Verified tail-recursive folds through dissection

Thesis defense

Carlos Tomé Cortiñas

July 19, 2018

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

1. Introduction

1.1 Motivation

Expression language

data Expr : Set where
  Val : $\mathbb{N}$ → Expr
  Add : Expr → Expr → Expr

$expr_1$: Expr
$expr_1$ = Add (Val 1)
          (Add (Val 2)
               (Val 2))

Expression language

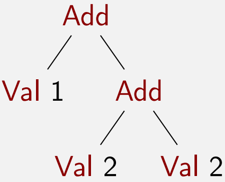
```
data Expr : Set where
  Val : ℕ          → Expr
  Add : Expr → Expr → Expr
```

```
expr₁ : Expr
expr₁ = Add (Val 1)
            (Add (Val 2)
                 (Val 2))
```

Expression language


```
data Expr : Set where
  Val : ℕ         → Expr
  Add : Expr → Expr → Expr
```

```
expr₁ : Expr
expr₁ = Add (Val 1)
            (Add (Val 2)
                 (Val 2))
```



Evaluating Expressions

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \to \mathbb{N} \\
&\mathsf{eval}\ (\mathsf{Val}\ n) = n \\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&prop_1 : \mathsf{eval}\ expr_1 \equiv 5 \\
&prop_1 = \mathsf{refl}
\end{aligned}
$$

Is there a **problem** with eval?

Evaluating Expressions

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \to \mathbb{N} \\
&\mathsf{eval}\ (\mathsf{Val}\ n) = n \\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&prop_1 : \mathsf{eval}\ expr_1 \equiv 5 \\
&prop_1 = \mathsf{refl}
\end{aligned}
$$

Is there a **problem** with eval?

Evaluating Expressions

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \to \mathbb{N} \\
&\mathsf{eval}\ (\mathsf{Val}\ \mathsf{n}) = \mathsf{n} \\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&prop_1 : \mathsf{eval}\ expr_1 \equiv 5 \\
&prop_1 = \mathsf{refl}
\end{aligned}
$$

Is there a **problem** with eval?

Evaluating Expressions (2)

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \rightarrow \mathbb{N}\\
&\mathsf{eval}\ (\mathsf{Val}\ n) = n\\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$\mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 1)\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2)))$

$\rightsquigarrow\ \mathsf{eval}\ (\mathsf{Val}\ 1) + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))$

$\rightsquigarrow\ 1 + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))$

$\rightsquigarrow\ 1 + (\mathsf{eval}\ (\mathsf{Val}\ 2) + \mathsf{eval}\ (\mathsf{Val}\ 2))$

$\rightsquigarrow\ 1 + (2 + \mathsf{eval}\ (\mathsf{Val}\ 2))\ \ldots$

Evaluating Expressions (2)

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \to \mathbb{N}\\
&\mathsf{eval}\ (\mathsf{Val}\ n) = n\\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 1)\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2)))\\
&\quad\rightsquigarrow \mathsf{eval}\ (\mathsf{Val}\ 1) + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))\\
&\quad\rightsquigarrow 1 + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))\\
&\quad\rightsquigarrow 1 + (\mathsf{eval}\ (\mathsf{Val}\ 2) + \mathsf{eval}\ (\mathsf{Val}\ 2))\\
&\quad\rightsquigarrow 1 + (2 + \mathsf{eval}\ (\mathsf{Val}\ 2))\ \ldots
\end{aligned}
$$

Evaluating Expressions (2)

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \to \mathbb{N} \\
&\mathsf{eval}\ (\mathsf{Val}\ n) = n \\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 1)\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))) \\
&\leadsto \mathsf{eval}\ (\mathsf{Val}\ 1) + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2)) \\
&\leadsto 1 + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2)) \\
&\leadsto 1 + (\mathsf{eval}\ (\mathsf{Val}\ 2) + \mathsf{eval}\ (\mathsf{Val}\ 2)) \\
&\leadsto 1 + (2 + \mathsf{eval}\ (\mathsf{Val}\ 2))\ \ldots
\end{aligned}
$$

Evaluating Expressions (2)

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \to \mathbb{N} \\
&\mathsf{eval}\ (\mathsf{Val}\ n) = n \\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 1)\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))) \\
&\quad \leadsto \mathsf{eval}\ (\mathsf{Val}\ 1) + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2)) \\
&\quad \leadsto 1 + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2)) \\
&\quad \leadsto 1 + (\mathsf{eval}\ (\mathsf{Val}\ 2) + \mathsf{eval}\ (\mathsf{Val}\ 2)) \\
&\quad \leadsto 1 + (2 + \mathsf{eval}\ (\mathsf{Val}\ 2))\ \ldots
\end{aligned}
$$

Evaluating Expressions (2)

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr} \to \mathbb{N}\\
&\mathsf{eval}\ (\mathsf{Val}\ n) = n\\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 1)\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2)))\\
&\quad\leadsto \mathsf{eval}\ (\mathsf{Val}\ 1) + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))\\
&\quad\leadsto 1 + \mathsf{eval}\ (\mathsf{Add}\ (\mathsf{Val}\ 2)\ (\mathsf{Val}\ 2))\\
&\quad\leadsto 1 + (\mathsf{eval}\ (\mathsf{Val}\ 2) + \mathsf{eval}\ (\mathsf{Val}\ 2))\\
&\quad\leadsto 1 + (2 + \mathsf{eval}\ (\mathsf{Val}\ 2))\ \ldots
\end{aligned}
$$

Evaluating Expressions (3)

Add (Val 1) (Add (Val 2) (Add (Val 3) (Add (Val 4) ...

1 + (2 + (3 + (4 + ...

- The execution *stack* grows linearly with the size of the Expr
- Stack Overflow!

A **well-typed** program *went wrong*[1]

[1] Robin Milner does not approve

Evaluating Expressions (3)

Add (Val 1) (Add (Val 2) (Add (Val 3) (Add (Val 4) ...

1 + (2 + (3 + (4 + ...

- The execution *stack* grows linearly with the size of the Expr
- Stack Overflow!

A **well-typed** program *went wrong*[1]

[1] Robin Milner does not approve

Evaluating Expressions (3)

Add (Val 1) (Add (Val 2) (Add (Val 3) (Add (Val 4) ...

1 + (2 + (3 + (4 + ...

- The execution *stack* grows linearly with the size of the Expr
- Stack Overflow!

A **well-typed** program *went wrong*[1]

[1] Robin Milner does not approve

Evaluating Expressions (3)

Add (Val 1) (Add (Val 2) (Add (Val 3) (Add (Val 4) ...

1 + (2 + (3 + (4 + ...

- The execution *stack* grows linearly with the size of the Expr
- Stack Overflow!

A **well-typed** program *went wrong*[1]

[1] Robin Milner does not approve

Evaluating Expressions (3)

Add (Val 1) (Add (Val 2) (Add (Val 3) (Add (Val 4) ...

1 + (2 + (3 + (4 + ...

- The execution *stack* grows linearly with the size of the Expr
- Stack Overflow!

A **well-typed** program *went wrong*[1]

[1] Robin Milner does not approve

A solution

- Make the *stack* explicit
- Write a *tail-recursive* function that recurses over it

```
data Stack : Set where
  Top   : Stack
  Left  : Expr → Stack → Stack
  Right : ℕ    → Stack → Stack
```

A solution

- Make the *stack* explicit
- Write a *tail-recursive* function that recurses over it

```
data Stack : Set where
  Top   : Stack
  Left  : Expr → Stack → Stack
  Right : ℕ    → Stack → Stack
```

A solution (2)

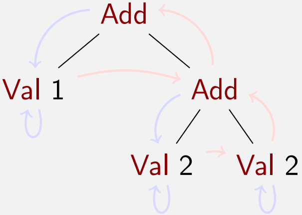

```
mutual
  load : Expr → Stack → ℕ
  load (Val n)        stk = unload n stk
  load (Add e₁ e₂) stk = load e₁ (Left e₂ stk)

  unload : ℕ → Stack → ℕ
  unload v Top              = v
  unload v (Right v' stk) = unload (v' + v) stk
  unload v (Left e stk)    = load e (Right v stk)

tail-rec-eval : Expr → ℕ
tail-rec-eval e = load e Top
```

A solution (2)

$$
\begin{array}{l}
\mathsf{mutual} \\
\quad \mathsf{load} : \mathsf{Expr} \to \mathsf{Stack} \to \mathbb{N} \\
\quad \mathsf{load}\ (\mathsf{Val}\ n)\ \mathit{stk} = \mathsf{unload}\ n\ \mathit{stk} \\
\quad \mathsf{load}\ (\mathsf{Add}\ e_1\ e_2)\ \mathit{stk} = \mathsf{load}\ e_1\ (\mathsf{Left}\ e_2\ \mathit{stk}) \\
\quad \mathsf{unload} : \mathbb{N} \to \mathsf{Stack} \to \mathbb{N} \\
\quad \mathsf{unload}\ v\ \mathsf{Top} = v \\
\quad \mathsf{unload}\ v\ (\mathsf{Right}\ v'\ \mathit{stk}) = \mathsf{unload}\ (v' + v)\ \mathit{stk} \\
\quad \mathsf{unload}\ v\ (\mathsf{Left}\ e\ \mathit{stk}) = \mathsf{load}\ e\ (\mathsf{Right}\ v\ \mathit{stk}) \\
\\
\mathsf{tail\text{-}rec\text{-}eval} : \mathsf{Expr} \to \mathbb{N} \\
\mathsf{tail\text{-}rec\text{-}eval}\ e = \mathsf{load}\ e\ \mathsf{Top}
\end{array}
$$

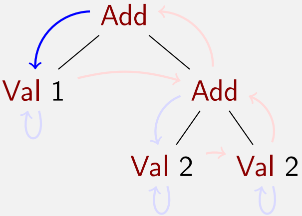
load and unload

load (Add (Val 1) (Add (Val 2) (Val 2))) Top



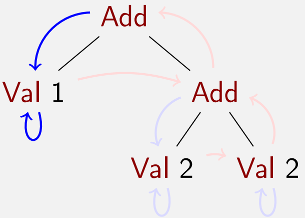
load and unload

load (Val 1) (Left (Add (Val 2) (Val 2)) Top)



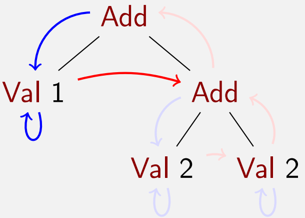
load and unload

unload 1 (Left (Add (Val 2) (Val 2)) Top)



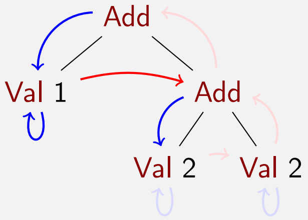
load and unload

load (Add (Val 2) (Val 2)) (Right 1 Top)



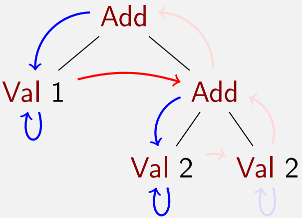
load and unload

load (Val 2) (Left (Val 2) (Right 1 Top))



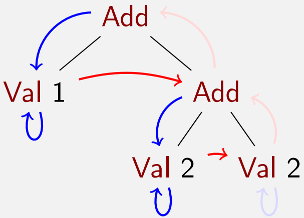
load and unload

unload 2 (Left (Val 2) (Right 1 Top))



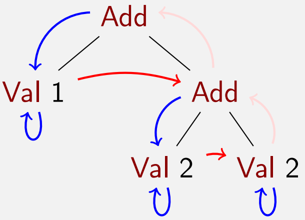
load and unload

load (Val 2) (Right 2 (Right 1 Top))



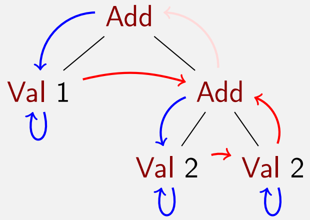
load and unload

unload 2 (Right 2 (Right 1 Top))



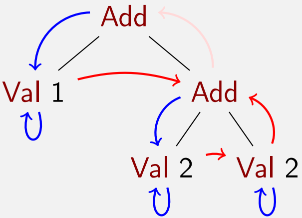
load and unload

unload (2 + 2) (Right 1 Top)



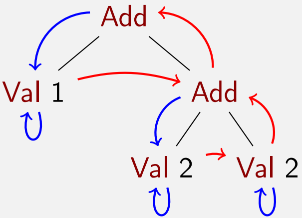
load and unload

unload 4 (Right 1 Top)



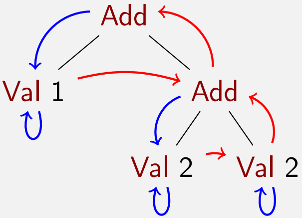
load and unload

unload (1 + 4) Top



load and unload

unload 5 Top



Problem solved?

Have we **actually** solved the problem?

- It seems so, however ...
- How do we know that

$$\forall\ (e\ :\ \mathsf{Expr})\ \rightarrow\ \mathsf{tail\text{-}rec\text{-}eval}\ e \equiv \mathsf{eval}\ e\ ?$$

- We **don't** know, we **don't** have a *mathematical proof*
- Let's *produce* it!

Problem solved?

Have we **actually** solved the problem?

- It seems so, however ...
- How do we know that

$$\forall\ (e : \mathsf{Expr}) \rightarrow \mathsf{tail\text{-}rec\text{-}eval}\ e \equiv \mathsf{eval}\ e\ ?$$

- We **don't** know, we **don't** have a *mathematical proof*
- Let's *produce* it!

Problem solved?

Have we **actually** solved the problem?

- It seems so, however ...
- How do we know that

$$\forall\ (e : \mathsf{Expr}) \rightarrow \mathsf{tail\text{-}rec\text{-}eval}\ e \equiv \mathsf{eval}\ e\ ?$$

- We **don't** know, we **don't** have a *mathematical proof*
- Let's *produce* it!

Problem solved?

Have we **actually** solved the problem?

- It seems so, however ...
- How do we know that

$$\forall\ (e\ :\ \mathsf{Expr})\ \rightarrow\ \mathsf{tail\text{-}rec\text{-}eval}\ e \equiv \mathsf{eval}\ e\ ?$$

- We **don't** know, we **don't** have a *mathematical proof*
- Let's *produce* it!

Problem solved?

Have we **actually** solved the problem?

- It seems so, however ...
- How do we know that

$$\forall\ (e\ :\ \mathsf{Expr})\ \rightarrow\ \mathsf{tail\text{-}rec\text{-}eval}\ e \equiv \mathsf{eval}\ e\ ?$$

- We **don't** know, we **don't** have a *mathematical proof*
- Let's *produce* it!

Problem solved? (2)

- Is **not** that easy; *tail-recursion* has come at a **price**

```
mutual
  load : Expr → Stack → ℕ
  load (Val n)       stk = unload n stk
  load (Add e₁ e₂) stk = load e₁ (Left e₂ stk)

  unload : ℕ → Stack → ℕ
  unload v Top            = v
  unload v (Right v′ stk) = unload (v′ + v) stk
  unload v (Left e stk)   = load e (Right v stk)
```

Problem solved? (2)

- Is **not** that easy; *tail-recursion* has come at a **price**

mutual

$$
\begin{array}{lll}
\boxed{\mathsf{load}} & : \mathsf{Expr} \to \mathsf{Stack} \to \mathbb{N} & \\
\mathsf{load}\ (\mathsf{Val}\ n) & stk = & \boxed{\mathsf{unload}}\ n\ stk \\
\mathsf{load}\ (\mathsf{Add}\ e_1\ e_2) & stk = & \boxed{\mathsf{load}}\ e_1\ (\mathsf{Left}\ e_2\ stk) \\
\boxed{\mathsf{unload}} & : \mathbb{N} \to \mathsf{Stack} \to \mathbb{N} & \\
\mathsf{unload}\ v\ \mathsf{Top} & = & v \\
\mathsf{unload}\ v\ (\mathsf{Right}\ v'\ stk) & = & \boxed{\mathsf{unload}}\ (v' + v)\ stk \\
\mathsf{unload}\ v\ (\mathsf{Left}\ e\ stk) & = & \boxed{\mathsf{load}}\ e\ (\mathsf{Right}\ v\ stk)
\end{array}
$$

1.2 Contributions of this Master Thesis

Contributions

- We construct a provably **terminating** tail-recursive function *similar* to tail-rec-eval
- We prove it **correct** with respect to eval
- We **generalize** our results to any fold over any (simple) algebraic datatype using McBride's notion of *dissection*

We have formalized everything in **Agda**

Contributions

- We construct a provably **terminating** tail-recursive function *similar* to tail-rec-eval
- We prove it **correct** with respect to eval
- We **generalize** our results to any fold over any (simple) algebraic datatype using McBride's notion of *dissection*

We have formalized everything in **Agda**

Contributions

- We construct a provably **terminating** tail-recursive function *similar* to tail-rec-eval
- We prove it **correct** with respect to eval
- We **generalize** our results to any fold over any (simple) algebraic datatype using McBride's notion of *dissection*

We have formalized everything in **Agda**

Contributions

- We construct a provably **terminating** tail-recursive function *similar* to tail-rec-eval
- We prove it **correct** with respect to eval
- We **generalize** our results to any fold over any (simple) algebraic datatype using McBride's notion of *dissection*

We have formalized everything in **Agda**

Contributions

- We construct a provably **terminating** tail-recursive function *similar* to tail-rec-eval
- We prove it **correct** with respect to eval
- We **generalize** our results to any fold over any (simple) algebraic datatype using McBride's notion of *dissection*

We have formalized everything in **Agda**

Outline

Introduction

A tail-recursive evaluator for Expr

A generic tail-recursive evaluator

Discussion

Conclusions

3. A tail-recursive evaluator for Expr

Problematic call


```
mutual
  load : Expr → Stack → ℕ
  load (Val n)        stk = unload n stk
  load (Add e₁ e₂)    stk = load e₁ (Left e₂ stk)

  unload : ℕ → Stack → ℕ
  unload v Top             = v
  unload v (Right v' stk)  = unload (v' + v) stk
  unload v (Left e stk)    = load e (Right v stk)
```

Terminating load and unload

- We rewrite load and unload so that they are **obviously** terminating

$$\mathsf{Config} \;=\; \mathbb{N} \times \mathsf{Stack}$$

$$
\begin{array}{lll}
\mathsf{unload} & : \mathbb{N} \to \mathsf{Stack} \to \mathsf{Config} \uplus \mathbb{N} & \\
\mathsf{unload}\ v\ \mathsf{Top} & = & \mathsf{inj}_2\ v \\
\mathsf{unload}\ v\ (\mathsf{Right}\ v'\ stk) & = & \mathsf{unload}\ (v' + v)\ stk \\
\mathsf{unload}\ v\ (\mathsf{Left}\ e\ stk) & = & \mathsf{load}\ e\ (\mathsf{Right}\ v\ stk)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{load} & : \mathsf{Expr} \to \mathsf{Stack} \to \mathsf{Config} \uplus \mathbb{N} & \\
\mathsf{load}\ (\mathsf{Val}\ n)\ stk & = & \mathsf{inj}_1\ (n\ ,\ stk) \\
\mathsf{load}\ (\mathsf{Add}\ e_1\ e_2)\ stk & = & \mathsf{load}\ e_1\ (\mathsf{Left}\ e_2\ stk)
\end{array}
$$

Terminating load and unload

- We rewrite load and unload so that they are **obviously** terminating

$$\mathsf{Config} = \mathbb{N} \times \mathsf{Stack}$$

$$\begin{array}{lcl}
\mathsf{unload} : \mathbb{N} \to \mathsf{Stack} & \to & \mathsf{Config} \uplus \mathbb{N} \\
\mathsf{unload}\ v\ \mathsf{Top} & = & \mathsf{inj}_2\ v \\
\mathsf{unload}\ v\ (\mathsf{Right}\ v'\ stk) & = & \mathsf{unload}\ (v' + v)\ stk \\
\mathsf{unload}\ v\ (\mathsf{Left}\ e\ stk) & = & \mathsf{load}\ e\ (\mathsf{Right}\ v\ stk)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{load} : \mathsf{Expr} \to \mathsf{Stack} & \to & \mathsf{Config} \uplus \mathbb{N} \\
\mathsf{load}\ (\mathsf{Val}\ n)\ stk & = & \mathsf{inj}_1\ (n, stk) \\
\mathsf{load}\ (\mathsf{Add}\ e_1\ e_2)\ stk & = & \mathsf{load}\ e_1\ (\mathsf{Left}\ e_2\ stk)
\end{array}$$

Terminating load and unload

- We rewrite load and unload so that they are **obviously** terminating

$$\mathsf{Config} = \mathbb{N} \times \mathsf{Stack}$$

$$\begin{aligned}
&\mathsf{unload} : \mathbb{N} \to \mathsf{Stack} \to \mathsf{Config} \uplus \mathbb{N} \\
&\mathsf{unload}\ v\ \mathsf{Top} = \mathsf{inj}_2\ v \\
&\mathsf{unload}\ v\ (\mathsf{Right}\ v'\ stk) = \mathsf{unload}\ (v' + v)\ stk \\
&\mathsf{unload}\ v\ (\mathsf{Left}\ e\ stk) = \mathsf{load}\ e\ (\mathsf{Right}\ v\ stk)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{load} : \mathsf{Expr} \to \mathsf{Stack} \to \mathsf{Config} \uplus \mathbb{N} \\
&\mathsf{load}\ (\mathsf{Val}\ n)\ stk = \mathsf{inj}_1\ (n , stk) \\
&\mathsf{load}\ (\mathsf{Add}\ e_1\ e_2)\ stk = \mathsf{load}\ e_1\ (\mathsf{Left}\ e_2\ stk)
\end{aligned}$$

A tail-recursive evaluator

- Iterate unload until a value is returned

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ (n , stk) = rec (n , stk)
  where
    rec : Config → ℕ
    rec (n , stk) with unload n stk
    ... | inj₁ (n' , stk') = rec (n' , stk')
    ... | inj₂ r           = r
```

- (n' , stk') is not structurally smaller than (n , stk)

A tail-recursive evaluator

- Iterate unload until a value is returned

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
   ... | inj₁ (n , stk) = rec (n , stk)
   where
     rec : Config → ℕ
     rec (n , stk) with unload n stk
     ... | inj₁ (n' , stk') = rec (n' , stk')
     ... | inj₂ r           = r
```

- (n', stk') **is not** structurally smaller than (n , stk)

A tail-recursive evaluator (2)

- Using **well-founded** recursion

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj1 (n , stk) = rec (n , stk) □1
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec (n , stk) (acc rs) with unload n stk
    ... | inj1 (n' , stk') = rec (n' , stk') (rs □2 )
    ... | inj2 r           = r
```

```
_<_ : Config → Config → Set
□1 : Acc _<_ (n , stk)
□2 : (n' , stk') < (n , stk)
```

A tail-recursive evaluator (2)

- Using **well-founded** recursion

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ (n , stk) = rec (n , stk) □₁
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec (n , stk) (acc rs) with unload n stk
    ... | inj₁ (n' , stk') = rec (n' , stk') (rs □₂ )
    ... | inj₂ r           = r

_<_ : Config → Config → Set
□₁ : Acc _<_ (n , stk)
□₂ : (n' , stk') < (n , stk)
```

A tail-recursive evaluator (2)

- Using **well-founded** recursion

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ (n , stk) = rec (n , stk) □₁
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec (n , stk) (acc rs) with unload n stk
    ... | inj₁ (n' , stk') = rec (n' , stk') (rs □₂ )
    ... | inj₂ r           = r
```

```
_<_ : Config → Config → Set
□₁ : Acc _<_ (n , stk)
□₂ : (n' , stk') < (n , stk)
```

A tail-recursive evaluator (2)

- Using **well-founded** recursion

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ (n , stk) = rec (n , stk) □₁
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec (n , stk) (acc rs) with unload n stk
    ... | inj₁ (n' , stk') = rec (n' , stk') (rs □₂ )
    ... | inj₂ r           = r

_<_ : Config → Config → Set
□₁ : Acc _<_ (n , stk)
□₂ : (n' , stk') < (n , stk)
```

Invariant preserving configurations §3

- The Config type is **too** liberal
- x : Config and y : Config might denote states of the evaluation over different Expr
- We can use dependent types to *statically* **enforce** the invariant

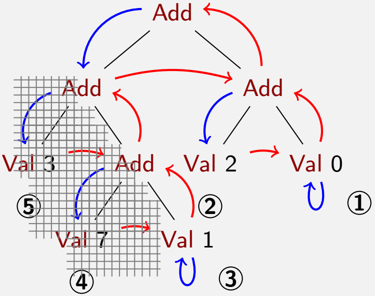
Invariant preserving configurations

- The Config type is **too** liberal
- x : Config and y : Config might denote states of the evaluation over <u>different</u> Expr
- We can use dependent types to *statically* **enforce** the invariant

Invariant preserving configurations

- The Config type is **too** liberal
- x : Config and y : Config might denote states of the evaluation over different Expr
- We can use dependent types to *statically* **enforce** the invariant

Invariant preserving configurations (2)



1 , [Right 7, Right 3, Left Add Val 2 Val 0]

Invariant preserving configurations (3)

- Modify the **stack** to remember subexpressions

```
data Stack⁺ : Set where
  Left  : Expr → Stack⁺ → Stack⁺
  Right : (n : ℕ) → (e : Expr) → eval e ≡ n
        → Stack⁺ → Stack⁺
  Top   : Stack⁺
```

- Recover the **input** expression

```
plug⇑ : Expr → Stack⁺ → Expr
plug⇑ e Top               = e
plug⇑ e (Left t      stk) = plug⇑ (Add e t) stk
plug⇑ e (Right _ t _ stk) = plug⇑ (Add t e) stk

data Config⇑ (e : Expr) : Set where
  _,_ : (c : Config) → plugC⇑ c ≡ e → Config⇑ e
```

Invariant preserving configurations (3)

- Modify the **stack** to remember subexpressions

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Stack}^{+} : \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{Left} : \mathsf{Expr} \to \mathsf{Stack}^{+} \to \mathsf{Stack}^{+}\\
&\quad \mathsf{Right} : (n : \mathbb{N}) \to (e : \mathsf{Expr}) \to \mathsf{eval}\ e \equiv n\\
&\qquad\quad \to \mathsf{Stack}^{+} \to \mathsf{Stack}^{+}\\
&\quad \mathsf{Top} : \mathsf{Stack}^{+}
\end{aligned}
$$

- Recover the **input** expression

$$
\begin{aligned}
&\mathsf{plug}_{\Uparrow} : \mathsf{Expr} \to \mathsf{Stack}^{+} \to \mathsf{Expr}\\
&\mathsf{plug}_{\Uparrow}\ e\ \mathsf{Top} = e\\
&\mathsf{plug}_{\Uparrow}\ e\ (\mathsf{Left}\ t\ stk) = \mathsf{plug}_{\Uparrow}\ (\mathsf{Add}\ e\ t)\ stk\\
&\mathsf{plug}_{\Uparrow}\ e\ (\mathsf{Right}\ \_\ t\ \_\ stk) = \mathsf{plug}_{\Uparrow}\ (\mathsf{Add}\ t\ e)\ stk
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Config}_{\Uparrow}\ (e : \mathsf{Expr}) : \mathsf{Set}\ \mathsf{where}\\
&\quad \_,\_ : (c : \mathsf{Config}) \to \mathsf{plugC}_{\Uparrow}\ c \equiv e \to \mathsf{Config}_{\Uparrow}\ e
\end{aligned}
$$

Well-founded Expr traversal

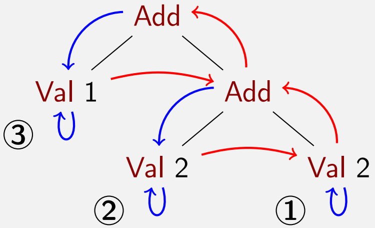
- load and unload traverse the Expr left to right
- Each Config $expr_1$ denotes a leaf of the input expression

Well-founded Expr traversal

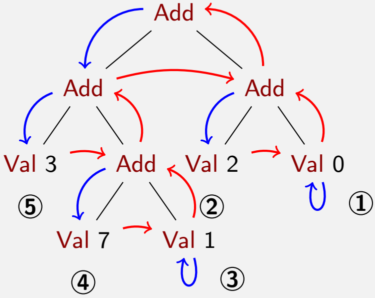
- load and unload traverse the Expr left to right
- Each Config $expr_1$ denotes a leaf of the input expression

Well-founded Expr traversal

- load and unload traverse the Expr left to right
- Each Config $expr_1$ denotes a leaf of the input expression



Up and down configurations



1 , [Right 7, Right 3, Left Add (Val 2) (Val 0)]

7 , [Left Val 1, Right 3, Left Add (Val 2) (Val 0)]

Up and down configurations (2)

- A **reversed** view of the stack

$$
\begin{array}{lll}
\mathsf{plug}_{\Downarrow} & : \mathsf{Expr} \to \mathsf{Stack}^{+} \to \mathsf{Expr} & \\
\mathsf{plug}_{\Downarrow}\ e\ \mathsf{Top} & = & e \\
\mathsf{plug}_{\Downarrow}\ e\ (\mathsf{Left}\ t\ \ \mathit{stk}) & = & \mathsf{Add}\ (\mathsf{plug}_{\Downarrow}\ e\ \mathit{stk})\ t \\
\mathsf{plug}_{\Downarrow}\ e\ (\mathsf{Right}\ \_\ t\ \_\ \mathit{stk}) & = & \mathsf{Add}\ t\ (\mathsf{plug}_{\Downarrow}\ e\ \mathit{stk})
\end{array}
$$

- **Top-down** type-indexed configurations

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Config}_{\Downarrow}\ (e : \mathsf{Expr}) : \mathsf{Set}\ \mathsf{where} \\
\quad \_,\_ : (c : \mathsf{Config}) \to \mathsf{plugC}_{\Downarrow}\ c \equiv e \to \mathsf{Config}_{\Downarrow}\ e
\end{array}
$$

Up and down configurations (2)

- A **reversed** view of the stack

$$
\begin{array}{lll}
\mathsf{plug}_{\Downarrow} : \mathsf{Expr} \to \mathsf{Stack}^{+} \to \mathsf{Expr} & & \\
\mathsf{plug}_{\Downarrow}\ e\ \mathsf{Top} & = & e \\
\mathsf{plug}_{\Downarrow}\ e\ (\mathsf{Left}\ t\ \ stk) & = & \mathsf{Add}\ (\mathsf{plug}_{\Downarrow}\ e\ stk)\ t \\
\mathsf{plug}_{\Downarrow}\ e\ (\mathsf{Right}\ \_\ t\ \_\ stk) & = & \mathsf{Add}\ t\ (\mathsf{plug}_{\Downarrow}\ e\ stk)
\end{array}
$$

- **Top-down** type-indexed configurations

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Config}_{\Downarrow}\ (e : \mathsf{Expr}) : \mathsf{Set}\ \mathsf{where} \\
\quad \_,\_ : (c : \mathsf{Config}) \to \mathsf{plugC}_{\Downarrow}\ c \equiv e \to \mathsf{Config}_{\Downarrow}\ e
\end{array}
$$

Up and down configurations (3)

- Convert between views of the stack

$$
\begin{aligned}
&\mathsf{convert} : \mathsf{Config} \to \mathsf{Config}\\
&\mathsf{convert}\ (n , s) = (n , \mathsf{reverse}\ s)\\
&\mathsf{plug}_{\Downarrow}\text{-to-}\mathsf{plug}_{\Uparrow} : \forall\ (c : \mathsf{Config})\\
&\qquad \to \mathsf{plugC}_{\Downarrow}\ c \equiv \mathsf{plugC}_{\Uparrow}\ (\mathsf{convert}\ c)
\end{aligned}
$$

- Invariant preserving conversion

$$
\begin{aligned}
&\mathsf{Config}_{\Downarrow}\text{-to-}\mathsf{Config}_{\Uparrow} : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Downarrow}\ e \to \mathsf{Config}_{\Uparrow}\ e\\
&\mathsf{Config}_{\Uparrow}\text{-to-}\mathsf{Config}_{\Downarrow} : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Uparrow}\ e \to \mathsf{Config}_{\Downarrow}\ e
\end{aligned}
$$

Up and down configurations (3)

- Convert between views of the stack

$$\begin{aligned}
&\mathsf{convert} : \mathsf{Config} \to \mathsf{Config}\\
&\mathsf{convert}\ (n , s) = (n , \mathsf{reverse}\ s)\\
&\mathsf{plug}_{\Downarrow}\text{-}\mathsf{to}\text{-}\mathsf{plug}_{\Uparrow} : \forall\ (c : \mathsf{Config})\\
&\qquad \to \mathsf{plugC}_{\Downarrow}\ c \equiv \mathsf{plugC}_{\Uparrow}\ (\mathsf{convert}\ c)
\end{aligned}$$

- Invariant preserving conversion

$$\begin{aligned}
&\mathsf{Config}_{\Downarrow}\text{-}\mathsf{to}\text{-}\mathsf{Config}_{\Uparrow} : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Downarrow}\ e \to \mathsf{Config}_{\Uparrow}\ e\\
&\mathsf{Config}_{\Uparrow}\text{-}\mathsf{to}\text{-}\mathsf{Config}_{\Downarrow} : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Uparrow}\ e \to \mathsf{Config}_{\Downarrow}\ e
\end{aligned}$$

Ordering configurations

- We use $\mathsf{Config}_{\Uparrow}$ to **compute**
- We use $\mathsf{Config}_{\Downarrow}$ to prove **termination**

```
data ⌞_⌟_<_ : (e : Expr) → Config⇓ e → Config⇓ e → Set where
  <-StepR : ⌞ r ⌟ ((t₁ , s₁) , ...) < ((t₂ , s₂) , ...)
    → ⌞ Add l r ⌟ ((t₁ , Right l n eq s₁) , eq₁) < ((t₂ , Right l n eq s₂) , ...
  <-StepL : ⌞ l ⌟ ((t₁ , s₁) , ...) < ((t₂ , s₂) , ...)
    → ⌞ Add l r ⌟ ((t₁ , Left r s₁) , eq₁) < ((t₂ , Left r s₂) , eq₂)
  <-Base : (eq₁ : Add e₁ e₂ ≡ Add e₁ (plugC⇓ t₁ s₁))
    →      (eq₂ : Add e₁ e₂ ≡ Add (plugC⇓ t₂ s₂) e₂)
    → ⌞ Add e₁ e₂ ⌟
         ((t₁ , Right n e₁ eq s₁) , eq₁) < ((t₂ , Left e₂ s₂) , eq₂)
```

Ordering configurations

- We use Config$_{\Uparrow}$ to **compute**
- We use Config$_{\Downarrow}$ to prove **termination**

```
data ⌞_⌟_<_ : (e : Expr) → Config⇓ e → Config⇓ e → Set where
  <-StepR : ⌞ r ⌟ ((t1 , s1) , ...) < ((t2 , s2) , ...)
    → ⌞ Add l r ⌟ ((t1 , Right l n eq s1) , eq1) < ((t2 , Right l n eq s2) ,
  <-StepL : ⌞ l ⌟ ((t1 , s1) , ...) < ((t2 , s2) , ...)
    → ⌞ Add l r ⌟ ((t1 , Left r s1) , eq1) < ((t2 , Left r s2) , eq2)
  <-Base : (eq1 : Add e1 e2 ≡ Add e1 (plugC⇓ t1 s1))
    →      (eq2 : Add e1 e2 ≡ Add (plugC⇓ t2 s2) e2)
    → ⌞ Add e1 e2 ⌟
           ((t1 , Right n e1 eq s1) , eq1) < ((t2 , Left e2 s2) , eq2)
```

Ordering configurations (2)

- The relation is **well-founded**

```
<-WF : ∀ (e : Expr) → Well-founded (⌞ e ⌟_<_)
<-WF e x = acc (aux e x)
  where
    aux : ∀ (e : Expr) (x y : Config⇓ e)
        → ⌞ e ⌟ y < x → Acc (⌞ e ⌟_<_) y
    aux = ...
```

- Indexing the relation by *e* is **necessary** for the proof!

Ordering configurations (2)

- The relation is **well-founded**

$$
\begin{array}{l}
\text{<-WF} : \forall\, (e : \mathsf{Expr}) \rightarrow \mathsf{Well\text{-}founded}\ (\llcorner\, e\, \lrcorner\_<\_) \\
\text{<-WF}\ e\ x = \mathsf{acc}\ (\mathsf{aux}\ e\ x) \\
\quad \text{where} \\
\qquad \mathsf{aux} : \forall\, (e : \mathsf{Expr})\ (x\ y : \mathsf{Config}_{\Downarrow}\ e) \\
\qquad\quad \rightarrow \llcorner\, e\, \lrcorner\ y < x \rightarrow \mathsf{Acc}\ (\llcorner\, e\, \lrcorner\_<\_)\ y \\
\qquad \mathsf{aux} = \ldots
\end{array}
$$

- Indexing the relation by e is **necessary** for the proof!

Completing the evaluator

- Invariant preserving step

$$
\begin{array}{l}
\mathsf{step} : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Uparrow}\ e \to \mathsf{Config}_{\Uparrow}\ e \uplus \mathbb{N} \\
\mathsf{step}\ e\ ((n\ ,\ stk)\ ,\ eq) \\
\quad \mathsf{with}\ \mathsf{unload}^{+}\ n\ (\mathsf{Val}\ n)\ \mathsf{refl}\ stk \\
\quad \ldots \mid \mathsf{inj}_1\ (n'\ ,\ stk') = \mathsf{inj}_1\ ((n'\ ,\ stk')\ ,\ \ldots) \\
\quad \ldots \mid \mathsf{inj}_2\ v \qquad\quad\ \ = \mathsf{inj}_2\ v
\end{array}
$$

- step delivers a smaller configuration

$$
\begin{array}{l}
\mathsf{step}\text{-}{<} : \forall\ (e : \mathsf{Expr}) \to (c\ c' : \mathsf{Config}_{\Uparrow}\ e) \\
\qquad \to \mathsf{step}\ e\ c \equiv \mathsf{inj}_1\ c' \\
\qquad \to \llcorner\ e\ \lrcorner\ \mathsf{Config}_{\Uparrow}\text{-to-}\mathsf{Config}_{\Downarrow}\ c' < \mathsf{Config}_{\Uparrow}\text{-to-}\mathsf{Config}_{\Downarrow}\ c
\end{array}
$$

Completing the evaluator

- Invariant preserving step

$$\begin{array}{l}
\mathsf{step} : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Uparrow}\ e \to \mathsf{Config}_{\Uparrow}\ e \uplus \mathbb{N} \\
\mathsf{step}\ e\ ((n\ ,\ stk)\ ,\ eq) \\
\quad \mathsf{with}\ \mathsf{unload}^{+}\ n\ (\mathsf{Val}\ n)\ \mathsf{refl}\ stk \\
\quad \ldots \mid \mathsf{inj}_1\ (n'\ ,\ stk') = \mathsf{inj}_1\ ((n'\ ,\ stk')\ ,\ \ldots) \\
\quad \ldots \mid \mathsf{inj}_2\ v \qquad\qquad\ = \mathsf{inj}_2\ v
\end{array}$$

- step delivers a **smaller** configuration

$$\begin{array}{l}
\mathsf{step\text{-}}{<} : \forall\,(e : \mathsf{Expr}) \to (c\ c' : \mathsf{Config}_{\Uparrow}\ e) \\
\qquad \to \mathsf{step}\ e\ c \equiv \mathsf{inj}_1\ c' \\
\qquad \to \llcorner\ e\ \lrcorner\ \mathsf{Config}_{\Uparrow}\text{-}\mathsf{to}\text{-}\mathsf{Config}_{\Downarrow}\ c' < \mathsf{Config}_{\Uparrow}\text{-}\mathsf{to}\text{-}\mathsf{Config}_{\Downarrow}\ c
\end{array}$$

A terminating tail-recursive evaluator

- Auxiliary recursor

$$
\begin{array}{l}
\mathsf{rec} : (e : \mathsf{Expr}) \to (c : \mathsf{Config}_{\Uparrow}\ e) \\
\quad \to \mathsf{Acc}\ (\llcorner\ e\ \lrcorner\_{<}\_)\ (\mathsf{Config}_{\Uparrow}\text{-to-}\mathsf{Config}_{\Downarrow}\ c) \\
\quad \to \mathsf{Config}_{\Uparrow}\ e \uplus \mathbb{N} \\
\mathsf{rec}\ e\ c\ (\mathsf{acc}\ rs) = \mathsf{with}\ \mathsf{step}\ e\ c \mid \mathsf{inspect}\ (\mathsf{step}\ e)\ c \\
\ldots \mid \mathsf{inj}_2\ n \mid \_ = \mathsf{inj}_2\ n \\
\ldots \mid \mathsf{inj}_1\ c' \mid [\ ls\ ] \\
\quad = \mathsf{rec}\ e\ c'\ (rs\ (\mathsf{Config}_{\Uparrow}\text{-to-}\mathsf{Config}_{\Downarrow}\ c')\ (\mathsf{step}\text{-}{<}\ e\ c\ c'\ ls))
\end{array}
$$

- Tail-recursive evaluator

$$
\begin{array}{l}
\mathsf{tail\text{-}rec\text{-}eval} : \mathsf{Expr} \to \mathbb{N} \\
\mathsf{tail\text{-}rec\text{-}eval}\ e\ \mathsf{with}\ \mathsf{load}\ e\ \mathsf{Top} \\
\ldots \mid \mathsf{inj}_1\ c = \mathsf{rec}\ e\ (c, \ldots)\ ({<}\text{-WF}\ e\ c)
\end{array}
$$

A terminating tail-recursive evaluator

- Auxiliary recursor

$$
\begin{aligned}
&\mathsf{rec} : (e : \mathsf{Expr}) \to (c : \mathsf{Config}_{\Uparrow}\ e) \\
&\quad \to \mathsf{Acc}\ (\llcorner\ e\ \lrcorner\_<\_)\ (\mathsf{Config}_{\Uparrow}\text{-to-}\mathsf{Config}_{\Downarrow}\ c) \\
&\quad \to \mathsf{Config}_{\Uparrow}\ e \uplus \mathbb{N} \\
&\mathsf{rec}\ e\ c\ (\mathsf{acc}\ rs) = \mathsf{with}\ \mathsf{step}\ e\ c \mid \mathsf{inspect}\ (\mathsf{step}\ e)\ c \\
&\ldots \quad \mid \mathsf{inj}_2\ n \mid \_ = \mathsf{inj}_2\ n \\
&\ldots \quad \mid \mathsf{inj}_1\ c' \mid [\ ls\ ] \\
&\quad = \mathsf{rec}\ e\ c'\ (rs\ (\mathsf{Config}_{\Uparrow}\text{-to-}\mathsf{Config}_{\Downarrow}\ c')\ (\mathsf{step}\text{-}{<}\ e\ c\ c'\ ls))
\end{aligned}
$$

- Tail-recursive evaluator

$$
\begin{aligned}
&\mathsf{tail\text{-}rec\text{-}eval} : \mathsf{Expr} \to \mathbb{N} \\
&\mathsf{tail\text{-}rec\text{-}eval}\ e\ \mathsf{with}\ \mathsf{load}\ e\ \mathsf{Top} \\
&\ldots \mid \mathsf{inj}_1\ c = \mathsf{rec}\ e\ (c , \ldots)\ ({<}\text{-}\mathsf{WF}\ e\ c)
\end{aligned}
$$

Correctness

- rec is correct by induction over Acc

```
rec-correct : ∀ (e : Expr) → (c : Config⇑ e)
    → (ac : Acc (⌞ e ⌟_<_) (Config⇑-to-Config⇓ c))
    → eval e ≡ rec e c ac
rec-correct e c (acc rs)
  with step e c | inspect (step e) c
... | inj₁ c' | [ ls ]
    = rec-correct e c' (rs (Config⇑-to-Config⇓ c') (step-< e c c' ls))
... | inj₂ n  | [ ls ] = step-correct e c n ls
```

- tail-rec-eval is correct

```
correctness : ∀ (e : Expr) → eval e ≡ tail-rec-eval e
```

Correctness

- rec is correct by induction over Acc

```
rec-correct : ∀ (e : Expr) → (c : Config⇑ e)
    → (ac : Acc (⌞ e ⌟_<_) (Config⇑-to-Config⇓ c))
    → eval e ≡ rec e c ac
rec-correct e c (acc rs)
   with step e c | inspect (step e) c
... | inj₁ c′ | [ ls ]
    = rec-correct e c′ (rs (Config⇑-to-Config⇓ c′) (step-< e c c′ ls))
... | inj₂ n  | [ ls ] = step-correct e c n ls
```

- tail-rec-eval is correct

```
correctness : ∀ (e : Expr) → eval e ≡ tail-rec-eval e
```

4. A generic tail-recursive evaluator

regular: Universe + Interpretation

$$
\begin{array}{lll}
\text{data } \mathsf{Reg} : \mathsf{Set}_1 \text{ where} & & \\
\quad \mathbb{0} & : & \mathsf{Reg} \\
\quad \mathbb{1} & : & \mathsf{Reg} \\
\quad \mathsf{I} & : & \mathsf{Reg} \\
\quad \mathsf{K} & : & (A : \mathsf{Set}) \to \mathsf{Reg} \\
\quad \_\oplus\_ & : & (R\ Q : \mathsf{Reg}) \to \mathsf{Reg} \\
\quad \_\otimes\_ & : & (R\ Q : \mathsf{Reg}) \to \mathsf{Reg}
\end{array}
$$

$$
\begin{array}{lll}
[\![\_]\!] : \mathsf{Reg} \to \mathsf{Set} \to \mathsf{Set} & & \\
[\![\ \mathbb{0}\ ]\!]\ X & = & \bot \\
[\![\ \mathbb{1}\ ]\!]\ X & = & \top \\
[\![\ \mathsf{I}\ ]\!]\ X & = & X \\
[\![\ (\mathsf{K}\ A)\ ]\!]\ X & = & A \\
[\![\ (R \oplus Q)\ ]\!]\ X & = & [\![\ R\ ]\!]\ X \uplus [\![\ Q\ ]\!]\ X \\
[\![\ (R \otimes Q)\ ]\!]\ X & = & [\![\ R\ ]\!]\ X \times [\![\ Q\ ]\!]\ X
\end{array}
$$

- Values of type $[\![\ R\ ]\!]\ X$ are functors over X

$$\mathsf{fmap} : (R : \mathsf{Reg}) \to (X \to Y) \to [\![\ R\ ]\!]\ X \to [\![\ R\ ]\!]\ Y$$

regular: Universe + Interpretation

$$
\begin{array}{lll}
\mathsf{data}\ \mathsf{Reg} : \mathsf{Set}_1\ \mathsf{where} & & \\
\quad \mathbb{0} & : & \mathsf{Reg} \\
\quad \mathbb{1} & : & \mathsf{Reg} \\
\quad \mathsf{I} & : & \mathsf{Reg} \\
\quad \mathsf{K} & : & (A : \mathsf{Set}) \to \mathsf{Reg} \\
\quad \_\oplus\_ & : & (R\ Q : \mathsf{Reg}) \to \mathsf{Reg} \\
\quad \_\otimes\_ & : & (R\ Q : \mathsf{Reg}) \to \mathsf{Reg}
\end{array}
$$

$$
\begin{array}{lll}
[\![\_]\!] : \mathsf{Reg} \to \mathsf{Set} \to \mathsf{Set} & & \\
[\![\ \mathbb{0}\ ]\!]\ X & = & \bot \\
[\![\ \mathbb{1}\ ]\!]\ X & = & \top \\
[\![\ \mathsf{I}\ ]\!]\ X & = & X \\
[\![\ (\mathsf{K}\ A)\ ]\!]\ X & = & A \\
[\![\ (R \oplus Q)\ ]\!]\ X & = & [\![\ R\ ]\!]\ X \uplus [\![\ Q\ ]\!]\ X \\
[\![\ (R \otimes Q)\ ]\!]\ X & = & [\![\ R\ ]\!]\ X \times [\![\ Q\ ]\!]\ X
\end{array}
$$

- Values of type $[\![\ R\ ]\!]\ X$ are functors over X

$$\mathsf{fmap} : (R : \mathsf{Reg}) \to (X \to Y) \to [\![\ R\ ]\!]\ X \to [\![\ R\ ]\!]\ Y$$

regular: Universe + Interpretation

$$
\begin{array}{lll}
\text{data Reg} & : \text{Set}_1 \text{ where} & \\
\quad \mathbb{0} & : \text{Reg} & \\
\quad \mathbb{1} & : \text{Reg} & \\
\quad \mathsf{I} & : \text{Reg} & \\
\quad \mathsf{K} & : (A : \text{Set}) \to \text{Reg} & \\
\quad \_\oplus\_ & : (R\ Q : \text{Reg}) \to \text{Reg} & \\
\quad \_\otimes\_ & : (R\ Q : \text{Reg}) \to \text{Reg} &
\end{array}
$$

$$
\begin{array}{lcl}
[\![\_]\!] : \text{Reg} \to \text{Set} \to \text{Set} & & \\
[\![\ \mathbb{0}\ ]\!]\ X & = & \bot \\
[\![\ \mathbb{1}\ ]\!]\ X & = & \top \\
[\![\ \mathsf{I}\ ]\!]\ X & = & X \\
[\![\ (\mathsf{K}\ A)\ ]\!]\ X & = & A \\
[\![\ (R \oplus Q)\ ]\!]\ X & = & [\![\ R\ ]\!]\ X \uplus [\![\ Q\ ]\!]\ X \\
[\![\ (R \otimes Q)\ ]\!]\ X & = & [\![\ R\ ]\!]\ X \times [\![\ Q\ ]\!]\ X
\end{array}
$$

- Values of type $[\![\ R\ ]\!]\ X$ are functors over X

$$\text{fmap} : (R : \text{Reg}) \to (X \to Y) \to [\![\ R\ ]\!]\ X \to [\![\ R\ ]\!]\ Y$$

regular: Universe + Interpretation (2)

- Fixed point

$$\begin{array}{l} \mathsf{data}\ \mu\ (R\ :\ \mathsf{Reg})\ :\ \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{In}\ :\ [\![\ R\ ]\!]\ (\mu\ R)\ \rightarrow\ \mu\ R \end{array}$$

- Fold (catamorphism)

$$\begin{array}{l} \mathsf{cata}\ :\ (R\ :\ \mathsf{Reg})\ \rightarrow\ ([\![\ R\ ]\!]\ X\ \rightarrow\ X)\ \rightarrow\ \mu\ R\ \rightarrow\ X \\ \mathsf{cata}\ R\ \psi\ (\mathsf{In}\ r)\ =\ \psi\ (\mathsf{fmap}\ R\ (\mathsf{cata}\ R\ \psi)\ r) \end{array}$$

regular: Universe + Interpretation (2)

- Fixed point

$$\begin{array}{l} \mathsf{data}\ \mu\ (R : \mathsf{Reg}) : \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{In} : [\![\, R \,]\!]\ (\mu\ R) \to \mu\ R \end{array}$$

- Fold (catamorphism)

$$\begin{array}{l} \mathsf{cata} : (R : \mathsf{Reg}) \to ([\![\, R \,]\!]\ X \to X) \to \mu\ R \to X \\ \mathsf{cata}\ R\ \psi\ (\mathsf{In}\ r) = \psi\ (\mathsf{fmap}\ R\ (\mathsf{cata}\ R\ \psi)\ r) \end{array}$$

regular: Example

$$
\begin{aligned}
&\mathsf{exprF} : \mathsf{Reg} \\
&\mathsf{exprF} = \mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I})
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{from} : \mathsf{Expr} \to \mathsf{Expr}^G \\
&\mathsf{from}\ (\mathsf{Val}\ n) = \mathsf{inj}_1\ n \\
&\mathsf{from}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{inj}_2\ (\mathsf{from}\ e_1\ ,\ \mathsf{from}\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Expr}^G : \mathsf{Set} \\
&\mathsf{Expr}^G = \mu\ \mathsf{exprF}
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{to} : \mathsf{Expr}^G \to \mathsf{Expr} \\
&\mathsf{to}\ (\mathsf{inj}_1\ n) = \mathsf{Val}\ n \\
&\mathsf{to}\ (\mathsf{inj}_2\ (e_1\ ,\ e_2)) = \mathsf{Add}\ (\mathsf{to}\ e_1)\ (\mathsf{to}\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr}^G \to \mathbb{N} \\
&\mathsf{eval} = \mathsf{cata}\ \mathsf{exprF}\ \phi \\
&\quad \text{where } \phi : [\![\ \mathsf{exprF}\ ]\!]\ \mathbb{N} \to \mathbb{N} \\
&\qquad\quad \phi\ (\mathsf{inj}_1\ n) = n \\
&\qquad\quad \phi\ (\mathsf{inj}_2\ (n\ ,\ n')) = n + n'
\end{aligned}
$$

regular: Example

$$
\begin{aligned}
&\mathsf{exprF} : \mathsf{Reg} \\
&\mathsf{exprF} = \mathsf{K}\,\mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}) \\
&\mathsf{Expr}^G : \mathsf{Set} \\
&\mathsf{Expr}^G = \mu\,\mathsf{exprF}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{from} : \mathsf{Expr} \to \mathsf{Expr}^G \\
&\mathsf{from}\ (\mathsf{Val}\ n) = \mathsf{inj}_1\ n \\
&\mathsf{from}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{inj}_2\ (\mathsf{from}\ e_1\ ,\ \mathsf{from}\ e_2) \\
&\mathsf{to} : \mathsf{Expr}^G \to \mathsf{Expr} \\
&\mathsf{to}\ (\mathsf{inj}_1\ n) = \mathsf{Val}\ n \\
&\mathsf{to}\ (\mathsf{inj}_2\ (e_1\ ,\ e_2)) = \mathsf{Add}\ (\mathsf{to}\ e_1)\ (\mathsf{to}\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr}^G \to \mathbb{N} \\
&\mathsf{eval} = \mathsf{cata}\ \mathsf{exprF}\ \phi \\
&\quad \mathsf{where}\ \phi : [\![\ \mathsf{exprF}\ ]\!]\ \mathbb{N} \to \mathbb{N} \\
&\quad\quad \phi\ (\mathsf{inj}_1\ n) = n \\
&\quad\quad \phi\ (\mathsf{inj}_2\ (n\ ,\ n')) = n + n'
\end{aligned}
$$

regular: Example

$$
\begin{aligned}
&\mathsf{exprF} : \mathsf{Reg} \\
&\mathsf{exprF} = \mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}) \\
&\mathsf{Expr}^G : \mathsf{Set} \\
&\mathsf{Expr}^G = \mu\ \mathsf{exprF}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{from} : \mathsf{Expr} \to \mathsf{Expr}^G \\
&\mathsf{from}\ (\mathsf{Val}\ n) = \mathsf{inj}_1\ n \\
&\mathsf{from}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{inj}_2\ (\mathsf{from}\ e_1 , \mathsf{from}\ e_2) \\
&\mathsf{to} : \mathsf{Expr}^G \to \mathsf{Expr} \\
&\mathsf{to}\ (\mathsf{inj}_1\ n) = \mathsf{Val}\ n \\
&\mathsf{to}\ (\mathsf{inj}_2\ (e_1 , e_2)) = \mathsf{Add}\ (\mathsf{to}\ e_1)\ (\mathsf{to}\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Expr}^G \to \mathbb{N} \\
&\mathsf{eval} = \mathsf{cata}\ \mathsf{exprF}\ \phi \\
&\quad \mathsf{where}\ \phi : [\![\ \mathsf{exprF}\ ]\!]\ \mathbb{N} \to \mathbb{N} \\
&\quad\quad \phi\ (\mathsf{inj}_1\ n) = n \\
&\quad\quad \phi\ (\mathsf{inj}_2\ (n , n')) = n + n'
\end{aligned}
$$

Dissection

- Another interpretation: codes $\rightarrow$ bifunctors

$$
\begin{aligned}
&\nabla : (R : \mathsf{Reg}) \rightarrow (\mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set})\\
&\nabla\ \mathbb{0}\ X\ Y = \bot\\
&\nabla\ \mathbb{1}\ X\ Y = \bot\\
&\nabla\ \mathsf{I}\ X\ Y = \top\\
&\nabla\ (\mathsf{K}\ A)\ X\ Y = \bot\\
&\nabla\ (R \oplus Q)\ X\ Y = \nabla\ R\ X\ Y \uplus \nabla\ Q\ X\ Y\\
&\nabla\ (R \otimes Q)\ X\ Y = (\nabla\ R\ X\ Y \times [\![\, Q \,]\!]\ Y) \uplus ([\![\, R \,]\!]\ X \times \nabla\ Q\ X\ Y)
\end{aligned}
$$

Example: $\nabla\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ X\ Y$

$$
\begin{aligned}
&= \nabla\ (\mathsf{K}\ \mathbb{N})\ X\ Y \uplus \nabla\ (\mathsf{I} \otimes \mathsf{I})\ X\ Y\\
&= (\nabla\ \mathsf{I}\ X\ Y \times [\![\, \mathsf{I} \,]\!]\ Y) \uplus ([\![\, \mathsf{I} \,]\!]\ X \times \nabla\ \mathsf{I}\ X\ Y)\\
&= (\top \times Y) \uplus (X \times \top)\\
&= Y \uplus X
\end{aligned}
$$

Dissection

- Another interpretation: codes → bifunctors

$$
\begin{array}{l}
\nabla : (R : \mathsf{Reg}) \rightarrow (\mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}) \\
\nabla\ \mathbb{0} \qquad X\ Y = \bot \\
\nabla\ \mathbb{1} \qquad X\ Y = \bot \\
\nabla\ \mathsf{I} \qquad X\ Y = \top \\
\nabla\ (\mathsf{K}\ A) \quad X\ Y = \bot \\
\nabla\ (R \oplus Q)\ X\ Y = \nabla\ R\ X\ Y \uplus \nabla\ Q\ X\ Y \\
\nabla\ (R \otimes Q)\ X\ Y = (\nabla\ R\ X\ Y \times [\![\ Q\ ]\!]\ Y) \uplus ([\![\ R\ ]\!]\ X \times \nabla\ Q\ X\ Y)
\end{array}
$$

Example: $\nabla\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ X\ Y$

$$
\begin{array}{l}
= \nabla\ (\mathsf{K}\ \mathbb{N})\ X\ Y \uplus \nabla\ (\mathsf{I} \otimes \mathsf{I})\ X\ Y \\
= (\nabla\ \mathsf{I}\ X\ Y \times [\![\ \mathsf{I}\ ]\!]\ Y) \uplus ([\![\ \mathsf{I}\ ]\!]\ X \times \nabla\ \mathsf{I}\ X\ Y) \\
= (\top \times Y) \uplus (X \times \top) \\
= Y \uplus X
\end{array}
$$

- Another interpretation: codes $\rightarrow$ bifunctors

$$
\begin{array}{lcl}
\nabla : (R : \mathsf{Reg}) & \rightarrow & (\mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}) \\
\nabla\ \mathbb{0}\ X\ Y & = & \bot \\
\nabla\ \mathbb{1}\ X\ Y & = & \bot \\
\nabla\ \mathsf{I}\ X\ Y & = & \top \\
\nabla\ (\mathsf{K}\ A)\ X\ Y & = & \bot \\
\nabla\ (R \oplus Q)\ X\ Y & = & \nabla\ R\ X\ Y \uplus \nabla\ Q\ X\ Y \\
\nabla\ (R \otimes Q)\ X\ Y & = & (\nabla\ R\ X\ Y \times [\![\ Q\ ]\!]\ Y) \uplus ([\![\ R\ ]\!]\ X \times \nabla\ Q\ X\ Y)
\end{array}
$$

Example: $\nabla\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ X\ Y$

$$
\begin{array}{l}
= \nabla\ (\mathsf{K}\ \mathbb{N})\ X\ Y \uplus \nabla\ (\mathsf{I} \otimes \mathsf{I})\ X\ Y \\
= (\nabla\ \mathsf{I}\ X\ Y \times [\![\ \mathsf{I}\ ]\!]\ Y) \uplus ([\![\ \mathsf{I}\ ]\!]\ X \times \nabla\ \mathsf{I}\ X\ Y) \\
= (\top \times Y) \uplus (X \times \top) \\
= Y \uplus X
\end{array}
$$

Dissection

- Another interpretation: codes → bifunctors

$$
\begin{array}{lcl}
\nabla : (R : \mathsf{Reg}) & \to & (\mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}) \\
\nabla\ \mathbb{0}\ X\ Y & = & \bot \\
\nabla\ \mathbb{1}\ X\ Y & = & \bot \\
\nabla\ \mathsf{I}\ X\ Y & = & \top \\
\nabla\ (\mathsf{K}\ A)\ X\ Y & = & \bot \\
\nabla\ (R \oplus Q)\ X\ Y & = & \nabla\ R\ X\ Y \uplus \nabla\ Q\ X\ Y \\
\nabla\ (R \otimes Q)\ X\ Y & = & (\nabla\ R\ X\ Y \times [\![\, Q \,]\!]\ Y) \uplus ([\![\, R \,]\!]\ X \times \nabla\ Q\ X\ Y)
\end{array}
$$

Example: $\nabla\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ X\ Y$

$$
\begin{array}{l}
= \nabla\ (\mathsf{K}\ \mathbb{N})\ X\ Y \uplus \nabla\ (\mathsf{I} \otimes \mathsf{I})\ X\ Y \\
= (\nabla\ \mathsf{I}\ X\ Y \times [\![\, \mathsf{I} \,]\!]\ Y) \uplus ([\![\, \mathsf{I} \,]\!]\ X \times \nabla\ \mathsf{I}\ X\ Y) \\
= (\top \times Y) \uplus (X \times \top) \\
= Y \uplus X
\end{array}
$$

Dissection

- Another interpretation: codes $\to$ bifunctors

$$
\begin{aligned}
&\nabla : (R : \mathsf{Reg}) \to (\mathsf{Set} \to \mathsf{Set} \to \mathsf{Set})\\
&\nabla\ \mathbb{0}\ X\ Y = \bot\\
&\nabla\ \mathbb{1}\ X\ Y = \bot\\
&\nabla\ \mathsf{I}\ X\ Y = \top\\
&\nabla\ (\mathsf{K}\ A)\ X\ Y = \bot\\
&\nabla\ (R \oplus Q)\ X\ Y = \nabla\ R\ X\ Y \uplus \nabla\ Q\ X\ Y\\
&\nabla\ (R \otimes Q)\ X\ Y = (\nabla\ R\ X\ Y \times [\![\, Q \,]\!]\ Y) \uplus ([\![\, R \,]\!]\ X \times \nabla\ Q\ X\ Y)
\end{aligned}
$$

Example: $\nabla\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ X\ Y$

$$
\begin{aligned}
&= \nabla\ (\mathsf{K}\ \mathbb{N})\ X\ Y \uplus \nabla\ (\mathsf{I} \otimes \mathsf{I})\ X\ Y\\
&= (\nabla\ \mathsf{I}\ X\ Y \times [\![\, \mathsf{I} \,]\!]\ Y) \uplus ([\![\, \mathsf{I} \,]\!]\ X \times \nabla\ \mathsf{I}\ X\ Y)\\
&= (\top \times Y) \uplus (X \times \top)\\
&= Y \uplus X
\end{aligned}
$$

Dissection

- Another interpretation: codes $\rightarrow$ bifunctors

$$
\begin{array}{llcl}
\nabla & : (R : \mathsf{Reg}) & \rightarrow & (\mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}) \\
\nabla\ \mathbb{0} & X\ Y & = & \bot \\
\nabla\ \mathbb{1} & X\ Y & = & \bot \\
\nabla\ \mathsf{I} & X\ Y & = & \top \\
\nabla\ (\mathsf{K}\ A) & X\ Y & = & \bot \\
\nabla\ (R \oplus Q) & X\ Y & = & \nabla\ R\ X\ Y \uplus \nabla\ Q\ X\ Y \\
\nabla\ (R \otimes Q) & X\ Y & = & (\nabla\ R\ X\ Y \times [\![\ Q\ ]\!]\ Y) \uplus ([\![\ R\ ]\!]\ X \times \nabla\ Q\ X\ Y)
\end{array}
$$

Example: $\nabla\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ X\ Y$

$$
\begin{array}{ll}
= & \nabla\ (\mathsf{K}\ \mathbb{N})\ X\ Y \uplus \nabla\ (\mathsf{I} \otimes \mathsf{I})\ X\ Y \\
= & (\nabla\ \mathsf{I}\ X\ Y \times [\![\ \mathsf{I}\ ]\!]\ Y) \uplus ([\![\ \mathsf{I}\ ]\!]\ X \times \nabla\ \mathsf{I}\ X\ Y) \\
= & (\top \times Y) \uplus (X \times \top) \\
= & Y \uplus X
\end{array}
$$

Generic stacks

- Store trees, values and proofs

```
record Computed (R : Reg) (X : Set) (ψ : ⟦ R ⟧ X → X)
                                                : Set where
  constructor _,_,_
  field
    Tree  : μ R
    Value : X
    Proof : cata R ψ Tree ≡ Value
```

- Computed to the left; trees to the right

```
StackG : (R : Reg) → (X : Set)
        → (ψ : ⟦ R ⟧ X → X) → Set
StackG R X ψ = List (∇ R (Computed R X ψ) (μ R))
```

Example: $\text{Stack}^G\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ \mathbb{N}\ \phi$

$= \text{List}\ (\text{Computed}\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ \phi \uplus \mu\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I})))$

$\simeq \text{Stack}^{+}$

Generic stacks

- Store trees, values and proofs

$$\begin{array}{l}
\text{record Computed } (R : \text{Reg})\ (X : \text{Set})\ (\psi : [\![R]\!]\ X \to X) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : \text{Set where} \\
\quad \text{constructor } _,_,_ \\
\quad \text{field} \\
\qquad \text{Tree} \quad : \mu\ R \\
\qquad \text{Value} : X \\
\qquad \text{Proof} : \text{cata } R\ \psi\ \text{Tree} \equiv \text{Value}
\end{array}$$

- Computed to the left; trees to the right

$$\begin{array}{l}
\text{Stack}^G : (R : \text{Reg}) \to (X : \text{Set}) \\
\qquad\quad \to (\psi : [\![R]\!]\ X \to X) \to \text{Set} \\
\text{Stack}^G\ R\ X\ \psi = \text{List}\ (\nabla\ R\ (\text{Computed}\ R\ X\ \psi)\ (\mu\ R))
\end{array}$$

Example: $\text{Stack}^G\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ \mathbb{N}\ \phi$

$$\begin{array}{l}
= \text{List}\ (\text{Computed}\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ \phi \uplus \mu\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))) \\
\simeq \text{Stack}^+
\end{array}$$

Generic stacks

- Store trees, values and proofs

$$
\begin{array}{l}
\text{record Computed } (R : \text{Reg})\ (X : \text{Set})\ (\psi : [\![R]\!]\, X \to X) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : \text{Set where} \\
\quad \text{constructor } \_,\_,\_ \\
\quad \text{field} \\
\qquad \text{Tree} \quad : \mu\, R \\
\qquad \text{Value} : X \\
\qquad \text{Proof} : \text{cata } R\, \psi\ \text{Tree} \equiv \text{Value}
\end{array}
$$

- Computed to the left; trees to the right

$$
\begin{array}{l}
\text{Stack}^G : (R : \text{Reg}) \to (X : \text{Set}) \\
\qquad\quad \to (\psi : [\![R]\!]\, X \to X) \to \text{Set} \\
\text{Stack}^G\, R\, X\, \psi = \text{List}\ (\nabla\ R\ (\text{Computed}\ R\, X\, \psi)\ (\mu\ R))
\end{array}
$$

Example: $\text{Stack}^G\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ \mathbb{N}\ \phi$

$$
\begin{array}{l}
= \text{List}\ (\text{Computed}\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))\ \phi \uplus \mu\ (\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I}))) \\
\simeq \text{Stack}^+
\end{array}
$$

Generic plug

- Plug a *single* layer

$$\text{plug} : (R : \text{Reg}) \to (X \to Y) \to R\,X\,Y \times Y \to [\![R]\!]\,Y$$

- Plug through the *stack*

$$\begin{aligned}
&\text{plug-}\mu_{\Downarrow} : (R : \text{Reg}) \to \{\psi : [\![R]\!]\,X \to X\} \\
&\qquad \to \mu\,R \to \text{Stack}^{G}\,R\,X\,\psi \to \mu\,R \\
&\text{plug-}\mu_{\Downarrow}\,R\,t\,[] = t \\
&\text{plug-}\mu_{\Downarrow}\,R\,t\,(h :: hs) \\
&\quad = \text{In}\,(\text{plug}\,R\,\text{Computed.Tree}\,h\,(\text{plug-}\mu_{\Downarrow}\,R\,t\,hs)) \\
&\text{plug-}\mu_{\Uparrow} : (R : \text{Reg}) \to \{\psi : [\![R]\!]\,X \to X\} \\
&\qquad \to \mu\,R \to \text{Stack}^{G}\,R\,X\,\psi \to \mu\,R \\
&\text{plug-}\mu_{\Uparrow}\,R\,t\,[] = t \\
&\text{plug-}\mu_{\Uparrow}\,R\,t\,(h :: hs) \\
&\quad = \text{plug-}\mu_{\Uparrow}\,R\,(\text{In}\,(\text{plug}\,R\,\text{Computed.Tree}\,h\,t))\,hs
\end{aligned}$$

Generic plug

- Plug a *single* layer

$$\mathsf{plug} : (R : \mathsf{Reg}) \to (X \to Y) \to R\,X\,Y \times Y \to [\![R]\!]\,Y$$

- Plug through the *stack*

$$\begin{aligned}
&\mathsf{plug}\text{-}\mu_{\Downarrow} : (R : \mathsf{Reg}) \to \{\psi : [\![R]\!]\,X \to X\} \\
&\qquad \to \mu\,R \to \mathsf{Stack}^{G}\,R\,X\,\psi \to \mu\,R \\
&\mathsf{plug}\text{-}\mu_{\Downarrow}\,R\,t\,[] = t \\
&\mathsf{plug}\text{-}\mu_{\Downarrow}\,R\,t\,(h :: hs) \\
&\quad = \mathsf{In}\,(\mathsf{plug}\,R\,\mathsf{Computed.Tree}\,h\,(\mathsf{plug}\text{-}\mu_{\Downarrow}\,R\,t\,hs)) \\
&\mathsf{plug}\text{-}\mu_{\Uparrow} : (R : \mathsf{Reg}) \to \{\psi : [\![R]\!]\,X \to X\} \\
&\qquad \to \mu\,R \to \mathsf{Stack}^{G}\,R\,X\,\psi \to \mu\,R \\
&\mathsf{plug}\text{-}\mu_{\Uparrow}\,R\,t\,[] = t \\
&\mathsf{plug}\text{-}\mu_{\Uparrow}\,R\,t\,(h :: hs) \\
&\quad = \mathsf{plug}\text{-}\mu_{\Uparrow}\,R\,(\mathsf{In}\,(\mathsf{plug}\,R\,\mathsf{Computed.Tree}\,h\,t))\,hs
\end{aligned}$$

Generic leaves

There are two levels of **recursion** in a generic tree

- Functor: composition of functors $R \otimes Q$
- Fixed point: $[\![\, R \,]\!] \; (\mu \; R)$

```
data NonRec : (R : Reg) → ⟦ R ⟧ X → Set where
  NonRec-𝟙   : NonRec 𝟙 tt
  NonRec-K   : (B : Set) → (b : B) → NonRec (K B) b
  NonRec-⊕₁  : (R Q : Reg) → (r : ⟦ R ⟧ X)
               → NonRec R r → NonRec (R ⊕ Q) (inj₁ r)
  NonRec-⊕₂  : ...
  NonRec-⊗   : ...
```

Example:

```
Val-NonRec : ∀ (n : ℕ) → NonRec (K ℕ ⊕ (I ⊗ I)) (inj₁ n)
Val-NonRec : n = NonRec-⊕₁ (K ℕ) (I ⊗ I) n (NonRec-K ℕ n)
```

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Generic leaves

There are two levels of **recursion** in a generic tree

- Functor: composition of functors $R \otimes Q$
- Fixed point: $[\![R]\!]\ (\mu\ R)$

```
data NonRec : (R : Reg) → ⟦ R ⟧ X → Set where
  NonRec-𝟙  : NonRec 𝟙 tt
  NonRec-K  : (B : Set) → (b : B) → NonRec (K B) b
  NonRec-⊕₁ : (R Q : Reg) → (r : ⟦ R ⟧ X)
            → NonRec R r → NonRec (R ⊕ Q) (inj₁ r)
  NonRec-⊕₂ : ...
  NonRec-⊗  : ...
```

Example:

```
Val-NonRec : ∀ (n : ℕ) → NonRec (K ℕ ⊕ (I ⊗ I)) (inj₁ n)
Val-NonRec : n = NonRec-⊕₁ (K ℕ) (I ⊗ I) n (NonRec-K ℕ n)
```

Generic leaves

There are two levels of **recursion** in a generic tree

- Functor: composition of functors $R \otimes Q$
- Fixed point: $[\![\, R \,]\!]\ (\mu\ R)$

```
data NonRec : (R : Reg) → ⟦ R ⟧ X → Set where
  NonRec-𝟙   : NonRec 𝟙 tt
  NonRec-K   : (B : Set) → (b : B) → NonRec (K B) b
  NonRec-⊕₁  : (R Q : Reg) → (r : ⟦ R ⟧ X)
             → NonRec R r → NonRec (R ⊕ Q) (inj₁ r)
  NonRec-⊕₂  : ...
  NonRec-⊗   : ...
```

Example:

```
Val-NonRec : ∀ (n : ℕ) → NonRec (K ℕ ⊕ (I ⊗ I)) (inj₁ n)
Val-NonRec : n = NonRec-⊕₁ (K ℕ) (I ⊗ I) n (NonRec-K ℕ n)
```

Generic leaves

There are two levels of **recursion** in a generic tree

- Functor: composition of functors $R \otimes Q$
- Fixed point: $[\![R]\!] \, (\mu\ R)$

$$
\begin{array}{lll}
\textsf{data NonRec} & : & (R : \textsf{Reg}) \to [\![R]\!]\ X \to \textsf{Set where} \\
\quad \textsf{NonRec-}\mathbb{1} & : & \textsf{NonRec}\ \mathbb{1}\ tt \\
\quad \textsf{NonRec-K} & : & (B : \textsf{Set}) \to (b : B) \to \textsf{NonRec}\ (\textsf{K}\ B)\ b \\
\quad \textsf{NonRec-}\oplus_1 & : & (R\ Q : \textsf{Reg}) \to (r : [\![R]\!]\ X) \\
 & & \to \textsf{NonRec}\ R\ r \to \textsf{NonRec}\ (R \oplus Q)\ (\textsf{inj}_1\ r) \\
\quad \textsf{NonRec-}\oplus_2 & : & \ldots \\
\quad \textsf{NonRec-}\otimes & : & \ldots
\end{array}
$$

Example:

$$
\begin{array}{lll}
\textsf{Val-NonRec} & : & \forall\ (n : \mathbb{N}) \to \textsf{NonRec}\ (\textsf{K}\ \mathbb{N} \oplus (\textsf{I} \otimes \textsf{I}))\ (\textsf{inj}_1\ n) \\
\textsf{Val-NonRec} & : & n = \textsf{NonRec-}\oplus_1\ (\textsf{K}\ \mathbb{N})\ (\textsf{I} \otimes \textsf{I})\ n\ (\textsf{NonRec-K}\ \mathbb{N}\ n)
\end{array}
$$

Generic leaves

There are two levels of **recursion** in a generic tree

- Functor: composition of functors $R \otimes Q$
- Fixed point: $[\![R]\!] (\mu R)$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{data NonRec} : (R : \text{Reg}) \to [\![R]\!]\, X \to \text{Set where}} \\
\quad \text{NonRec-}\mathbb{1} & : & \text{NonRec}\ \mathbb{1}\ tt \\
\quad \text{NonRec-K} & : & (B : \text{Set}) \to (b : B) \to \text{NonRec}\ (\text{K}\ B)\ b \\
\quad \text{NonRec-}\oplus_1 & : & (R\ Q : \text{Reg}) \to (r : [\![R]\!]\, X) \\
 & & \to \text{NonRec}\ R\ r \to \text{NonRec}\ (R \oplus Q)\ (\text{inj}_1\ r) \\
\quad \text{NonRec-}\oplus_2 & : & \ldots \\
\quad \text{NonRec-}\otimes & : & \ldots
\end{array}
$$

Example:

$$
\begin{array}{lll}
\text{Val-NonRec} & : & \forall\, (n : \mathbb{N}) \to \text{NonRec}\ (\text{K}\ \mathbb{N} \oplus (\text{I} \otimes \text{I}))\ (\text{inj}_1\ n) \\
\text{Val-NonRec} & : & n = \text{NonRec-}\oplus_1\ (\text{K}\ \mathbb{N})\ (\text{I} \otimes \text{I})\ n\ (\text{NonRec-K}\ \mathbb{N}\ n)
\end{array}
$$

Generic configurations

- Generic configuration = **leaf** + **stack**

$$\begin{aligned}\mathsf{Config}^G &: (R : \mathsf{Reg}) \to (X : \mathsf{Set}) \\ &\quad \to (\psi : [\![R]\!]\, X \to X) \to \mathsf{Set} \\ \mathsf{Config}^G\ R\ X\ \psi &= \Sigma\ ([\![R]\!]\ X)\ (\mathsf{NonRec}\ R) \times \mathsf{Stack}^G\ R\ X\ \psi\end{aligned}$$

- *Embed* a leaf into a generic tree

$$\mathsf{coerce} : (R : \mathsf{Reg}) \to (x : [\![R]\!]\ X) \to \mathsf{NonRec}\ R\ x \to [\![R]\!]\ Y$$

- Recover the **input** tree

$$\begin{aligned}\mathsf{plugC}\text{-}\mu_{\Downarrow} &: (R : \mathsf{Reg})\ \{\psi : [\![R]\!]\ X \to X\} \\ &\quad \to \mathsf{Config}^G\ R\ X\ \psi \to \mu\ R \to \mathsf{Set} \\ \mathsf{plugC}\text{-}\mu_{\Downarrow}\ R\ ((l\ , isl)\ , s)\ t &= \mathsf{plug}\text{-}\mu_{\Downarrow}\ R\ (\mathsf{In}\ (\mathsf{coerce}\ l\ isl))\ s\ t\end{aligned}$$

Generic configurations

- Generic configuration = **leaf** + **stack**

$$\begin{aligned}\mathsf{Config}^G &: (R : \mathsf{Reg}) \to (X : \mathsf{Set}) \\ &\to (\psi : [\![R]\!]\, X \to X) \to \mathsf{Set} \\ \mathsf{Config}^G\ R\ X\ \psi &= \Sigma\ ([\![R]\!]\, X)\ (\mathsf{NonRec}\ R) \times \mathsf{Stack}^G\ R\ X\ \psi\end{aligned}$$

- *Embed* a leaf into a generic tree

$$\mathsf{coerce} : (R : \mathsf{Reg}) \to (x : [\![R]\!]\, X) \to \mathsf{NonRec}\ R\ x \to [\![R]\!]\, Y$$

- Recover the **input** tree

$$\begin{aligned}\mathsf{plugC}\text{-}\mu_{\Downarrow} &: (R : \mathsf{Reg})\ \{\psi : [\![R]\!]\, X \to X\} \\ &\to \mathsf{Config}^G\ R\ X\ \psi \to \mu\ R \to \mathsf{Set} \\ \mathsf{plugC}\text{-}\mu_{\Downarrow}\ R\ ((l\ ,\ isl)\ ,\ s)\ t &= \mathsf{plug}\text{-}\mu_{\Downarrow}\ R\ (\mathsf{In}\ (\mathsf{coerce}\ l\ isl))\ s\ t\end{aligned}$$

Generic configurations

- Generic configuration = **leaf** + **stack**

$$
\begin{aligned}
\mathsf{Config}^G &: (R : \mathsf{Reg}) \to (X : \mathsf{Set}) \\
&\to (\psi : [\![R]\!]\, X \to X) \to \mathsf{Set} \\
\mathsf{Config}^G\; R\; X\; \psi &= \Sigma\, ([\![R]\!]\, X)\, (\mathsf{NonRec}\; R) \times \mathsf{Stack}^G\; R\; X\; \psi
\end{aligned}
$$

- *Embed* a leaf into a generic tree

$$
\mathsf{coerce} : (R : \mathsf{Reg}) \to (x : [\![R]\!]\, X) \to \mathsf{NonRec}\; R\; x \to [\![R]\!]\, Y
$$

- Recover the **input** tree

$$
\begin{aligned}
\mathsf{plugC}\text{-}\mu_{\Downarrow} &: (R : \mathsf{Reg})\, \{\psi : [\![R]\!]\, X \to X\} \\
&\to \mathsf{Config}^G\; R\; X\; \psi \to \mu\; R \to \mathsf{Set} \\
\mathsf{plugC}\text{-}\mu_{\Downarrow}\; R\; ((l\, ,\, isl)\, ,\, s)\; t &= \mathsf{plug}\text{-}\mu_{\Downarrow}\; R\; (\mathsf{In}\; (\mathsf{coerce}\; l\; isl))\; s\; t
\end{aligned}
$$

One step of a catamorphism

Generic unload^G

$$
\begin{aligned}
&\mathsf{unload}^G : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \\
&\quad \to (t : \mu\, R) \to (x : X) \to \mathsf{cata}\; R\; \psi\; t \equiv x \\
&\quad \to \mathsf{Stack}^G\; R\; X\; \psi \to \mathsf{Config}^G\; R\; X\; \psi \uplus X \\
&\mathsf{unload}^G\; R\; \psi\; t\; x\; eq\; [\,] = \mathsf{inj}_2\; x \\
&\mathsf{unload}^G\; R\; \psi\; t\; x\; eq\; (h :: hs)\ \mathsf{with}\ \mathsf{right}\; R\; h\; (t , x , eq) \\
&\mathsf{unload}^G\; R\; \psi\; t\; x\; eq\; (h :: hs) \mid \mathsf{inj}_1\; r\ \mathsf{with}\ \mathsf{compute}\; R\; r \\
&\ldots \mid (rx , rr) , eq' = \mathsf{unload}^G\; R\; \psi\; (\mathsf{In}\; rp)\; (\psi\; rx)\; eq'\; hs \\
&\mathsf{unload}^G\; R\; \psi\; t\; x\; eq\; (h :: hs) \mid \mathsf{inj}_2\; (dr , q) \\
&\qquad\qquad\qquad = \mathsf{load}^G\; R\; q\; (dr :: hs)
\end{aligned}
$$

$$
\mathsf{right} : (R : \mathsf{Reg}) \to \nabla\; R\; X\; Y \to X \to [\![R]\!]\, X \uplus (\nabla\; R\; X\; Y \times Y)
$$

$$
\begin{aligned}
&\mathsf{compute} : (R : \mathsf{Reg})\; \{\psi : [\![R]\!]\, X \to X\} \\
&\quad \to [\![R]\!]\, (\mathsf{Computed}\; R\; X\; \psi) \\
&\quad \to \Sigma\, ([\![R]\!]\, X \times [\![R]\!]\, (\mu\, R))\; \lambda\, \{(r , t) \to \mathsf{cata}\; R\; \psi\; (\mathsf{In}\; t) \equiv \psi\; r\}
\end{aligned}
$$

One step of a catamorphism

Generic unload^G

$$
\begin{aligned}
&\text{unload}^G : (R : \text{Reg}) \to (\psi : [\![R]\!]\, X \to X) \\
&\quad \to (t : \mu\, R) \to (x : X) \to \text{cata}\ R\ \psi\ t \equiv x \\
&\quad \to \text{Stack}^G\ R\ X\ \psi \to \text{Config}^G\ R\ X\ \psi \uplus X \\
&\text{unload}^G\ R\ \psi\ t\ x\ eq\ [] = \text{inj}_2\ x \\
&\text{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs)\ \text{with right}\ R\ h\ (t , x , eq) \\
&\text{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs)\ \mid \text{inj}_1\ r\ \text{with compute}\ R\ r \\
&\ldots \mid (rx , rr) , eq' = \text{unload}^G\ R\ \psi\ (\text{In}\ rp)\ (\psi\ rx)\ eq'\ hs \\
&\text{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs)\ \mid \text{inj}_2\ (dr , q) \\
&\qquad\qquad = \text{load}^G\ R\ q\ (dr :: hs)
\end{aligned}
$$

$$
\text{right} : (R : \text{Reg}) \to \nabla\ R\ X\ Y \to X \to [\![R]\!]\, X \uplus (\nabla\ R\ X\ Y \times Y)
$$

$$
\begin{aligned}
&\text{compute} : (R : \text{Reg})\ \{\psi : [\![R]\!]\, X \to X\} \\
&\quad \to [\![R]\!]\, (\text{Computed}\ R\ X\ \psi) \\
&\quad \to \Sigma\, ([\![R]\!]\, X \times [\![R]\!]\, (\mu\ R))\ \lambda\, \{(r , t) \to \text{cata}\ R\ \psi\ (\text{In}\ t) \equiv \psi\ r\}
\end{aligned}
$$

One step of a catamorphism

Generic unload^G

$$
\begin{aligned}
&\mathsf{unload}^G : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X)\\
&\quad \to (t : \mu\, R) \to (x : X) \to \mathsf{cata}\ R\ \psi\ t \equiv x\\
&\quad \to \mathsf{Stack}^G\ R\ X\ \psi \to \mathsf{Config}^G\ R\ X\ \psi \uplus X\\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ [\,] = \mathsf{inj}_2\ x\\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs)\ \mathsf{with}\ \mathsf{right}\ R\ h\ (t , x , eq)\\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs) \mid \mathsf{inj}_1\ r\ \mathsf{with}\ \mathsf{compute}\ R\ r\\
&\ldots \mid (rx , rr) , eq' = \mathsf{unload}^G\ R\ \psi\ (\mathsf{In}\ rp)\ (\psi\ rx)\ eq'\ hs\\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs) \mid \mathsf{inj}_2\ (dr , q)\\
&\qquad\qquad\qquad\qquad = \mathsf{load}^G\ R\ q\ (dr :: hs)
\end{aligned}
$$

$$
\mathsf{right} : (R : \mathsf{Reg}) \to \nabla\ R\ X\ Y \to X \to [\![R]\!]\, X \uplus (\nabla\ R\ X\ Y \times Y)
$$

$$
\begin{aligned}
&\mathsf{compute} : (R : \mathsf{Reg})\ \{\psi : [\![R]\!]\, X \to X\}\\
&\quad \to [\![R]\!]\ (\mathsf{Computed}\ R\ X\ \psi)\\
&\quad \to \Sigma\ ([\![R]\!]\, X \times [\![R]\!]\ (\mu\, R))\ \lambda\ \{(r , t) \to \mathsf{cata}\ R\ \psi\ (\mathsf{In}\ t) \equiv \psi\ r\}
\end{aligned}
$$

One step of a catamorphism

Generic unload^G

$$
\begin{aligned}
&\mathsf{unload}^G : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \\
&\quad \to (t : \mu\, R) \to (x : X) \to \mathsf{cata}\ R\ \psi\ t \equiv x \\
&\quad \to \mathsf{Stack}^G\ R\ X\ \psi \to \mathsf{Config}^G\ R\ X\ \psi \uplus X \\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ [] = \mathsf{inj}_2\ x \\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs)\ \mathsf{with}\ \mathsf{right}\ R\ h\ (t , x , eq) \\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs)\ \mid\ \mathsf{inj}_1\ r\ \mathsf{with}\ \mathsf{compute}\ R\ r \\
&\ldots \mid (rx , rr) , eq' = \mathsf{unload}^G\ R\ \psi\ (\mathsf{In}\ rp)\ (\psi\ rx)\ eq'\ hs \\
&\mathsf{unload}^G\ R\ \psi\ t\ x\ eq\ (h :: hs)\ \mid\ \mathsf{inj}_2\ (dr , q) \\
&\qquad\qquad\qquad = \mathsf{load}^G\ R\ q\ (dr :: hs)
\end{aligned}
$$

$$
\mathsf{right} : (R : \mathsf{Reg}) \to \nabla\ R\ X\ Y \to X \to [\![R]\!]\, X \uplus (\nabla\ R\ X\ Y \times Y)
$$

$$
\begin{aligned}
&\mathsf{compute} : (R : \mathsf{Reg})\ \{\psi : [\![R]\!]\, X \to X\} \\
&\quad \to [\![R]\!]\, (\mathsf{Computed}\ R\ X\ \psi) \\
&\quad \to \Sigma\ ([\![R]\!]\, X \times [\![R]\!]\, (\mu\, R))\ \lambda\ \{(r , t) \to \mathsf{cata}\ R\ \psi\ (\mathsf{In}\ t) \equiv \psi\ r\}
\end{aligned}
$$

A well-founded generic relation

The two levels of recursion induce two relations

A well-founded generic relation

The two levels of recursion induce two relations

- Functor

$$
\begin{array}{lll}
\text{data } \llcorner\_\lrcorner\_<_{\nabla}\_ & : & (R : \mathsf{Reg}) \\
 & & \rightarrow \nabla\ R\ X\ Y \times Y \rightarrow \nabla\ R\ X\ Y \times Y \rightarrow \mathsf{Set} \text{ where} \\
\mathsf{step}\text{-}\oplus_1 & : & \llcorner R \lrcorner\ (r , t_1) <_{\nabla} (r' , t_2) \\
 & \rightarrow & \llcorner R \oplus Q \lrcorner\ (\mathsf{inj}_1\ r , t_1) <_{\nabla} (\mathsf{inj}_1\ r' , t_2) \\
\mathsf{step}\text{-}\oplus_2 & : & \ldots \\
\mathsf{step}\text{-}\otimes_1 & : & \llcorner R \lrcorner\ (dr , t_1) <_{\nabla} (dr' , t_2) \\
 & \rightarrow & \llcorner R \otimes Q \lrcorner\ (\mathsf{inj}_1\ (dr , q) , t_1) <_{\nabla} (\mathsf{inj}_1\ (dr' , q) , t_2) \\
\mathsf{step}\text{-}\otimes_2 & : & \ldots \\
\mathsf{base}\text{-}\otimes & : & \llcorner R \otimes Q \lrcorner\ (\mathsf{inj}_2\ (r , dq) , t_1) <_{\nabla} (\mathsf{inj}_1\ (dr , q) , t_2)
\end{array}
$$

A well-founded generic relation

The <u>two</u> levels of recursion induce <u>two</u> relations

- Fixed point

$$
\begin{aligned}
&\text{data } \_<_{\mathbb{C}}\_ \;:\; \mathsf{Config}^{G}\; R\; X\; \psi \to \mathsf{Config}^{G}\; R\; X\; \psi \to \mathsf{Set} \text{ where}\\
&\quad \mathsf{Step} : \quad (t_1\;,\; s_1) <_{\mathbb{C}} (t_2\;,\; s_2)\\
&\qquad\qquad \to (t_1\;,\; h \;::\; s_1) <_{\mathbb{C}} (t_2\;,\; h \;::\; s_2)\\
&\quad \mathsf{Base} : \quad \mathsf{plugC}\text{-}\mu_{\Downarrow}\; R\; (t_1\;,\; s_1) \equiv e_1\\
&\qquad\qquad \to \mathsf{plugC}\text{-}\mu_{\Downarrow}\; R\; (t_2\;,\; s_2) \equiv e_2\\
&\qquad\qquad \to (h_1\;,\; e_1) <_{\nabla} (h_2\;,\; e_2)\\
&\qquad\qquad \to (t_1\;,\; h_1 \;::\; s_1) <_{\mathbb{C}} (t_2\;,\; h_2 \;::\; s_2)
\end{aligned}
$$

A well-founded generic relation (2)

- Type-indexed relation over $\mathsf{Config}^{G}_{\Downarrow}$

  $$\begin{aligned}&\text{data} \llcorner\_\lrcorner\llcorner\_\lrcorner\_<_{\mathbb{C}_\Downarrow}\_ \; \{X : \mathsf{Set}\}\;(R : \mathsf{Reg})\;\{\psi : [\![R]\!]\, X \to X\}\\&\quad : (t : \mu\, R)\\&\quad \to \mathsf{Config}^{G}_{\Downarrow}\; R\; X\; \psi\; t \to \mathsf{Config}^{G}_{\Downarrow}\; R\; X\; \psi\; t \to \mathsf{Set} \text{ where}\end{aligned}$$

- The relation is **well-founded**

  $$\begin{aligned}&<_{\mathbb{C}}\text{-WF} : \forall\,(R : \mathsf{Reg}) \to (t : \mu\, R)\\&\quad \to \mathsf{Well\text{-}founded}\;(\llcorner R \lrcorner\llcorner t \lrcorner\_<_{\mathbb{C}_\Downarrow}\_)\end{aligned}$$

A well-founded generic relation (2)

- Type-indexed relation over $\mathsf{Config}^{G}_{\Downarrow}$

$$
\begin{array}{l}
\mathsf{data}\ \llcorner\_\lrcorner\llcorner\_\lrcorner\_<_{\mathbb{C}_{\Downarrow}}\_\ \{X : \mathsf{Set}\}\ (R : \mathsf{Reg})\ \{\psi : [\![R]\!]\ X \to X\} \\
\quad : (t : \mu\ R) \\
\quad \to \mathsf{Config}^{G}_{\Downarrow}\ R\ X\ \psi\ t \to \mathsf{Config}^{G}_{\Downarrow}\ R\ X\ \psi\ t \to \mathsf{Set}\ \mathsf{where}
\end{array}
$$

- The relation is **well-founded**

$$
\begin{array}{l}
<_{\mathbb{C}}\text{-WF} : \forall\ (R : \mathsf{Reg}) \to (t : \mu\ R) \\
\quad \to \mathsf{Well\text{-}founded}\ (\llcorner\ R\ \lrcorner\llcorner\ t\ \lrcorner\_<_{\mathbb{C}_{\Downarrow}}\_)
\end{array}
$$

Invariant preserving step

- One step of the catamorphism

$$
\begin{aligned}
\mathsf{step}^G \;:\; & (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \to (t : \mu\, R) \\
& \to \mathsf{Config}^G_{\Uparrow}\; R\; X\; \psi\; t \to \mathsf{Config}^G_{\Uparrow}\; R\; X\; \psi\; t \uplus X
\end{aligned}
$$

- step^G delivers a **smaller** configuration

$$
\begin{aligned}
\mathsf{step}^G\text{-}\!< \;:\; & (R : \mathsf{Reg})\; (\psi : [\![R]\!]\, X \to X) \to (t : \mu\, R) \\
& \to (c_1\; c_2 : \mathsf{Config}^G_{\Uparrow}\; R\; X\; \psi\; t) \\
& \to \mathsf{step}^G\; R\; \psi\; t\; c_1 \equiv \mathsf{inj}_1\; c_2 \to \llcorner R \lrcorner \llcorner t \lrcorner\; c_2 \;\_\!<_{\mathbb{C}}\!\_\; c_1
\end{aligned}
$$

Invariant preserving step

- One step of the catamorphism

$$\mathsf{step}^G : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \to (t : \mu\, R) \to \mathsf{Config}^G_{\Uparrow}\, R\, X\, \psi\, t \to \mathsf{Config}^G_{\Uparrow}\, R\, X\, \psi\, t \uplus X$$

- step^G delivers a **smaller** configuration

$$\begin{aligned}\mathsf{step}^G\text{-}{<} : (R : \mathsf{Reg})\, (\psi : [\![R]\!]\, X \to X) \to (t : \mu\, R) \\ \to (c_1\ c_2 : \mathsf{Config}^G_{\Uparrow}\, R\, X\, \psi\, t) \\ \to \mathsf{step}^G\, R\, \psi\, t\, c_1 \equiv \mathsf{inj}_1\, c_2 \to \llcorner R \lrcorner\llcorner t \lrcorner\, c_2 \ \_{<_{\mathbb{C}}}\_\ c_1\end{aligned}$$

A generic tail-recursive evaluator

- Auxiliary recursor

$$
\begin{array}{l}
\mathsf{rec} : (R : \mathsf{Reg})\ (\psi : [\![R]\!]\ X \to X)\ (t : \mu\ R) \\
\quad \to (c : \mathsf{Config}^G_{\Uparrow}\ R\ X\ \psi\ t) \\
\quad \to \mathsf{Acc}\ (\llcorner R \lrcorner\llcorner t \lrcorner\_ <_{\mathbb{C}_{\Downarrow}} \_)\ (\mathsf{Config}^G_{\Uparrow}\text{-to-}\mathsf{Config}^G_{\Downarrow}\ c) \to X \\
\mathsf{rec}\ R\ \psi\ t\ c\ (\mathsf{acc}\ rs)\ \text{with}\ \mathsf{step}^G\ R\ \psi\ t\ c \mid \mathit{inspect}\ (\mathsf{step}^G\ R\ \psi\ t)\ c \\
\ldots \mid \mathsf{inj}_1\ x \mid [\ ls\ ] = \mathsf{rec}\ R\ \psi\ t\ x\ (rs\ x\ (\mathsf{step}^G\text{-}{<}\ R\ \psi\ t\ c\ x\ ls)) \\
\ldots \mid \mathsf{inj}_2\ y \mid [\ \_\ ] = y
\end{array}
$$

- Tail-recursive evaluator

$$
\begin{array}{l}
\mathsf{tail\text{-}rec\text{-}cata} : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\ X \to X) \to \mu\ R \to X \\
\mathsf{tail\text{-}rec\text{-}cata}\ R\ \psi\ x\ \text{with}\ \mathsf{load}^G\ R\ \psi\ x\ [] \\
\ldots \mid \mathsf{inj}_1\ c = \mathsf{rec}\ R\ \psi\ (c\ , \ldots)\ ({<_{\mathbb{C}}}\text{-WF}\ R\ c)
\end{array}
$$

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

A generic tail-recursive evaluator

- Auxiliary recursor

$$
\begin{aligned}
&\mathsf{rec} : (R : \mathsf{Reg})\ (\psi : [\![\ R\ ]\!]\ X \to X)\ (t : \mu\ R)\\
&\quad \to (c : \mathsf{Config}^{G}_{\Uparrow}\ R\ X\ \psi\ t)\\
&\quad \to \mathsf{Acc}\ (\llcorner\ R \lrcorner \llcorner\ t \lrcorner \_ <_{\mathbb{C}_{\Downarrow}} \_)\ (\mathsf{Config}^{G}_{\Uparrow}\text{-to-}\mathsf{Config}^{G}_{\Downarrow}\ c) \to X\\
&\mathsf{rec}\ R\ \psi\ t\ c\ (\mathsf{acc}\ rs)\ \text{with}\ \mathsf{step}^{G}\ R\ \psi\ t\ c \mid \mathit{inspect}\ (\mathsf{step}^{G}\ R\ \psi\ t)\ c\\
&\ldots \mid \mathsf{inj}_1\ x \mid [\ ls\ ] = \mathsf{rec}\ R\ \psi\ t\ x\ (rs\ x\ (\mathsf{step}^{G}\text{-}<\ R\ \psi\ t\ c\ x\ ls))\\
&\ldots \mid \mathsf{inj}_2\ y \mid [\ \_\ ] = y
\end{aligned}
$$

- Tail-recursive evaluator

$$
\begin{aligned}
&\mathsf{tail\text{-}rec\text{-}cata} : (R : \mathsf{Reg}) \to (\psi : [\![\ R\ ]\!]\ X \to X) \to \mu\ R \to X\\
&\mathsf{tail\text{-}rec\text{-}cata}\ R\ \psi\ x\ \text{with}\ \mathsf{load}^{G}\ R\ \psi\ x\ []\\
&\ldots \mid \mathsf{inj}_1\ c = \mathsf{rec}\ R\ \psi\ (c, \ldots)\ (<_{\mathbb{C}}\text{-WF}\ R\ c)
\end{aligned}
$$

Correctness, generically

- The correctness proof follows the same pattern as in tail-rec-eval
- Induction over Acc and an auxiliary lemma step^G-correct

$$\begin{aligned}\text{correctness}^G &: \forall\, (R : \text{Reg})\, (\psi : [\![\, R \,]\!]\, X \to X)\, (t : \mu\, R) \\ &\to \text{cata}\; R\; \psi\; t \equiv \text{tail-rec-cata}\; R\; \psi\; t\end{aligned}$$

Correctness, generically

- The correctness proof follows the same pattern as in tail-rec-eval
- Induction over Acc and an auxiliary lemma step^G-correct

$$\begin{aligned}\text{correctness}^G &: \forall\,(R : \text{Reg})\;(\psi : [\![\, R \,]\!]\; X \to X)\;(t : \mu\, R)\\ &\to \text{cata}\; R\; \psi\; t \equiv \text{tail-rec-cata}\; R\; \psi\; t\end{aligned}$$

Correctness, generically

- The correctness proof follows the same pattern as in tail-rec-eval
- Induction over Acc and an auxiliary lemma step^G-correct

$$
\begin{aligned}
\text{correctness}^G\ &: \forall\,(R : \text{Reg})\,(\psi : [\![\, R \,]\!]\ X \to X)\,(t : \mu\, R)\\
&\to \text{cata}\ R\ \psi\ t \equiv \text{tail-rec-cata}\ R\ \psi\ t
\end{aligned}
$$

5. Discussion

Discussion

- Runtime impact of storing proofs and subtrees
- Regular universe is limited
- Directly executable machine in comparison with other techniques

Discussion

- Runtime impact of storing proofs and subtrees
- Regular universe is limited
- Directly executable machine in comparison with other techniques

Discussion

- Runtime impact of storing proofs and subtrees
- Regular universe is limited
- Directly executable machine in comparison with other techniques

6. Conclusions

Conclusions

- We have developed a **tail-recursive** evaluator for Expr
- We generalized it for any algebra over any **regular** datatype
- We proved the evaluators to be **terminating** and **correct**

Conclusions

- We have developed a **tail-recursive** evaluator for Expr
- We generalized it for any algebra over any **regular** datatype
- We proved the evaluators to be **terminating** and **correct**

Conclusions

- We have developed a **tail-recursive** evaluator for Expr
- We generalized it for any algebra over any **regular** datatype
- We proved the evaluators to be **terminating** and **correct**

Future work

- More expressive universes
- Other theorem provers
- Long-term goal: abstract machine for λ-calculus

Future work

- More expressive universes
- Other theorem provers
- Long-term goal: abstract machine for λ-calculus

Future work

- More expressive universes
- Other theorem provers
- Long-term goal: abstract machine for λ-calculus

Thank you very much for your attention!