y₂ = y₁ , fill C₂ cy₂ y₂
fill ( C ▷ _ ) (≡-refl , cy)       y  = ≡-refl , fill C cy y
```

Navigating up

```
open RawMonadPlus MaybeMonadPlus

Nav : Set
Nav = {x : Ix} → Loc x → Maybe (Loc x)

up : Nav
up (x, [])       = ∅
up (x, c :: cs)  = return (to (fill FC c x), cs)
```

Navigating down

```
first : {A : Set} (C : Code) {y : Ix} →
        ({x : Ix} → ⟨ x ⟩ → Ctx C x y → A) →
        ⟦ C ⟧ ⟨_⟩ y → Maybe A
first (   I _    ) k y          = return (k y ≡-refl)
first (   K _    ) k _          = ∅
first (C₁ + C₂) k (inj₁ y₁)     = first C₁ (\z cy₁ → k z (inj₁ cy₁)        ) y₁
first (C₁ + C₂) k (inj₂ y₂)     = first C₂ (\z cy₂ → k z (inj₂ cy₂)        ) y₂
first (C₁ × C₂) k (y₁ , y₂)     = first C₁ (\z cy₁ → k z (inj₁ (cy₁ , y₂))) y₁
                                | first C₂ (\z cy₂ → k z (inj₂ (y₁ , cy₂))) y₂
first ( C ▷ _ ) k (≡-refl , y)  = first C  (\z cy  → k z (≡-refl , cy)     ) y

down : Nav
down (x , cs) = first FC (\z c → z , (c :: cs)) (from x)
```

Navigating right

```
next : {A : Set} (C : Code) {y : Ix} →
       ({x : Ix} → ⟨ x ⟩ → Ctx C x y → A) →
       {x : Ix} → Ctx C x y → ⟨ x ⟩ → Maybe A
next (   I _   ) k ≡-refl          y   = ∅
next (   K _   ) k ()              _
next (C₁ + C₂) k (inj₁ cy₁)        y₁  = next C₁ (\z cy₁′ → k z (inj₁ cy₁′)                        ) cy₁ y₁
next (C₁ + C₂) k (inj₂ cy₂)        y₂  = next C₂ (\z cy₂′ → k z (inj₂ cy₂′)                        ) cy₂ y₂
next (C₁ × C₂) k (inj₁ (cy₁, y₂)) y₁  = next C₁ (\z cy₁′ → k z (inj₁ (cy₁′               , y₂   ))) cy₁ y₁
                                       | first C₂ (\z cy₁′ → k z (inj₂ (fill C₁ cy₁ y₁, cy₁′)))       y₂
next (C₁ × C₂) k (inj₂ (y₁, cy₂)) y₂  = next C₂ (\z cy₂′ → k z (inj₂ (y₁                , cy₂′))) cy₂ y₂
next (  C ▷ _ ) k (≡-refl, cy)     y   = next C  (\z cy′  → k z (≡-refl, cy′)                       ) cy  y

right : Nav
right (x, []        ) = ∅
right (x, (c :: cs)) = next FC (\z c′ → z, (c′ :: cs)) c x
```

Completing the navigation functions

The functions prev and left are similar to next and right.

$$
\begin{array}{l}
\text{enter} : \{x : Ix\} \to \langle x \rangle \to \text{Loc}\ x \\
\text{enter}\ x = x, [\,] \\[4pt]
\text{leave} : \{x : Ix\} \to \text{Loc}\ x \to \langle x \rangle \\
\text{leave}\ (x, [\,]) = x \\
\text{leave}\ (x, (c :: cs)) = \text{leave}\ (\text{to}\ (\text{fill}\ FC\ c\ x), cs) \\[4pt]
\text{update} : ((x : Ix) \to \langle x \rangle \to \langle x \rangle) \to \text{Nav} \\
\text{update}\ f\ (x, cs) = \text{return}\ (f\ \_\ x, cs) \\[4pt]
\text{on} : \{A : \text{Set}\} \to ((x : Ix) \to \langle x \rangle \to A) \to \\
\qquad \{x : Ix\} \to \text{Loc}\ x \to A \\
\text{on}\ f\ (x, cs) = f\ \_\ x
\end{array}
$$

Using the Zipper – preparations

```
module ZipperExample where
  open AST
  open Zipper AST
  open Fam AST
  open FoldExample
  open RawMonadPlus MaybeMonadPlus
```

Using the Zipper – test

```
source  :  Expr
source  =  elet ("x" := emul (econst 6) (econst 9))
                (eadd (evar "x") (evar "y"))

callZipper  :  Maybe Expr
callZipper  =
   return (enter {expr} source)  >>=
   down  >>=  down  >>=  right  >>=  update simp  >>=
   return ∘ leave
      where simp  :  (n  :  Ix)  →  ⟨ n ⟩  →  ⟨ n ⟩
            simp n x with viewAST n
            simp ._ e  |  vexpr  =  econst (eval e empty)
            simp ._ d  |  vdecl  =  d
            simp ._ v  |  vvar   =  v

target  =  elet ("x" := econst 54) (eadd (evar "x") (evar "y"))
testZipper  :  callZipper ≡ just target
testZipper  =  ≡-refl
```