From algebra to abstract machine

IFIP WG 2.1, Brandenburg

Wouter Swierstra

joint work with Carlos Tomé Cortiñas

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Evaluation

We can define a small expression language:

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

And define a recursive evaluation function:

```
eval : Expr → Nat
eval (Expr l r) = eval l + eval r
eval (Val i)    = i
```

Or using folds...

Instead of defining the evaluator using recursion directly, we can define it using the **fold** over expressions:

```
foldExpr : (a → a → a) → (Nat → a) → Expr → a
foldExpr add val (Add l r) =
  add (foldExpr add val l) (foldExpr add val r)
foldExpr add val (Val i) = val i

eval :: Expr → Nat
eval = foldExpr _+_ id
```

A problem

```
> eval (Add (Add (Add ... (Add (Val 1) (Val 1)).....)))
*** Exception: stack overflow
```

What went wrong?

A problem

```
> eval (Add (Add (Add ... (Add (Val 1) (Val 1))....)))
*** Exception: stack overflow
```

What went wrong?

The `foldExpr` function needs to fully evaluate both subtrees of an addition before the addition function can trigger further reduction.

For large subtrees, this involves pushing stack frames for each recursive call until we have reached the leaves.

As the stack grows, it may overflow on large inputs.

A solution: a tail-recursive evaluator

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

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

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

Termination woes

```
mutual
  load : Expr → Stack → Nat
  ...
  unload : Nat → Stack → Nat
  unload v (Right v' stk)  = unload (v' + v) stk
  unload v (Left r stk)    = load r (Right v stk)
```

This definition, however, is not obviously structurally recursive – and therefore rejected by Agda.

The problematic call is in the last line – why is it safe to call `load` on some expression that you happen to find on the stack?

Folds vs tail-recursive functions

Folds closely follow the structure of our data...

Folds vs tail-recursive functions

Folds closely follow the structure of our data...

... but may lead to stack overflows.

Folds vs tail-recursive functions

Folds closely follow the structure of our data...

... but may lead to stack overflows.

Defining a **tail-recursive** evaluator fixes this last problem...

Folds vs tail-recursive functions

Folds closely follow the structure of our data…

… but may lead to stack overflows.

Defining a **tail-recursive** evaluator fixes this last problem…

… but we've lost the structural recursion that guarantees termination.

Folds vs tail-recursive functions

Folds closely follow the structure of our data…

… but may lead to stack overflows.

Defining a **tail-recursive** evaluator fixes this last problem…

… but we've lost the structural recursion that guarantees termination.

Challenge: How to show the tail-recursive evaluator terminates? And can we prove that it produces the same result as the fold-based evaluator?

Plan of attack

1. Tease apart the mutual recursion in the load-unload functions;
2. Iteratively call unload to navigate between the leaves of our tree, until we produce a value;
3. Show that each recursive call is to a ‘smaller’ `Stack` $\times$ `Nat` (or *configuration*) – and therefore is guaranteed to terminate.
4. Prove that the result produced in this style is equal to our original evaluator.
5. Generalize all these steps to work for *any* function defined as a fold over an algebraic data type.

Breaking the mutual recursion

```
load : Expr → Stack → (Nat × Stack) ⊎ Nat
load (Val n)     stk = inj₁ (n , stk)
load (Add e₁ e₂) stk = load e₁ (Left e₂ stk)

unload : Nat → Stack → (Nat × Stack) ⊎ Nat
unload v Top              = inj₂ v
unload v (Right v' stk)   = unload (v' + v) stk
unload v (Left r stk)     = load r (Right v stk)
```

The `load` function no longer calls unload upon reaching a value, instead it returns a new configuration.

The `unload` function no longer computes the final result – it may also return a new configuration, if `load` does.

Instead of consuming the whole tree, we stop at the leaves.

Iterating through the tree

```
tail-rec-eval : Expr → Nat
tail-rec-eval e = rec (load e Top)
  where
    rec : (Nat × Stack) → Nat
    rec (n , stk) with unload n stk
    ... | inj₁ (n' , stk' )  = rec (n' , stk')
    ... | inj₂ v             = v
```

We can now call `unload` over and over again until the stack is empty and we have the final value.

Iterating through the tree

```
tail-rec-eval : Expr → Nat
tail-rec-eval e = rec (load e Top)
  where
    rec : (Nat × Stack) → Nat
    rec (n , stk) with unload n stk
    ... | inj₁ (n' , stk' )  = rec (n' , stk')
    ... | inj₂ v             = v
```

We can now call `unload` over and over again until the stack is empty and we have the final value.

But why does this terminate? The result of calling `unload` is not structurally smaller in any way...

Well-founded recursion

When a function `f` is not obviously structurally recursive, *well-founded recursion* can be used to offer an explanation why it terminates.

The idea is that when defining `f(a)`, we're allowed to call `f` on any argument 'smaller than' `a`...

... provided we cannot construct infinitely long decreasing chains, i.e., any series of calls terminates.

The canonical example is `quickSort`, where you make recursive calls to *strictly shorter* lists that are *not* the immediate tail of the list.

Well-founded recursion

When a function `f` is not obviously structurally recursive, *well-founded recursion* can be used to offer an explanation why it terminates.

The idea is that when defining `f(a)`, we're allowed to call `f` on any argument 'smaller than' `a`...

... provided we cannot construct infinitely long decreasing chains, i.e., any series of calls terminates.

The canonical example is `quickSort`, where you make recursive calls to *strictly shorter* lists that are *not* the immediate tail of the list.

Problem: What relation can we find between the configurations of our evaluator?

A few observations

Each configuration of our evaluator, that is `Stack` $\times$ `Nat` pair, corresponds uniquely to a leaf in our original input.

After each call to `unload` we navigate to the ‘next’ leaf to the right.

As our input is finite, this process terminates.

Now to make this precise...

One problem

The `Stack` data type is a variation of *zippers*, where we can easily navigate up and down through a tree.

The parent node of the current subtree in focus is stored at the head of the list – making it easy to move upwards.

One problem

The `Stack` data type is a variation of *zippers*, where we can easily navigate up and down through a tree.

The parent node of the current subtree in focus is stored at the head of the list – making it easy to move upwards.

But now we want to compare two positions in the **overall** input expression, rather than navigate to immediate neighbours – our stack is backwards.

Well-founded recursion

If we assume our stacks are reversed, we can define the following relation to relate two positions in the original input:

```
data _<_ : Stack → Stack → Set where
  <-StepR  : s₁ < s₂ → Right n s₁ < Right n s₂
  <-StepL  : s₁ < s₂ → Left e s₁ < Left e s₂
  <-Base   : (Right n :: s₁) < (Left e₂ :: s₂)
```

Well-founded recursion

If we assume our stacks are reversed, we can define the following relation to relate two positions in the original input:

```
data _<_ : Stack → Stack → Set where
  <-StepR  : s₁ < s₂ → Right n s₁ < Right n s₂
  <-StepL  : s₁ < s₂ → Left e s₁ < Left e s₂
  <-Base   : (Right n :: s₁) < (Left e₂ :: s₂)
```

But this relation is not well-founded in general...

Well-founded recursion

If we assume our stacks are reversed, we can define the following relation to relate two positions in the original input:

```
data _<_ : Stack → Stack → Set where
  <-StepR  : s₁ < s₂ → Right n s₁ < Right n s₂
  <-StepL  : s₁ < s₂ → Left e s₁ < Left e s₂
  <-Base   : (Right n :: s₁) < (Left e₂ :: s₂)
```

But this relation is not well-founded in general...

... but we can show it is well-founded if we embellish it with the invariant that we **only** ever compare positions in the **same** original input tree.

What is still missing?

1. Proof that this relation is well-founded (requires careful choice of types, but not too hard);
2. Proof that the `unload` function navigates to smaller configurations (induction over the stacks, 200loc, mostly bookkeeping).
3. Proof relating the tail-recursive evaluator to the original evaluator (follows almost immediately, using well-founded recursion)

What is still missing?

1. Proof that this relation is well-founded (requires careful choice of types, but not too hard);
2. Proof that the `unload` function navigates to smaller configurations (induction over the stacks, 200loc, mostly bookkeeping).
3. Proof relating the tail-recursive evaluator to the original evaluator (follows almost immediately, using well-founded recursion)

These details won't be in the talk – but we have a draft paper online

Generalizing further

1. Instead of fixing our `Expr` data type, we can define a *universe* of regular data types;
2. The configurations, `Stack` $\times$ `Nat`, are particular to our data type and evaluator. McBride's *dissections* give a generic construction for any data type
3. We can generalize our `load` and `unload` functions accordingly;
4. And show that a generic version of the `_<_` relation is well-founded;
5. And define a generic tail-recursive evaluator, mapping any algebra to an abstract machine.
6. And prove that this abstract machine terminates and satisfies its specification.

Polynomial functors

```
data Reg : Set where
  Zero  : Reg
  One   : Reg
  I     : Reg
  K     : (A : Set) → Reg
  _+_   : Reg → Reg → Reg
  _*_   : Reg → Reg → Reg

el : Reg → Set → Set
```

By taking the fix-point of the functors arising from `Reg`, we can represent a recursive data type.

Generic folds

```
data Fix (R : Reg) : Set where
  In : el R (Fix R) → Fix R

cata : (R : Reg) → (el R X → X) → Fix R → X
cata R alg (In t) = alg (fmap R (cata R alg) t)
```

Example:

```
exprF : R
exprF = K Nat + (I * I)

cataExpr = cata exprF

eval : Fix exprF → Nat
eval = cataExpr [ id , _+_]
```

Dissections

Dissections represent ‘one-hole contexts’ – where the elements to the left and right may be of different types.

```
∇ : (R : Reg) → (Set → Set → Set)
∇ I         X Y  = ⊤
∇ (R + Q)   X Y  = ∇ R X Y ⊎ ∇ Q X Y
∇ (R * Q)   X Y  = ∇ R X Y × el Q Y ⊎ el R X × ∇ Q X Y
∇ _         X Y  = ⊥
```

```
D : (R : Reg) → Set → Set → Set
D R X Y = ∇ R X Y × Y
```

For example, `D exprF (Fix exprF) Nat ≈ Nat ⊎ Fix exprF`

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Generic configurations

Each configuration of our abstract machine consists of:

- a leaf (a value of our data type without recursive subtrees);
- a stack, given by a list of dissections storing:
  - unevaluated subtrees
  - or the partial results of evaluating subtrees (and a proof that these are equal to the fold)

Generic configurations

Each configuration of our abstract machine consists of:

- a leaf (a value of our data type without recursive subtrees);
- a stack, given by a list of dissections storing:
  - unevaluated subtrees
  - or the partial results of evaluating subtrees (and a proof that these are equal to the fold)

If we extend our _<_ relation to these configurations and show it is well-founded, we can write a generic version of our tail-recursive traversal.

Results

```
tail-rec-cata : (R : Reg) → (el R X → X) → Fix R → X

correctness : (R : Reg) → (alg : el R X → X) (t : Fix R) →
  cata R alg t ≡ tail-rec-cata R alg t
```

The generic constructions are a bit messier than what I've shown on the slides...

Results

```
tail-rec-cata : (R : Reg) → (el R X → X) → Fix R → X

correctness : (R : Reg) → (alg : el R X → X) (t : Fix R) →
  cata R alg t ≡ tail-rec-cata R alg t
```

The generic constructions are a bit messier than what I've shown on the slides...

- what is a leaf, generically?
- recording intermediate correctness proofs;
- 'reversing' stacks to compare them easily;
- ensuring that we only ever consider decompositions of the same original input;
- ...

Questions?

Questions?

See our draft paper

From algebra to abstract machine: a verified generic construction, Carlos Tomé Cortiñas and Wouter Swierstra, TyDe 2018