From Algebra to Abstract Machine: A Verified Generic Construction

TyDe'18

Carlos Tomé Cortiñas* and Wouter Swierstra

September 2018

* Sponsored by

Utrechts Universiteitsfonds | Universiteit Utrecht

To put it another way ...

Verified tail-recursive folds through dissection

Motivation

Motivation

We start with a small Expression language

Motivation

We start with a small Expression language

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

Motivation

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.

Motivation

We start with a small Expression language

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

and we write an evaluator for it.

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

A Problem with eval


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

A Problem with eval


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

$* * * \; Exception : stack \; overflow$

A Problem with eval


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

What happened?

A Problem with eval

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

$***\ Exception : stack\ overflow$

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

A Problem with eval

$>$ 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.

A Problem with eval

> 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.

A Problem with eval

> 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

A solution to the problem

Write a **tail-recursive** evaluator.

Write a **tail-recursive** evaluator.

We proceed in *three steps*:

A solution to the problem

Write a **tail-recursive** evaluator.

We proceed in *three steps*:

- Make the underlying stack explicit.

A solution to the problem

Write a **tail-recursive** evaluator.

We proceed in *three steps*:

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

A solution to the problem

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

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Stack}\ :\ \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{Top}\ \ \ :\ \mathsf{Stack} \\
\quad \mathsf{Left}\ \ :\ \mathsf{Expr} \to \mathsf{Stack} \to \mathsf{Stack} \\
\quad \mathsf{Right}\ :\ \mathbb{N} \ \ \ \ \to \mathsf{Stack} \to \mathsf{Stack}
\end{array}
$$

A tail-recursive evaluator

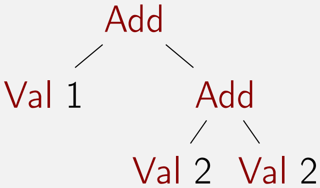
```
data Stack : Set where
  Top   : Stack
  Left  : Expr → Stack → Stack
  Right : ℕ    → Stack → Stack

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 (Left e stk)     = load e (Right v stk)
  unload v (Right v' stk)   = unload (v' + v) stk
```

A tail-recursive evaluator


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

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 (Left e stk)     = load e (Right v stk)
  unload v (Right v' stk)   = unload (v' + v) stk

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

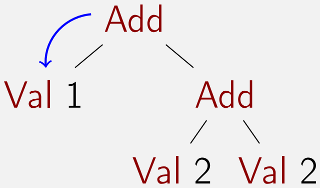
tail-rec-eval in action

load (Add (Val 1) (Add (Val 2) (Val 2))) <u>Top</u>



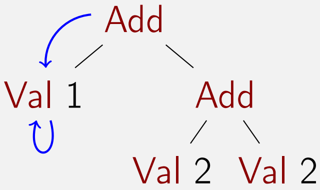
tail-rec-eval in action

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



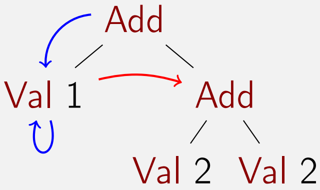
tail-rec-eval in action

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



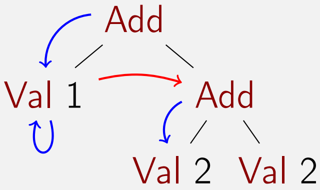
tail-rec-eval in action

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



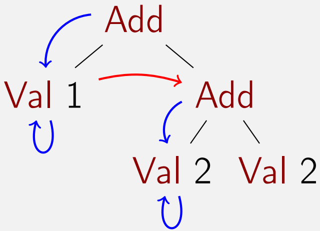
tail-rec-eval in action

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



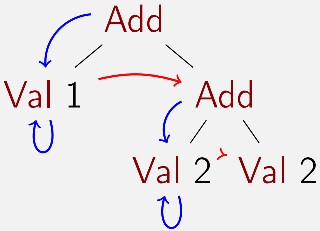
tail-rec-eval in action

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



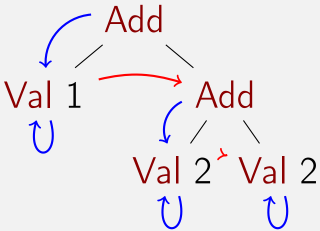
tail-rec-eval in action

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



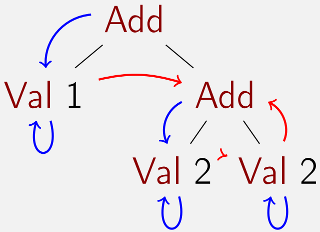
tail-rec-eval in action

unload 2 (Right 2 (Right 1 Top))



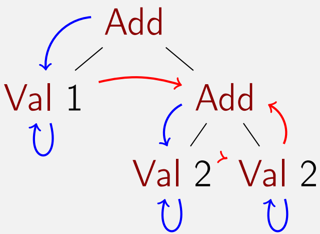
tail-rec-eval in action

unload (2 + 2) (Right 1 Top)



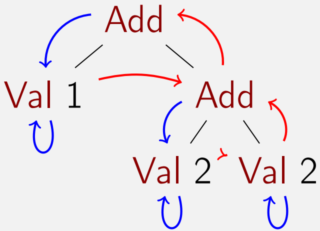
tail-rec-eval in action

unload 4 (Right 1 Top)



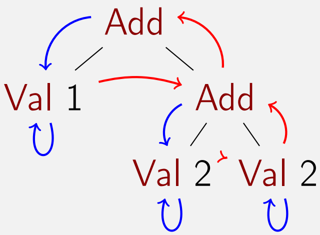
tail-rec-eval in action

unload $(1 + 4)$ Top



tail-rec-eval in action

unload 5 Top



Problem solved?

Have we **actually** solved the problem?

Problem solved?

Have we **actually** solved the problem?

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

Problem solved?

Have we **actually** solved the problem?

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

Problem solved?

Have we **actually** solved the problem?

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

mutual

$$
\begin{aligned}
&\boxed{\mathsf{load}} : \mathsf{Expr} \to \mathsf{Stack} \to \mathbb{N}\\
&\ldots\\
&\boxed{\mathsf{unload}} : \mathbb{N} \to \mathsf{Stack} \to \mathbb{N}\\
&\mathsf{unload}\ v\ \mathsf{Top} = v\\
&\mathsf{unload}\ v\ (\mathsf{Left}\ e\ stk) = \boxed{\mathsf{load}}\ e\ (\mathsf{Right}\ v\ stk)\\
&\mathsf{unload}\ v\ (\mathsf{Right}\ v'\ stk) = \boxed{\mathsf{unload}}\ (v' + v)\ stk
\end{aligned}
$$

Problem solved?

Have we **actually** solved the problem?

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

mutual

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

We **lost** termination guarantees.

Motivation

Contributions

Solving the problem

Generalization

Conclusion

Contributions

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

Contributions

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

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

That's not all

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

That's not all

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

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

That's not all

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

$$
\begin{array}{lcl}
\mathsf{fold} & : & (\alpha \to \alpha \to \alpha) \to (\mathbb{N} \to \alpha) \to \mathsf{Expr} \to \mathbb{N} \\
\mathsf{fold}\ \phi_1\ \phi_2\ (\mathsf{Val}\ n) & = & \phi_2\ n \\
\mathsf{fold}\ \phi_1\ \phi_2\ (\mathsf{Add}\ e_1\ e_2) & = & \phi_1\ (\mathsf{fold}\ \phi_1\ \phi_2\ e_1)\ (\mathsf{fold}\ \phi_1\ \phi_2\ e_2)
\end{array}
$$

Both versions of the function are <u>the same</u>.

That's not all

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

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

Both versions of the function are <u>the same</u>.

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

That's not all

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

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

Both versions of the function are <u>the same</u>.

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

Moreover:

That's not all

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

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

Both versions of the function are <u>the same</u>.

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

Moreover:

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

That's not all

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

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

Both versions of the function are <u>the same</u>.

$$\forall\,(e : \mathsf{Expr}) \to \mathsf{eval}\ e \equiv \mathsf{fold}\ \_+\_\ \mathsf{id}\ e$$

Moreover:

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

Contributions

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

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

Contributions

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.

Contributions

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

Plan of attack

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

Plan of attack

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

Plan of attack

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.

Plan of attack

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.

Breaking down mutual recursion

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

Breaking down mutual recursion

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

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

Breaking down mutual recursion

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

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

Now, both functions obviously terminate.

A step back

$$\mathsf{unload}^{+} \;:\; (\mathbb{N} \times \mathsf{Stack}) \;\to\; (\mathbb{N} \times \mathsf{Stack}) \uplus \mathbb{N}$$

A step back

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

- unload^{+} is the “step“ function of an <u>abstract machine</u>!

A step back

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

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

A step back

$$\mathsf{unload}^+ : (\mathbb{N} \times \mathsf{Stack}) \to (\mathbb{N} \times \mathsf{Stack}) \uplus \mathbb{N}$$

- unload^+ is the “step“ function of an abstract machine!
- Configurations: $\mathsf{Config} = \mathbb{N} \times \mathsf{Stack}$
- Step function: $\mathsf{unload}^+ : \mathsf{Config} \to \mathsf{Config} \uplus \mathbb{N}$

Well-founded recursion

- $f : \alpha \to \alpha$ **not** defined by structural recursion.

Well-founded recursion

- $f : \alpha \to \alpha$ **not** defined by structural recursion.
- A relation $\_<\_ : \alpha \to \alpha \to \mathsf{Set}$ such that $f\ \alpha < \alpha$.

Well-founded recursion

- $f : \alpha \to \alpha$ **not** defined by structural recursion.
- A relation $\_<\_ : \alpha \to \alpha \to \mathsf{Set}$ such that $f\ \alpha < \alpha$.
- No infinitely long **decreasing chains**
  $\Longrightarrow$ recursion eventually terminates.

Well-founded recursion

- $f : \alpha \to \alpha$ **not** defined by structural recursion.
- A relation $\_<\_ : \alpha \to \alpha \to \mathsf{Set}$ such that $f\ \alpha < \alpha$.
- No infinitely long **decreasing chains**
  $\Longrightarrow$ recursion eventually terminates.
- In *Agda* we encode such property as an accessibility predicate.

Well-founded recursion

- $f : \alpha \to \alpha$ **not** defined by structural recursion.
- A relation $\_<\_ : \alpha \to \alpha \to \mathsf{Set}$ such that $f\ \alpha < \alpha$.
- No infinitely long **decreasing chains**
  $\Longrightarrow$ recursion eventually terminates.
- In *Agda* we encode such property as an accessibility predicate.
- A relation is **well-founded** if every element is accessible.

Another tail-recursive evaluator §3

Iterate unload$^+$ until a value is produced.

Another tail-recursive evaluator

Iterate unload$^+$ until a value is produced.

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ c = rec (<-Well-founded c) c
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec c (acc rs) with unload⁺ c
    ... | inj₁ c' = rec (rs ...) c'
    ... | inj₂ r  = r
```


Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Another tail-recursive evaluator §3

Iterate unload^{+} until a value is produced.

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ c = rec (<-Well-founded c) c
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec c (acc rs) with unload⁺ c
    ... | inj₁ c' = rec (rs ...) c'
    ... | inj₂ r  = r
```

Still, we have to:

Another tail-recursive evaluator

Iterate unload^{+} until a value is produced.

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ c = rec (<-Well-founded c) c
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec c (acc rs) with unload⁺ c
    ... | inj₁ c′ = rec (rs ...) c′
    ... | inj₂ r  = r
```

Still, we have to:

- `_<_ : Config → Config → Set`

Another tail-recursive evaluator

Iterate unload^{+} until a value is produced.

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ c = rec (<-Well-founded c) c
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec c (acc rs) with unload⁺ c
    ... | inj₁ c' = rec (rs ...) c'
    ... | inj₂ r  = r
```

Still, we have to:

- $\_{<}\_ : \text{Config} \to \text{Config} \to \text{Set}$
- $\forall\, (c\ c' : \text{Config}) \to \text{unload}^{+}\ c \equiv \text{inj}_1\ c' \to c' < c$

Another tail-recursive evaluator

Iterate unload^{+} until a value is produced.

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
  ... | inj₁ c = rec (<-Well-founded c) c
  where
    rec : (c : Config) → Acc _<_ c → ℕ
    rec c (acc rs) with unload⁺ c
    ... | inj₁ c' = rec (rs ...) c'
    ... | inj₂ r  = r
```

Still, we have to:

- $\_<\_ \; : \; \mathsf{Config} \to \mathsf{Config} \to \mathsf{Set}$
- $\forall \, (c \; c' \; : \; \mathsf{Config}) \to \mathsf{unload}^{+} \; c \equiv \mathsf{inj}_1 \; c' \to c' < c$
- $\mathsf{Well\text{-}founded} \; \_<\_$

The relation

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

The relation

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

The relation

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

The relation

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

$$
\begin{aligned}
&\text{data } \_<\_ : \text{Config} \to \text{Config} \to \text{Set where} \\
&\quad \text{<-StepR} : (t_1 , s_1) < (t_2 , s_2) \\
&\qquad\qquad \to (t_1 , \text{Right } l\ n\ eq\ s_1) < (t_2 , \text{Right } l\ n\ eq\ s_2) \\
&\quad \text{<-StepL} : (t_1 , s_1) < (t_2 , s_2) \\
&\qquad\qquad \to (t_1 , \text{Left } r\ s_1) < (t_2 , \text{Left } r\ s_2) \\
&\quad \text{<-Base} : (t_1 , \text{Right } n\ e_1\ eq\ s_1) < (t_2 , \text{Left } e_2\ s_2)
\end{aligned}
$$

The details

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

The details

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

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

The details

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

$$\text{data Config}_{\Downarrow}\ (e : \text{Expr}) : \text{Set where}$$
$$\_,\_ : (c : \text{Config}) \rightarrow \text{plugC}_{\Downarrow}\ c \equiv e \rightarrow \text{Config}_{\Downarrow}\ e$$

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

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

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

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

$$\llcorner\_\lrcorner\_<\_ : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Downarrow}\ e \to \mathsf{Config}_{\Downarrow}\ e \to \mathsf{Set}$$

The details

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

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

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

$$\llcorner\_\lrcorner\_<\_ : (e : \mathsf{Expr}) \to \mathsf{Config}_{\Downarrow}\ e \to \mathsf{Config}_{\Downarrow}\ e \to \mathsf{Set}$$

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

The rest (omitted)

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

The rest (omitted)

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

The rest (omitted)

- 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

Generalization

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

Generalization

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

Generalization

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

Generalization

- 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.

Generalization

- 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.

Generalization

- 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.

regular universe

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

regular universe

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

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

regular universe

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

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

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

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

regular universe

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

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

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

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

An example

- Original type:

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

An example

- Original type:

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

- Generic representation:

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

An example

- Original type:

$$\begin{aligned}&\mathsf{data}\ \mathsf{Expr} : \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{Val} : \mathbb{N} \to \mathsf{Expr}\\ &\quad \mathsf{Add} : \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}\end{aligned}$$

- Generic representation:

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

- Isomorphic:

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

- "One hole" context of a functor.

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

Dissections

- "One hole" context of a functor.

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

- An example:

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

Our results

Our results

A generic tail-recursive evaluator

$$\text{tail-rec-cata} : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \to \mu\, R \to X$$

Our results

A generic tail-recursive evaluator

$$\mathsf{tail\text{-}rec\text{-}cata} : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \to \mu\, R \to X$$

and its correctness proof.

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

Our results

A generic tail-recursive evaluator

$$\mathsf{tail\text{-}rec\text{-}cata} : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \to \mu\, R \to X$$

and its correctness proof.

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

Still a lot of details to fill in:

- What is a leaf, generically?

Our results

A generic tail-recursive evaluator

$$\mathsf{tail\text{-}rec\text{-}cata} : (R : \mathsf{Reg}) \to (\psi : [\![R]\!]\, X \to X) \to \mu\, R \to X$$

and its correctness proof.

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

Still a lot of details to fill in:

- What is a leaf, generically?
- How to we compare generic configurations.

Our results

A generic tail-recursive evaluator

$$\text{tail-rec-cata} : (R : \text{Reg}) \to (\psi : [\![R]\!]\, X \to X) \to \mu\, R \to X$$

and its correctness proof.

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

Still a lot of details to fill in:

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

Our results

A generic tail-recursive evaluator

$$\text{tail-rec-cata} : (R : \text{Reg}) \to (\psi : [\![R]\!]\ X \to X) \to \mu\ R \to X$$

and its correctness proof.

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

Still a lot of details to fill in:

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

For more details read the paper
and the *Agda* code.

For more details read the paper
and the *Agda* code.

Thank you very much for your attention!