

Verified tail-recursive folds through dissection

Carlos Tomé Cortiñas

9 November 2018



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

To put it another way ...

From Algebra to Abstract Machine: A Verified Generic Construction

TyDe'18

Carlos Tomé Cortiñas and Wouter Swierstra



Motivation

Contributions

Solving the problem

Generalization

Conclusion



We start with a small Expression language



We start with a small Expression language

data Expr : Set where

Val : \mathbb{N} \rightarrow Expr

Add : Expr \rightarrow Expr \rightarrow Expr



We start with a small **Expression** language

data **Expr** : **Set** where

Val : \mathbb{N} \rightarrow **Expr**

Add : **Expr** \rightarrow **Expr** \rightarrow **Expr**

and we write an **evaluator** for it.



We start with a small Expression language

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

and we write an evaluator for it.

```
eval : Expr → ℕ
eval (Val n)      = n
eval (Add e1 e2) = eval e1 + eval e2
```



```
> eval (Add (Add (Add ... (Add (Val 1) (Val 2))))) -- large expression
```




```
> eval (Add (Add (Add ... (Add (Val 1) (Val 2))))) -- large expression
```

```
* * * Exception : stack overflow
```



```
> eval (Add (Add (Add ... (Add (Val 1) (Val 2))))) -- large expression  
*** Exception : stack overflow
```

What happened?



```
> eval (Add (Add (Add ... (Add (Val 1) (Val 2))))) -- large expression  
*** Exception : stack overflow
```

What happened? A **well-typed** program *went wrong*.



```
> eval (Add (Add (Add ... (Add (Val 1) (Val 2))))) -- large expression  
*** Exception : stack overflow
```

What happened? A **well-typed** program *went wrong*.

- **eval** evaluates both subtrees before reducing `_+_` further.



```
> eval (Add (Add (Add ... (Add (Val 1) (Val 2))))) -- large expression  
*** Exception : stack overflow
```

What happened? A **well-typed** program *went wrong*.

- ▶ **eval** evaluates both subtrees before reducing `_+_` further.
- ▶ Record unevaluated subtrees on the stack.



```
> eval (Add (Add (Add ... (Add (Val 1) (Val 2)))))) -- large expression  
*** Exception : stack overflow
```

What happened? A **well-typed** program *went wrong*.

- ▶ **eval** evaluates both subtrees before reducing `_+_` further.
- ▶ Record unevaluated subtrees on the stack.
- ▶ On large inputs, the **stack** might **overflow**.



A solution to the problem

§1



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Write a **tail-recursive** evaluator.



Write a **tail-recursive** evaluator.

We proceed in *three steps*:



Write a **tail-recursive** evaluator.

We proceed in *three steps*:

- ▶ Make the underlying stack explicit.



Write a **tail-recursive** evaluator.

We proceed in *three steps*:

- ▶ Make the underlying stack explicit.
- ▶ Define a *tail-recursive* function over the stack.



Write a **tail-recursive** evaluator.

We proceed in *three steps*:

- ▶ Make the underlying stack explicit.
- ▶ Define a *tail-recursive* function over the stack.
- ▶ Show that it is equivalent to *eval*.



A tail-recursive evaluator

§1

data **Stack** : **Set** where

Top : **Stack**

Left : **Expr** \rightarrow **Stack** \rightarrow **Stack**

Right : **N** \rightarrow **Stack** \rightarrow **Stack**



data **Stack** : **Set** where

Top : **Stack**

Left : **Expr** \rightarrow **Stack** \rightarrow **Stack**

Right : \mathbb{N} \rightarrow **Stack** \rightarrow **Stack**

mutual

load : **Expr** \rightarrow **Stack** \rightarrow \mathbb{N}

load (**Val** n) stk = **unload** n stk

load (**Add** e_1 e_2) stk = **load** e_1 (**Left** e_2 stk)

unload : \mathbb{N} \rightarrow **Stack** \rightarrow \mathbb{N}

unload v **Top** = v

unload v (**Left** e stk) = **load** e (**Right** v stk)

unload v (**Right** v' stk) = **unload** ($v' + v$) stk



data **Stack** : **Set** where

Top : **Stack**

Left : **Expr** \rightarrow **Stack** \rightarrow **Stack**

Right : \mathbb{N} \rightarrow **Stack** \rightarrow **Stack**

mutual

load : **Expr** \rightarrow **Stack** \rightarrow \mathbb{N}

load (**Val** n) stk = **unload** n stk

load (**Add** e_1 e_2) stk = **load** e_1 (**Left** e_2 stk)

unload : \mathbb{N} \rightarrow **Stack** \rightarrow \mathbb{N}

unload v **Top** = v

unload v (**Left** e stk) = **load** e (**Right** v stk)

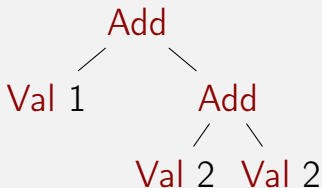
unload v (**Right** v' stk) = **unload** ($v' + v$) stk

tail-rec-eval : **Expr** \rightarrow \mathbb{N}

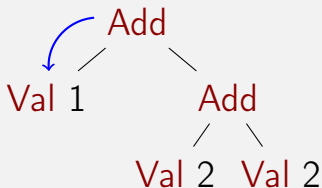
tail-rec-eval e = **load** e **Top**



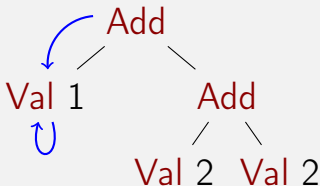
load (Add (Val 1) (Add (Val 2) (Val 2))) Top



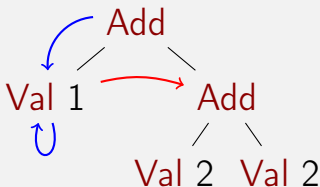
load (Val 1) (Left (Add (Val 2) (Val 2)) Top)



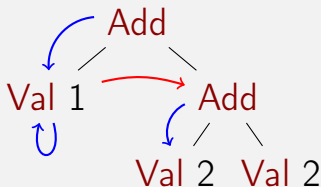
unload 1 (Left (Add (Val 2) (Val 2)) Top)



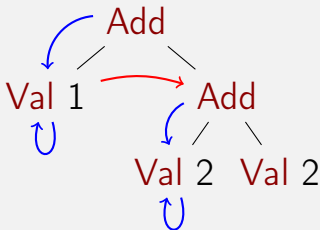
load (Add (Val 2) (Val 2)) (Right 1 Top)



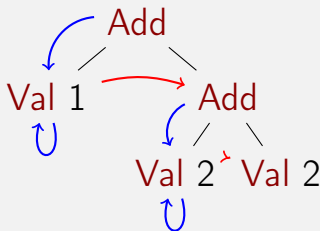
load (Val 2) (Left (Val 2) (Right 1 Top))



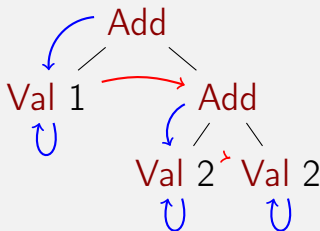
unload 2 (Left (Val 2) (Right 1 Top))



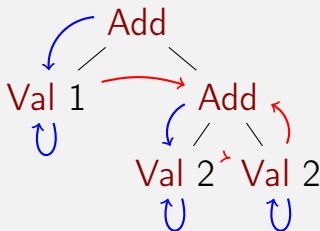
load (Val 2) (Right 2 (Right 1 Top))



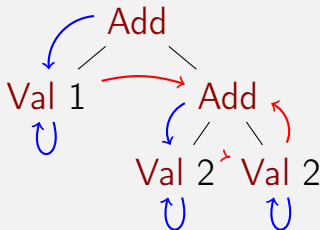
unload 2 (Right 2 (Right 1 Top))



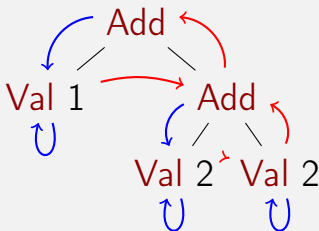
unload (2 + 2) (Right 1 Top)



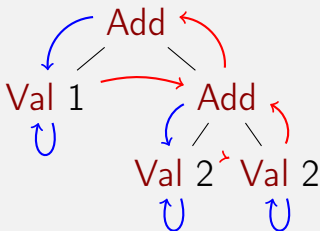
unload 4 (Right 1 Top)



unload (1 + 4) Top



unload 5 Top



Have we **actually** solved the problem?



Have we **actually** solved the problem?

Is really **tail-rec-eval** *equivalent* to **eval**?



Have we **actually** solved the problem?

Is really **tail-rec-eval** *equivalent* to **eval**? Can we prove it?



Have we **actually** solved the problem?

Is really **tail-rec-eval** *equivalent* to **eval**? Can we prove it?

mutual

load : Expr \rightarrow Stack \rightarrow \mathbb{N}

...

unload : $\mathbb{N} \rightarrow$ Stack \rightarrow \mathbb{N}

unload v Top = v

unload v (Left e stk) = **load** e (Right v stk)

unload v (Right v' stk) = **unload** ($v' + v$) stk



Have we **actually** solved the problem?

Is really **tail-rec-eval** *equivalent* to **eval**? Can we prove it?

mutual

load : Expr \rightarrow Stack \rightarrow \mathbb{N}

...

unload : $\mathbb{N} \rightarrow$ Stack \rightarrow \mathbb{N}

unload v Top = v

unload v (Left e stk) = **load** e (Right v stk)

unload v (Right v' stk) = **unload** ($v' + v$) stk

We **lost** termination guarantees.



Motivation
Contributions
Solving the problem
Generalization
Conclusion



Termination We **construct** a tail-recursive evaluator that terminates for every input.



Termination We **construct** a tail-recursive evaluator that terminates for every input.

Correctness We **prove** it equal to the original **eval** function.



`eval` is an example of a *fold* over the `Expr` datatype.



`eval` is an example of a *fold* over the `Expr` datatype.

$$\text{fold} : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \text{Expr} \rightarrow \mathbb{N}$$
$$\text{fold } \phi_1 \phi_2 (\text{Val } n) = \phi_2 n$$
$$\text{fold } \phi_1 \phi_2 (\text{Add } e_1 e_2) = \phi_1 (\text{fold } \phi_1 \phi_2 e_1) (\text{fold } \phi_1 \phi_2 e_2)$$


`eval` is an example of a *fold* over the `Expr` datatype.

$$\begin{aligned}\text{fold} &: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \text{Expr} \rightarrow \mathbb{N} \\ \text{fold } \phi_1 \phi_2 (\text{Val } n) &= \phi_2 n \\ \text{fold } \phi_1 \phi_2 (\text{Add } e_1 e_2) &= \phi_1 (\text{fold } \phi_1 \phi_2 e_1) (\text{fold } \phi_1 \phi_2 e_2)\end{aligned}$$

Both versions of the function are the same.



`eval` is an example of a *fold* over the `Expr` datatype.

$$\begin{aligned}\text{fold} &: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \text{Expr} \rightarrow \mathbb{N} \\ \text{fold } \phi_1 \phi_2 (\text{Val } n) &= \phi_2 n \\ \text{fold } \phi_1 \phi_2 (\text{Add } e_1 e_2) &= \phi_1 (\text{fold } \phi_1 \phi_2 e_1) (\text{fold } \phi_1 \phi_2 e_2)\end{aligned}$$

Both versions of the function are the same.

$$\forall (e : \text{Expr}) \rightarrow \text{eval } e \equiv \text{fold } _ + _ \text{ id } e$$


`eval` is an example of a *fold* over the `Expr` datatype.

$$\begin{aligned}\text{fold} &: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \text{Expr} \rightarrow \mathbb{N} \\ \text{fold } \phi_1 \phi_2 (\text{Val } n) &= \phi_2 n \\ \text{fold } \phi_1 \phi_2 (\text{Add } e_1 e_2) &= \phi_1 (\text{fold } \phi_1 \phi_2 e_1) (\text{fold } \phi_1 \phi_2 e_2)\end{aligned}$$

Both versions of the function are the same.

$$\forall (e : \text{Expr}) \rightarrow \text{eval } e \equiv \text{fold } _ + _ \text{ id } e$$

Moreover:



`eval` is an example of a *fold* over the `Expr` datatype.

$$\begin{aligned}\text{fold} &: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \text{Expr} \rightarrow \mathbb{N} \\ \text{fold } \phi_1 \phi_2 (\text{Val } n) &= \phi_2 n \\ \text{fold } \phi_1 \phi_2 (\text{Add } e_1 e_2) &= \phi_1 (\text{fold } \phi_1 \phi_2 e_1) (\text{fold } \phi_1 \phi_2 e_2)\end{aligned}$$

Both versions of the function are the same.

$$\forall (e : \text{Expr}) \rightarrow \text{eval } e \equiv \text{fold } _ + _ \text{ id } e$$

Moreover:

- `Expr` is an example of a **regular** type.



`eval` is an example of a *fold* over the `Expr` datatype.

$$\begin{aligned}\text{fold} &: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \text{Expr} \rightarrow \alpha \\ \text{fold } \phi_1 \phi_2 (\text{Val } n) &= \phi_2 n \\ \text{fold } \phi_1 \phi_2 (\text{Add } e_1 e_2) &= \phi_1 (\text{fold } \phi_1 \phi_2 e_1) (\text{fold } \phi_1 \phi_2 e_2)\end{aligned}$$

Both versions of the function are the same.

$$\forall (e : \text{Expr}) \rightarrow \text{eval } e \equiv \text{fold } _ + _ \text{ id } e$$

Moreover:

- ▶ `Expr` is an example of a **regular** type.
- ▶ `fold` is an instance of a **catamorphism**.



Termination We **construct** a tail-recursive evaluator that terminates for every input.

Correctness We **prove** it equal to the original **eval** function.



Termination We **construct** a tail-recursive evaluator that terminates for every input.

Correctness We **prove** it equal to the original **eval** function.

Generalization We generalize our result to any *catamorphism* and any *algebra* over any **regular** datatype.



Termination We **construct** a tail-recursive evaluator that terminates for every input.

Correctness We **prove** it equal to the original **eval** function.

Generalization We generalize our result to any *catamorphism* and any *algebra* over any **regular** datatype.

Everything is machine-checked by *Agda*.



Motivation
Contributions
Solving the problem
Generalization
Conclusion



1. Break down the mutual recursion in **load** and **unload**.



1. Break down the mutual recursion in **load** and **unload**.
2. Use **unload⁺** to iterate through an **Expr** to get a value.



1. Break down the mutual recursion in **load** and **unload**.
2. Use **unload**⁺ to iterate through an **Expr** to get a value.
3. Show that **unload**⁺ delivers something “smaller”, thus the iteration terminates.



1. Break down the mutual recursion in **load** and **unload**.
2. Use **unload**⁺ to iterate through an **Expr** to get a value.
3. Show that **unload**⁺ delivers something “smaller“, thus the iteration terminates.
4. Prove that the resulting function is equivalent to **eval**.



$$\begin{aligned}\text{unload}^+ &: (\mathbb{N} \times \text{Stack}) \rightarrow (\mathbb{N} \times \text{Stack}) \uplus \mathbb{N} \\ \text{unload}^+ (v, \text{Top}) &= \text{inj}_2 v \\ \text{unload}^+ (v, \text{Right } v' \text{ stk}) &= \text{unload}^+ (v' + v, \text{stk}) \\ \text{unload}^+ (v, \text{Left } e \text{ stk}) &= \text{load } e (\text{Right } v \text{ stk})\end{aligned}$$



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


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

Now, both functions **obviously** terminate.



$$\text{unload}^+ : (\mathbb{N} \times \text{Stack}) \rightarrow (\mathbb{N} \times \text{Stack}) \uplus \mathbb{N}$$



$$\text{unload}^+ : (\mathbb{N} \times \text{Stack}) \rightarrow (\mathbb{N} \times \text{Stack}) \uplus \mathbb{N}$$

- unload^+ is the “step” function of an abstract machine!



$$\text{unload}^+ : (\mathbb{N} \times \text{Stack}) \rightarrow (\mathbb{N} \times \text{Stack}) \uplus \mathbb{N}$$

- ▶ unload^+ is the “step” function of an abstract machine!
- ▶ Configurations: $\text{Config} = \mathbb{N} \times \text{Stack}$



$$\text{unload}^+ : \text{Config} \rightarrow \text{Config} \uplus \mathbb{N}$$

- ▶ unload^+ is the “step” function of an abstract machine!
- ▶ Configurations: $\text{Config} = \mathbb{N} \times \text{Stack}$



- ▶ Function $f : \alpha \rightarrow \alpha$ **not** defined by structural recursion.



- ▶ Function $f : \alpha \rightarrow \alpha$ **not** defined by structural recursion.
- ▶ A relation $_<_ : \alpha \rightarrow \alpha \rightarrow \text{Set}$, such that $f' \alpha < \alpha$.



- ▶ Function $f : \alpha \rightarrow \alpha$ **not** defined by structural recursion.
- ▶ A relation $_<_ : \alpha \rightarrow \alpha \rightarrow \text{Set}$, such that $f' \alpha < \alpha$.
- ▶ No infinitely long **decreasing chains**
 \implies recursion eventually terminates.



- ▶ Function $f : \alpha \rightarrow \alpha$ **not** defined by structural recursion.
- ▶ A relation $_<_ : \alpha \rightarrow \alpha \rightarrow \text{Set}$, such that $f' \alpha < \alpha$.
- ▶ No infinitely long **decreasing chains**
 \implies recursion eventually terminates.
- ▶ In *Agda*, the **accessibility predicate**.



- ▶ Function $f : \alpha \rightarrow \alpha$ **not** defined by structural recursion.
- ▶ A relation $_<_ : \alpha \rightarrow \alpha \rightarrow \text{Set}$, such that $f' \alpha < \alpha$.
- ▶ No infinitely long **decreasing chains**
 \implies recursion eventually terminates.
- ▶ In *Agda*, the **accessibility predicate**.
- ▶ A relation is **well-founded** if every element is accessible.



Iterate `unload+` until a value is produced.



Iterate unload^+ until a value is produced.

$\text{tail-rec-eval} : \text{Expr} \rightarrow \mathbb{N}$

$\text{tail-rec-eval } e \text{ with load } e \text{ Top}$

$\dots \mid \text{inj}_1 c = \text{rec } (<\text{-Well-founded } c) c$

where

$\text{rec} : (c : \text{Config}) \rightarrow \text{Acc } _<_ c \rightarrow \mathbb{N}$

$\text{rec } c (\text{acc } rs) \text{ with } \text{unload}^+ c$

$\dots \mid \text{inj}_1 c' = \text{rec } (rs \dots) c'$

$\dots \mid \text{inj}_2 r = r$



Iterate unload^+ until a value is produced.

$\text{tail-rec-eval} : \text{Expr} \rightarrow \mathbb{N}$

$\text{tail-rec-eval } e \text{ with load } e \text{ Top}$

$\dots \mid \text{inj}_1 c = \text{rec } (<\text{-Well-founded } c) c$

where

$\text{rec} : (c : \text{Config}) \rightarrow \text{Acc } _<_ c \rightarrow \mathbb{N}$

$\text{rec } c (\text{acc } rs) \text{ with } \text{unload}^+ c$

$\dots \mid \text{inj}_1 c' = \text{rec } (rs \dots) c'$

$\dots \mid \text{inj}_2 r = r$

Still, we have to:



Iterate unload^+ until a value is produced.

$\text{tail-rec-eval} : \text{Expr} \rightarrow \mathbb{N}$

$\text{tail-rec-eval } e \text{ with load } e \text{ Top}$

$\dots \mid \text{inj}_1 c = \text{rec } (<- \text{Well-founded } c) c$

where

$\text{rec} : (c : \text{Config}) \rightarrow \text{Acc } _<_ c \rightarrow \mathbb{N}$

$\text{rec } c (\text{acc } rs) \text{ with } \text{unload}^+ c$

$\dots \mid \text{inj}_1 c' = \text{rec } (rs \dots) c'$

$\dots \mid \text{inj}_2 r = r$

Still, we have to:

► $_<_ : \text{Config} \rightarrow \text{Config} \rightarrow \text{Set}$



Iterate unload^+ until a value is produced.

$\text{tail-rec-eval} : \text{Expr} \rightarrow \mathbb{N}$

$\text{tail-rec-eval } e \text{ with load } e \text{ Top}$

$\dots \mid \text{inj}_1 c = \text{rec } (<\text{-Well-founded } c) c$

where

$\text{rec} : (c : \text{Config}) \rightarrow \text{Acc } _<_ c \rightarrow \mathbb{N}$

$\text{rec } c (\text{acc } rs) \text{ with } \text{unload}^+ c$

$\dots \mid \text{inj}_1 c' = \text{rec } (rs \dots) c'$

$\dots \mid \text{inj}_2 r = r$

Still, we have to:

▶ $_<_ : \text{Config} \rightarrow \text{Config} \rightarrow \text{Set}$

▶ $\forall (c c' : \text{Config}) \rightarrow \text{unload}^+ c \equiv \text{inj}_1 c' \rightarrow c' < c$



Iterate unload^+ until a value is produced.

$\text{tail-rec-eval} : \text{Expr} \rightarrow \mathbb{N}$

$\text{tail-rec-eval } e \text{ with load } e \text{ Top}$

$\dots \mid \text{inj}_1 c = \text{rec } (<\text{-Well-founded } c) c$

where

$\text{rec} : (c : \text{Config}) \rightarrow \text{Acc } _<_ c \rightarrow \mathbb{N}$

$\text{rec } c (\text{acc } rs) \text{ with } \text{unload}^+ c$

$\dots \mid \text{inj}_1 c' = \text{rec } (rs \dots) c'$

$\dots \mid \text{inj}_2 r = r$

Still, we have to:

► $_<_ : \text{Config} \rightarrow \text{Config} \rightarrow \text{Set}$

► $\forall (c c' : \text{Config}) \rightarrow \text{unload}^+ c \equiv \text{inj}_1 c' \rightarrow c' < c$

► $\text{Well-founded } _<_$



- **Config** *uniquely* represents a leaf in an **Expression**.



- ▶ **Config** *uniquely* represents a leaf in an **Expression**.
- ▶ Order configurations left to right.



- ▶ **Config** *uniquely* represents a leaf in an **Expression**.
- ▶ Order configurations left to right.
- ▶ Inductive relation on the **Stack**.



- ▶ **Config** *uniquely* represents a leaf in an **Expression**.
- ▶ Order configurations left to right.
- ▶ Inductive relation on the **Stack**.

data $_ < _$: **Config** \rightarrow **Config** \rightarrow **Set** where

<-StepR : $(t_1, s_1) < (t_2, s_2)$
 $\rightarrow (t_1, \text{Right } l \text{ } n \text{ eq } s_1) < (t_2, \text{Right } l \text{ } n \text{ eq } s_2)$

<-StepL : $(t_1, s_1) < (t_2, s_2)$
 $\rightarrow (t_1, \text{Left } r \text{ } s_1) < (t_2, \text{Left } r \text{ } s_2)$

<-Base : $(t_1, \text{Right } n \text{ } e_1 \text{ eq } s_1) < (t_2, \text{Left } e_2 \text{ } s_2)$



- ▶ Two distinct values of **Config** can represent folds over different **Expressions**.



- ▶ Two distinct values of **Config** can represent folds over different **Expressions**.

data **Config**_↓ (**e** : **Expr**) : **Set** where
 _, _ : (**c** : **Config**) → **plugC**_↓ **c** ≡ **e** → **Config**_↓ **e**



- ▶ Two distinct values of **Config** can represent folds over different **Expressions**.

data **Config**_↓ (**e** : **Expr**) : **Set** where
 _, _ : (**c** : **Config**) → **plugC**_↓ **c** ≡ **e** → **Config**_↓ **e**

- ▶ A **type-indexed** relation to enforce the invariant.



- ▶ Two distinct values of **Config** can represent folds over different **Expressions**.

data **Config**_↓ (**e** : **Expr**) : **Set** where
 _, _ : (**c** : **Config**) → **plugC**_↓ **c** ≡ **e** → **Config**_↓ **e**

- ▶ A **type-indexed** relation to enforce the invariant.

⊆_↓ : (**e** : **Expr**) → **Config**_↓ **e** → **Config**_↓ **e** → **Set**



- ▶ Two distinct values of **Config** can represent folds over different **Expressions**.

data **Config**_↓ (**e** : **Expr**) : **Set** where
 _, _ : (**c** : **Config**) → **plugC**_↓ **c** ≡ **e** → **Config**_↓ **e**

- ▶ A **type-indexed** relation to enforce the invariant.

⊆_↓ : (**e** : **Expr**) → **Config**_↓ **e** → **Config**_↓ **e** → **Set**

- ▶ It is needed to show the relation is **well-founded**.



- ▶ Proof of well-foundedness: not too hard.



- ▶ Proof of **well-foundedness**: not too hard.
- ▶ Proof that **unload⁺** delivers a smaller result: tedious but easy.



- ▶ Proof of **well-foundedness**: not too hard.
- ▶ Proof that **unload⁺** delivers a smaller result: tedious but easy.
- ▶ Proof of correctness: follows from well-founded recursion.



Motivation
Contributions
Solving the problem
Generalization
Conclusion



- ▶ Universe of **regular** datatypes of which **Expr** is an instance.



- ▶ Universe of **regular** datatypes of which **Expr** is an instance.
- ▶ McBride's dissection to calculate generic **Stacks**.



- ▶ Universe of **regular** datatypes of which **Expr** is an instance.
- ▶ McBride's dissection to calculate generic **Stacks**.
- ▶ Generalize **load^G** and **unload^G**.



- ▶ Universe of **regular** datatypes of which **Expr** is an instance.
- ▶ McBride's dissection to calculate generic **Stacks**.
- ▶ Generalize **load^G** and **unload^G**.
- ▶ Generic well-founded relation.



- ▶ Universe of **regular** datatypes of which **Expr** is an instance.
- ▶ McBride's dissection to calculate generic **Stacks**.
- ▶ Generalize **load^G** and **unload^G**.
- ▶ Generic well-founded relation.
- ▶ **unload^G** decreases.



- ▶ Universe of **regular** datatypes of which **Expr** is an instance.
- ▶ McBride's dissection to calculate generic **Stacks**.
- ▶ Generalize **load^G** and **unload^G**.
- ▶ Generic well-founded relation.
- ▶ **unload^G** decreases.
- ▶ Assemble everything together.



```
data Reg : Set1 where
  0      : Reg
  1      : Reg
  I      : Reg
  K      : (A : Set) → Reg
  _⊕_    : (R Q : Reg) → Reg
  _⊗_    : (R Q : Reg) → Reg
```



data $\text{Reg} : \text{Set}_1$ where

$0 : \text{Reg}$

$1 : \text{Reg}$

$I : \text{Reg}$

$K : (A : \text{Set}) \rightarrow \text{Reg}$

$_ \oplus _ : (R\ Q : \text{Reg}) \rightarrow \text{Reg}$

$_ \otimes _ : (R\ Q : \text{Reg}) \rightarrow \text{Reg}$

$\llbracket _ \rrbracket : \text{Reg} \rightarrow (\text{Set} \rightarrow \text{Set})$

$\llbracket 0 \rrbracket X = \perp$

$\llbracket 1 \rrbracket X = \top$

$\llbracket I \rrbracket X = X$

$\llbracket (K\ A) \rrbracket X = A$

$\llbracket (R \oplus Q) \rrbracket X = \llbracket R \rrbracket X \uplus \llbracket Q \rrbracket X$

$\llbracket (R \otimes Q) \rrbracket X = \llbracket R \rrbracket X \times \llbracket Q \rrbracket X$



data $\text{Reg} : \text{Set}_1$ where

$0 : \text{Reg}$

$1 : \text{Reg}$

$I : \text{Reg}$

$K : (A : \text{Set}) \rightarrow \text{Reg}$

$_ \oplus _ : (R\ Q : \text{Reg}) \rightarrow \text{Reg}$

$_ \otimes _ : (R\ Q : \text{Reg}) \rightarrow \text{Reg}$

$\llbracket _ \rrbracket : \text{Reg} \rightarrow (\text{Set} \rightarrow \text{Set})$

$\llbracket 0 \rrbracket X = \perp$

$\llbracket 1 \rrbracket X = \top$

$\llbracket I \rrbracket X = X$

$\llbracket (K\ A) \rrbracket X = A$

$\llbracket (R \oplus Q) \rrbracket X = \llbracket R \rrbracket X \uplus \llbracket Q \rrbracket X$

$\llbracket (R \otimes Q) \rrbracket X = \llbracket R \rrbracket X \times \llbracket Q \rrbracket X$

data $\mu (R : \text{Reg}) : \text{Set}$ where

$\text{In} : \llbracket R \rrbracket (\mu R) \rightarrow \mu R$



data $\text{Reg} : \text{Set}_1$ where

$0 : \text{Reg}$

$1 : \text{Reg}$

$I : \text{Reg}$

$K : (A : \text{Set}) \rightarrow \text{Reg}$

$_ \oplus _ : (R\ Q : \text{Reg}) \rightarrow \text{Reg}$

$_ \otimes _ : (R\ Q : \text{Reg}) \rightarrow \text{Reg}$

$\llbracket _ \rrbracket : \text{Reg} \rightarrow (\text{Set} \rightarrow \text{Set})$

$\llbracket 0 \rrbracket X = \perp$

$\llbracket 1 \rrbracket X = \top$

$\llbracket I \rrbracket X = X$

$\llbracket (K\ A) \rrbracket X = A$

$\llbracket (R \oplus Q) \rrbracket X = \llbracket R \rrbracket X \uplus \llbracket Q \rrbracket X$

$\llbracket (R \otimes Q) \rrbracket X = \llbracket R \rrbracket X \times \llbracket Q \rrbracket X$

data $\mu (R : \text{Reg}) : \text{Set}$ where

$\text{In} : \llbracket R \rrbracket (\mu R) \rightarrow \mu R$

$\text{cata} : (R : \text{Reg}) \rightarrow (\llbracket R \rrbracket X \rightarrow X) \rightarrow \mu R \rightarrow X$

$\text{cata}\ R\ \psi (\text{In}\ r) = \psi (\text{fmap}\ R\ (\text{cata}\ R\ \psi)\ r)$



- Original type:

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



- Original type:

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

- Generic representation:

```
exprF : Reg
exprF = K ℕ ⊕ (I ⊗ I)
```



- ▶ Original type:

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

- ▶ Generic representation:

```
exprF : Reg
exprF = K ℕ ⊕ (I ⊗ I)
```

- ▶ Isomorphism:

$$\text{Expr} \simeq \mu \text{exprF}$$


- “One hole” context of a functor.

$$\nabla : (R : \text{Reg}) \rightarrow (\text{Set} \rightarrow \text{Set} \rightarrow \text{Set})$$

$$\nabla \emptyset \quad XY = \perp$$

$$\nabla \mathbb{1} \quad XY = \perp$$

$$\nabla \mathbb{I} \quad XY = \top$$

$$\nabla (K A) \quad XY = \perp$$

$$\nabla (R \oplus Q) \quad XY = \nabla RXY \uplus \nabla QXY$$

$$\nabla (R \otimes Q) \quad XY = (\nabla RXY \times \llbracket Q \rrbracket Y) \uplus (\llbracket R \rrbracket X \times \nabla QXY)$$



- “One hole” context of a functor.

$$\nabla : (R : \text{Reg}) \rightarrow (\text{Set} \rightarrow \text{Set} \rightarrow \text{Set})$$

$$\nabla \mathbb{0} \quad XY = \perp$$

$$\nabla \mathbb{1} \quad XY = \perp$$

$$\nabla \mathbb{I} \quad XY = \top$$

$$\nabla (\mathbb{K} A) \quad XY = \perp$$

$$\nabla (R \oplus Q) \quad XY = \nabla RXY \uplus \nabla QXY$$

$$\nabla (R \otimes Q) \quad XY = (\nabla RXY \times \llbracket Q \rrbracket Y) \uplus (\llbracket R \rrbracket X \times \nabla QXY)$$

- By example:

$$\text{Stack} \simeq \text{List} (\nabla \text{exprF } \mathbb{N} (\mu \text{exprF}))$$



Motivation
Contributions
Solving the problem
Generalization
Conclusion



Our results

§5



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

A generic tail-recursive evaluator

$\text{tail-rec-cata} : (R : \text{Reg}) \rightarrow (\psi : \llbracket R \rrbracket X \rightarrow X) \rightarrow \mu R \rightarrow X$



A generic tail-recursive evaluator

$\text{tail-rec-cata} : (R : \text{Reg}) \rightarrow (\psi : \llbracket R \rrbracket X \rightarrow X) \rightarrow \mu R \rightarrow X$

and its correctness proof.

$\text{correctness}^G : \forall (R : \text{Reg}) (\psi : \llbracket R \rrbracket X \rightarrow X) (t : \mu R)$
 $\rightarrow \text{cata } R \psi t \equiv \text{tail-rec-cata } R \psi t$



A generic tail-recursive evaluator

$\text{tail-rec-cata} : (R : \text{Reg}) \rightarrow (\psi : \llbracket R \rrbracket X \rightarrow X) \rightarrow \mu R \rightarrow X$

and its correctness proof.

$\text{correctness}^G : \forall (R : \text{Reg}) (\psi : \llbracket R \rrbracket X \rightarrow X) (t : \mu R)$
 $\rightarrow \text{cata } R \psi t \equiv \text{tail-rec-cata } R \psi t$

Still a lot of details to fill in:

- What is a leaf, generically?



A generic tail-recursive evaluator

$\text{tail-rec-cata} : (R : \text{Reg}) \rightarrow (\psi : \llbracket R \rrbracket X \rightarrow X) \rightarrow \mu R \rightarrow X$

and its correctness proof.

$\text{correctness}^G : \forall (R : \text{Reg}) (\psi : \llbracket R \rrbracket X \rightarrow X) (t : \mu R)$
 $\rightarrow \text{cata } R \psi t \equiv \text{tail-rec-cata } R \psi t$

Still a lot of details to fill in:

- ▶ What is a leaf, generically?
- ▶ How do we compare generic configurations?



A generic tail-recursive evaluator

$\text{tail-rec-cata} : (R : \text{Reg}) \rightarrow (\psi : \llbracket R \rrbracket X \rightarrow X) \rightarrow \mu R \rightarrow X$

and its correctness proof.

$\text{correctness}^G : \forall (R : \text{Reg}) (\psi : \llbracket R \rrbracket X \rightarrow X) (t : \mu R)$
 $\rightarrow \text{cata } R \psi t \equiv \text{tail-rec-cata } R \psi t$

Still a lot of details to fill in:

- ▶ What is a leaf, generically?
- ▶ How do we compare generic configurations?
- ▶ Generic relation and its well-foundedness proof



A generic tail-recursive evaluator

$\text{tail-rec-cata} : (R : \text{Reg}) \rightarrow (\psi : \llbracket R \rrbracket X \rightarrow X) \rightarrow \mu R \rightarrow X$

and its correctness proof.

$\text{correctness}^G : \forall (R : \text{Reg}) (\psi : \llbracket R \rrbracket X \rightarrow X) (t : \mu R)$
 $\rightarrow \text{cata } R \psi t \equiv \text{tail-rec-cata } R \psi t$

Still a lot of details to fill in:

- ▶ What is a leaf, generically?
- ▶ How do we compare generic configurations?
- ▶ Generic relation and its well-foundedness proof
- ▶ ...



For more technical details:
My master's thesis (long) or the **paper** (short)
and the *Agda code*.



For more technical details:
My master's thesis (long) or the **paper** (short)
and the *Agda code*.

Thank you very much for your attention!

