Verified tail-recursive folds through dissection

Thesis defense

Carlos Tomé Cortiñas

July 14, 2018

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

1. Introduction

1.1 Motivation

Expression language[1]

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Expr} : \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{Val} : \mathbb{N} \rightarrow \mathsf{Expr} \\
\quad \mathsf{Add} : \mathsf{Expr} \rightarrow \mathsf{Expr} \rightarrow \mathsf{Expr}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{expr}_1 : \mathsf{Expr} \\
\mathsf{expr}_1 = \mathsf{Add}\ (\mathsf{Val}\ 1) \\
\qquad\qquad (\mathsf{Add}\ (\mathsf{Val}\ 2) \\
\qquad\qquad\qquad (\mathsf{Val}\ 2))
\end{array}
$$

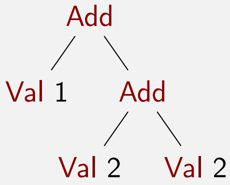
```
       Add
      /   \
  Val 1    Add
          /   \
      Val 2   Val 2
```

[1] Hutton's razor

Expression language[1]

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

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



[1] Hutton's razor

Evaluating Expressions

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

$$
\begin{aligned}
&\mathsf{prop}_1 : \mathsf{eval}\ \mathsf{expr}_1 \equiv 5\\
&\mathsf{prop}_1 = \mathsf{refl}
\end{aligned}
$$

Is there a **problem** with eval?

Evaluating Expressions

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

$$
\begin{aligned}
&\mathsf{prop}_1 : \mathsf{eval}\ \mathsf{expr}_1 \equiv 5 \\
&\mathsf{prop}_1 = \mathsf{refl}
\end{aligned}
$$

Is there a **problem** with eval?

Evaluating Expressions

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

$$
\begin{aligned}
&\mathsf{prop}_1 : \mathsf{eval}\ \mathsf{expr}_1 \equiv 5\\
&\mathsf{prop}_1 = \mathsf{refl}
\end{aligned}
$$

Is there a **problem** with eval?

Evaluating Expressions (2)

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

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

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

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)

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

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

Evaluating Expressions (2)

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

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

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

Evaluating Expressions (2)

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

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

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

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

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

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

Evaluating Expressions (3)

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

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

Yes, eval is not a *tail-recursive* function

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

A **well-typed** program *went wrong*[2]

[2]Robin Milner does not approve

Evaluating Expressions (3)

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

1 + (2 + (3 + (4 + …

Yes, eval <ins>is not</ins> a *tail-recursive* function

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

A **well-typed** program *went wrong*[2]

[2] Robin Milner does not approve

Evaluating Expressions (3)

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

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

Yes, eval is not a *tail-recursive* function

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

A **well-typed** program *went wrong*[2]

[2]Robin Milner does not approve

Evaluating Expressions (3)

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

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

Yes, eval is not a *tail-recursive* function

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

A **well-typed** program *went wrong*[2]

[2] Robin Milner does not approve

Evaluating Expressions (3)

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

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

Yes, eval is not a *tail-recursive* function

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

A **well-typed** program *went wrong*[2]

[2] Robin Milner does not approve

Evaluating Expressions (3)

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

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

Yes, eval is not a *tail-recursive* function

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

A **well-typed** program *went wrong*[2]

[2] Robin Milner does not approve

Can we solve the problem?

- 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
```

Can we solve the problem?

- 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
```

Can we solve the problem?

- 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
```

Can we solve the problem? (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
```

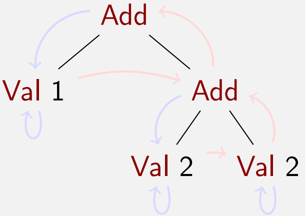
Can we solve the problem? (2)

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

$$
\begin{array}{l}
\mathsf{tail\text{-}rec\text{-}eval} : \mathsf{Expr} \to \mathbb{N} \\
\mathsf{tail\text{-}rec\text{-}eval}\ e = \mathsf{load}\ e\ \mathsf{Top}
\end{array}
$$

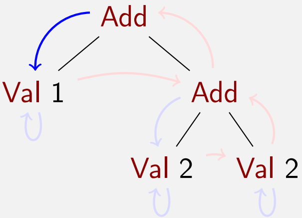
Can we solve the problem? (3)

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



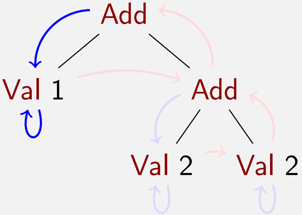
Can we solve the problem? (3)

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



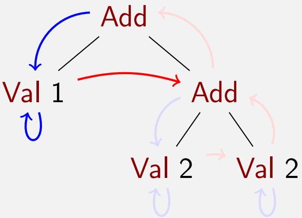
Can we solve the problem? (3)

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



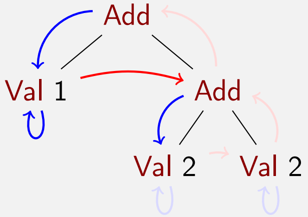
Can we solve the problem? (3)

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



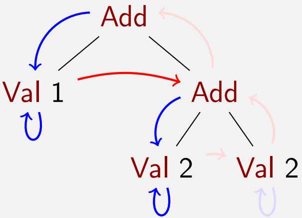
Can we solve the problem? (3)

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



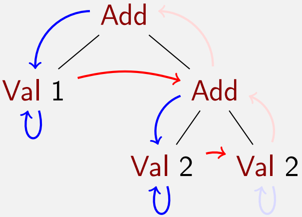
Can we solve the problem? (3)

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



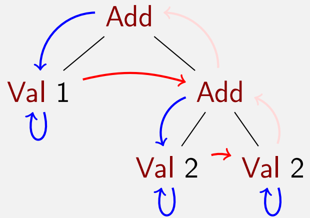
Can we solve the problem? (3)

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



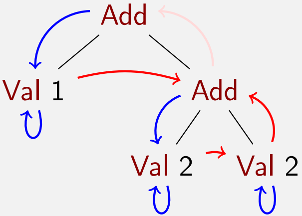
Can we solve the problem? (3)

unload 2 (Right 2 (Right 1 Top))



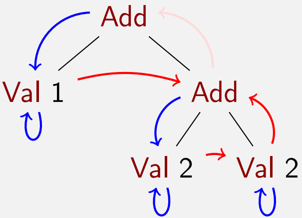
Can we solve the problem? (3)

unload (2 + 2) (Right 1 Top)



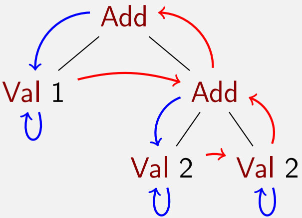
Can we solve the problem? (3)

unload 4 (Right 1 Top)



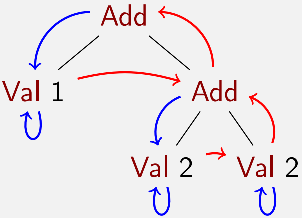
Can we solve the problem? (3)

unload (1 + 4) Top



Can we solve the problem? (3)

unload 5 Top



Can we solve the problem? (4)

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!

Can we solve the problem? (4)

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!

Can we solve the problem? (4)

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!

Can we solve the problem? (4)

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!

Can we solve the problem? (4)

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!

Can we solve the problem? (5)

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

mutual

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

Can we solve the problem? (5)

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

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

Can we solve the problem? (5)

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

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

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

3. A tail-recursive evaluator for Expr

3.1 Problem with tail-rec-eval

Problematic call

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

Traversing the Expr

- load and unload traverse the Expr left to right
- The functions use the Stack only to store subexpressions of the input expression
- How do we convince **Agda** of this fact?

Traversing the Expr

- load and unload traverse the Expr left to right
- The functions use the Stack only to store subexpressions of the input expression
- How do we convince **Agda** of this fact?

Traversing the Expr

- load and unload traverse the Expr left to right
- The functions use the Stack only to store subexpressions of the input expression
- How do we convince **Agda** of this fact?

Traversing the Expr (2)

- We rewrite load and unload to compute **one** step of the evaluation

$$
\begin{aligned}
&\mathsf{load} : \mathsf{Expr} \to \mathsf{Stack} \to (\mathbb{N} \times \mathsf{Stack}) \uplus \mathbb{N}\\
&\mathsf{load}\ (\mathsf{Val}\ n)\ \mathit{stk} = \mathsf{inj}_1\ (n\ ,\ \mathit{stk})\\
&\mathsf{load}\ (\mathsf{Add}\ e_1\ e_2)\ \mathit{stk} = \mathsf{load}\ e_1\ (\mathsf{Left}\ e_2\ \mathit{stk})\\
&\mathsf{unload} : \mathbb{N} \to \mathsf{Stack} \to (\mathbb{N} \times \mathsf{Stack}) \uplus \mathbb{N}\\
&\mathsf{unload}\ v\ \mathsf{Top} = \mathsf{inj}_2\ v\\
&\mathsf{unload}\ v\ (\mathsf{Right}\ v'\ \mathit{stk}) = \mathsf{unload}\ (v' + v)\ \mathit{stk}\\
&\mathsf{unload}\ v\ (\mathsf{Left}\ e\ \mathit{stk}) = \mathsf{load}\ e\ (\mathsf{Right}\ v\ \mathit{stk})
\end{aligned}
$$

Traversing the Expr (2)

- We rewrite load and unload to compute **one** step of the evaluation

$$
\begin{aligned}
&\mathsf{load} : \mathsf{Expr} \to \mathsf{Stack} \to (\mathbb{N} \times \mathsf{Stack}) \uplus \mathbb{N} \\
&\mathsf{load}\ (\mathsf{Val}\ n)\ \ \mathit{stk} = \mathsf{inj}_1\ (n\ ,\ \mathit{stk}) \\
&\mathsf{load}\ (\mathsf{Add}\ e_1\ e_2)\ \mathit{stk} = \mathsf{load}\ e_1\ (\mathsf{Left}\ e_2\ \mathit{stk}) \\
&\mathsf{unload} : \mathbb{N} \to \mathsf{Stack} \to (\mathbb{N} \times \mathsf{Stack}) \uplus \mathbb{N} \\
&\mathsf{unload}\ v\ \mathsf{Top} = \mathsf{inj}_2\ v \\
&\mathsf{unload}\ v\ (\mathsf{Right}\ v'\ \mathit{stk}) = \mathsf{unload}\ (v' + v)\ \mathit{stk} \\
&\mathsf{unload}\ v\ (\mathsf{Left}\ e\ \mathit{stk}) = \mathsf{load}\ e\ (\mathsf{Right}\ v\ \mathit{stk})
\end{aligned}
$$

4. A generic tail-recursive evaluator

5. Conclusions