From algebra to abstract machine: a verified generic construction

Carlos Tomé Cortiñas
Department of Information and Computing Sciences
Utrecht University
The Netherlands
c.tomecortinas@students.uu.nl

Wouter Swierstra
Department of Information and Computing Sciences
Utrecht University
The Netherlands
w.s.swierstra@uu.nl

Abstract

Many functions over algebraic datatypes can be expressed in terms of a fold. Doing so, however, has one notable drawback: folds are not tail-recursive. As a result, a function defined in terms of a fold may raise a stack overflow when executed. This paper defines a datatype generic, tail-recursive higher-order function that is guaranteed to produce the same result as the fold. Doing so combines the compositional nature of folds and the performance benefits of a hand-written tail-recursive function in a single setting.

Keywords datatype generic programming, catamorphisms, dissection, dependent types, Agda, well-founded recursion

1 Introduction

Folds, or catamorphisms, are a pervasive programming pattern. Folds generalize many simple traversals over algebraic data types. Functions implemented by means of a fold are both compositional and structurally recursive. Consider, for instance, the following expression datatype, written in the programming language Agda [Norell 2007]:

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

We can write a simple evaluator, mapping expressions to natural numbers, as follows:

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

In the case for `Add e₁ e₂`, the `eval` function makes two recursive calls and sums their results. Such a function can be implemented using a fold, passing the addition and identity functions as the argument algebra.

```
fold : (N → X) → (X → X → X) → Expr → X
fold ϕ₁ ϕ₂ (Val n) = ϕ₁ n
fold ϕ₁ ϕ₂ (Add e₁ e₂) = ϕ₂ (fold ϕ₁ ϕ₂ e₁) (fold ϕ₁ ϕ₂ e₂)
```

Unfortunately, not everything in the garden is rosy. The operator `_+_` needs both of its parameters to be fully evaluated before it can reduce further. As a result, the size of the stack used during execution grows linearly with the size of the input, potentially leading to a stack overflow on large inputs.

To address this problem, we can manually rewrite the evaluator to be tail-recursive. Modern compilers typically map tail-recursive functions to machine code that runs in constant memory. To write such a tail-recursive function, we need to introduce an explicit stack storing both intermediate results and the subtrees that have not yet been evaluated.

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

We can define a tail-recursive evaluation function by means of a pair of mutually recursive functions, `load` and `unload`. The `load` function traverses the expressions, pushing subtrees on the stack; the `unload` function unloads the stack, while accumulating a (partial) result.

```
mutable
   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 r stk) = load r (Right v stk)
```

We can now define a tail-recursive version of `eval` by calling `load` with an initially empty stack:

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

Implementing this tail-recursive evaluator comes at a price: Agda’s termination checker flags the `load` and `unload` functions as potentially non-terminating by highlighting them orange. Indeed, in the very last clause of the `unload` function a recursive call is made to arguments that are not syntactically smaller. Furthermore, it is not clear at all that the tail-recursive evaluator produces the same result as our original one. It is precisely these issues that this paper tackles by making the following novel contributions:

```
ify
```

TyDe’18, September 23–29, 2018, St. Louis, MO, USA
2018.
2 Termination and tail-recursion

Before tackling the generic case, we will present the termination and correctness proof for the tail-recursive evaluator presented in the introduction in some detail.

The problematic call for Agda’s termination checker is the last clause of the \( \text{unload} \) function, that calls \( \text{load} \) on the expression stored on the top of the stack. From the definition of \( \text{load} \), it is clear that we only ever push subtrees of the input on the stack. However, the termination checker has no reason to believe that the expression at the top of the stack is structurally smaller in any way. Indeed, if we were to redefine \( \text{load} \) as follows:

\[
\text{load} \ (\text{Add} \ e_1 \ e_2) \ stk = \text{load} \ e_1 \ (\text{Left} \ (f \ e_2) \ stk)
\]

we might use some function \( f : \text{Expr} \to \text{Expr} \) to push arbitrary expressions on the stack, potentially leading to non-termination.

The functions \( \text{load} \) and \( \text{unload} \) use the stack to store subtrees and partial results while folding the input expression. Thus, every node in the original tree is visited twice during the execution: first when the function \( \text{load} \) traverses the tree, until it finds the leftmost leaf; second when \( \text{unload} \) inspects the stack in searching of an unevaluated subtree. This process is depicted in Figure 1.

As there are finitely many nodes on a tree, the depicted traversal using \( \text{load} \) and \( \text{unload} \) must terminate – but how can we convince Agda’s termination checker of this?

As a first approximation, we revise the definitions of \( \text{load} \) and \( \text{unload} \). Rather than consuming the entire input in one go with a pair of mutually recursive functions, we rewrite them to compute one ’step’ of the fold.

The function \( \text{unload} \) is defined by recursion over the stack as before, but with one crucial difference. Instead of always returning the final result, it may also2 return a new configuration of our abstract machine, that is, a pair \( \text{N} \times \text{Stack} \):

\[
\text{unload} : \text{N} \to \text{Stack} \to (\text{N} \times \text{Stack}) \cup \text{N}
\]

\[
\text{unload} \ v \ Top = \text{inj}_1 \ v
\]

\[
\text{unload} \ v \ (\text{Right} \ v' \ stk) = \text{unload} \ v' \ (v +) \ stk
\]

\[
\text{unload} \ v \ (\text{Left} \ r \ stk) = \text{load} \ r \ (\text{Right} \ v \ stk)
\]

The other key difference arises in the definition of \( \text{load} \):

\[
\text{load} : \text{Expr} \to \text{Stack} \to (\text{N} \times \text{Stack}) \cup \text{N}
\]

\[
\text{load} \ (\text{Val} \ n) \ stk = \text{inj}_1 \ (n, stk)
\]

\[
\text{load} \ (\text{Add} \ e_1 \ e_2) \ stk = \text{load} \ e_1 \ (\text{Left} \ e_2 \ stk)
\]

Rather than calling \( \text{unload} \) upon reaching a value, it returns the current stack and the value of the leftmost leaf. Even though the function never returns an \( \text{inj}_2 \), its type is aligned with the type of \( \text{unload} \) so the definition of both functions resembles an abstract machine more closely.

Both these functions are now accepted by Agda’s termination checker as they are clearly structurally recursive. We can use both these functions to define the following evaluator3:

\[
\text{tail-rec-eval} : \text{Expr} \to \text{N}
\]

\[
\text{tail-rec-eval} \ e \ with \ load \ e \ Top \ \\
\ldots \ | \ \text{inj}_1 \ (n, stk) = \text{rec} \ (n, stk)
\]

where

\[
\text{rec} : (\text{N} \times \text{Stack}) \to \text{N}
\]

\[
\text{rec} \ (n, stk) \ with \ \text{unload} \ n \ stk
\]

\[
\ldots \ | \ \text{inj}_1 \ (n', stk') = \text{rec} \ (n', stk')
\]

\[
\ldots \ | \text{inj}_2 \ r = r
\]

Here we use \( \text{load} \) to compute the initial configuration of our machine – that is, it finds the leftmost leaf in our initial expression and its associated stack. We proceed by repeatedly calling \( \text{unload} \) until it returns a value. This version of our evaluator, however, does not pass the termination checker.

The new state, \( (n', stk') \), is not structurally smaller than the initial state \( (n, stk) \). If we work under the assumption that we

1https://github.com/carlostome/Dissection-thesis

2\( \top \) is Agda’s type of disjoint union.

3We ignore \( \text{load} \)’s impossible case, it can always be discharged with \( \bot \text{-elim} : \forall \{ X : \text{Set} \} \to \bot \to X \).

Figure 1. Traversing a tree with \( \text{load} \) and \( \text{unload} \)
We now define the following version of the tail-recursive evaluator:

\[
\text{tail-rec-eval} : \text{Expr} \rightarrow \text{bi}
\]

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

\[
\ldots | \text{inj}_1 (\_ , \text{stk}) = \text{rec} (\_ , \text{stk}) \quad \square_1
\]

where

\[
\text{rec} : (c : \mathbb{N} \times \text{Stack}) \rightarrow \text{Acc} \rightarrow \text{c} \rightarrow \mathbb{N}
\]

\[
\text{rec} (\_ , \text{stk}) (\text{acc} rs) \text{ with unload } n \text{ stk}
\]

\[
\ldots | \text{inj}_1 (n' , \text{stk}') = \text{rec} (n' , \text{stk}') (rs \quad \square_2)
\]

\[
\ldots | \text{inj}_2 r = r
\]

To complete this definition, we still need to define a suitable relation \(\_ \prec \_\) between configurations of type \(\mathbb{N} \times \text{Stack}\), prove the relation to be well-founded (\(\square_1 : \text{Acc} \prec_\triangle (\_ , \text{stk})\)), and show that the calls to \text{unload} produce ‘smaller’ states (\(\square_2 : ((n' , \text{stk}') < (n , \text{stk})\)). In the next section, we will define such a relation and prove it is well-founded.

### 3 Well-founded tree traversals

The type of configurations of our abstract machine can be seen as a variation of Huet’s zippers [1997]. The zipper associated with an expression \(e : \text{Expr}\) is pair of a (sub)expression of \(e\) and its context. As demonstrated by McBride [2008], the zippers can be generalized further to dissections, where the values to the left and right of the current subtree may have different types. It is precisely this observation that we will exploit when considering the generic tail-recursive traversals in the later sections; for now, however, we will only rely on the intuition that the configurations of our abstract machine, given by the type \(\mathbb{N} \times \text{Stack}\), are an instance of dissections, corresponding to a partially evaluated expression:

\[
\text{Config} : \text{Set}
\]

\[
\text{Config} = \mathbb{N} \times \text{Stack}
\]

These configurations, are more restrictive than dissections in general. In particular, the configurations presented in the previous section only ever denote a leaf in the input expression.

The tail-recursive evaluator, \(\text{tail-rec-eval}\), processes the leaves of the input expression in a left-to-right fashion. The leftmost leaf – that is the first leaf found after the initial call to \text{load} – is the greatest element; the rightmost leaf is the smallest. In our example expression from Section 1, we would number the leaves as follows:

This section aims to formalize the relation that orders elements of the \text{Config} type (that is, the configurations of the abstract machine) and prove it is well-founded. However, before doing so there are two central problems with our choice of \text{Config} datatype:

1. The \text{Config} datatype is too liberal. As we evaluate our input expression the configuration of our abstract machine changes constantly, but satisfies one important invariant: each configuration is a decomposition of the original input. Unless this invariant is captured, we will be hard pressed to prove the well-foundedness of any relation defined on configurations.

2. The choice of the \text{Stack} datatype, as a path from the leaf to the root is convenient to define the tail-recursive machine, but impractical when defining the coveted order relation. The top of a stack stores information about neighbouring nodes, but to compare two leaves we need global information about their positions relative to the root.

We will now address these limitations one by one. Firstly, by refining the type of \text{Config}, we will show how to capture the desired invariant (Section 3.1). Secondly, we explore a different representation of stacks, as paths from the root, that facilitates the definition of the desired order relation (Section 3.2). Finally we will define the relation over configurations, Section 3.3, and sketch the proof that it is well-founded.

#### 3.1 Invariant preserving configurations

A value of type \text{Config} denotes a leaf in our input expression. In the previous example, the following \text{Config} corresponds to the third leaf:

As we observed previously, we would like to refine the type \text{Config} to capture the invariant that execution preserves: every \text{Config} denotes a unique leaf in our input expression, or equivalently, a state of the abstract machine that computes the fold. There is one problem still: the \text{Stack} datatype stores the values of the subtrees that have been evaluated, but does not store the subtrees themselves. In the example in Figure 3, when the traversal has reached the third leaf, all the subexpressions to its left have been evaluated.

In order to record the necessary information, we redefine the \text{Stack} type as follows:

\[
\text{data Stack}^* : \text{Set where}
\]

\[
\text{Left} : \text{Expr} \rightarrow \text{Stack}^* \rightarrow \text{Stack}^*
\]
The \textbf{Right} constructor now not only stores the value \( n \), but also records the subexpression \( e \) and the proof that \( e \) evaluates to \( n \). Although we are modifying the definition of the \textbf{Stack} data type, we claim that the expression \( e \) and equality are not necessary at run-time, but only required for the proof of well-foundedness – a point we will return to in our discussion (Section 5). From now onwards, the type \textbf{Config} uses \textbf{Stack} as its right component:

\begin{equation*}
\text{Config} = \text{N} \times \text{Stack}^*
\end{equation*}

The function \texttt{unload} was previously defined by induction over the stack (Section 2), thus, it needs to be modified to work over the new type of stacks, \textbf{Stack}:

\begin{align*}
\text{unload}^* : (n : \text{N}) \rightarrow (e : \text{Expr}) \rightarrow \text{eval} e \equiv n \rightarrow \text{Stack}^* \\
& \rightarrow \text{Config} \equiv \text{N}
\end{align*}

\begin{align*}
\text{unload}^* n e q \text{Top} &= \text{inj} \_ n \\
\text{unload}^* n e q (\text{Left} \ e' \ sk) &= \text{load} e' (\text{Right} n e q \ sk) \\
\text{unload}^* n e q (\text{Right} n' e' q' \ sk) &= \text{unload}^* (n' + n) (\text{Add} e' e) (\text{cong}_2 \_ \_ \_ q' q' e q) \ sk
\end{align*}

A value of type \textbf{Config} contains enough information to recover the input expression. This is analogous to the \texttt{plug} operation on zippers:

\begin{align*}
\text{plug}_\text{i} : \text{Expr} \rightarrow \text{Stack}^* \rightarrow \text{Expr} \\
\text{plug}_\text{i} e \text{Top} &= e \\
\text{plug}_\text{i} (\text{Left} t \ sk) &= \text{plug}_\text{i} (\text{Add} e t) \ sk \\
\text{plug}_\text{i} (\text{Right} t \ sk) &= \text{plug}_\text{i} (\text{Add} t e) \ sk
\end{align*}

Any two terms of type \textbf{Config} may still represent states of a fold over two entirely different expressions. As we aim to define an order relation comparing configurations during the fold of the input expression, we need to ensure that we only ever compare configurations within the same expression. We can \textit{statically} enforce such requirement by defining a new wrapper data type over \textbf{Config} that records the original input expression:

\begin{align*}
data \text{Config}_\text{i} (e : \text{Expr}) : \text{Set} \text{where} \\
\_ \_ : (e : \text{Config}) \rightarrow \text{plug}_{\text{i}} e \equiv e \rightarrow \text{Config}_\text{i} e
\end{align*}

For a given expression \( e : \text{Expr} \), any two terms of type \textbf{Config} are configurations of the same abstract machine during the tail-recursive fold over the expression \( e \).

### 3.2 Up and down configurations

Next, we would like to formalize the left-to-right order on the configurations of our abstract machine. The \textbf{Stack} in the \textbf{Config} represents a path upwards, from the leaf to the root of the input expression. This is useful when navigating to neighbouring nodes, but makes it harder to compare the relative positions of two configurations. We now consider the value of \textbf{Config} corresponding to leaves with numbers 3 and 4 in our running example:

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3}
\caption{Example: Configuration of leaf number 3}
\end{figure}

\begin{align*}
1, \quad \begin{bmatrix}
\text{Right 7, Right 3, Left} & \text{Add} \\
\text{Val 2, Val 0}
\end{bmatrix}
\end{align*}

\begin{align*}
7, \quad \begin{bmatrix}
\text{Left} & \text{Val 1, Right 3, Left} & \text{Add} \\
\text{Val 2, Val 0}
\end{bmatrix}
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4}
\caption{Comparison of configurations for leaves 3 and 4}
\end{figure}

The natural way to define the desired order relation is by induction over the \textbf{Stack}. However, there is a problem. The first element of both stacks does not provide us with sufficient information to decide which position is ‘smaller.’ The top of the stack only stores information about the location of the leaf with respect to its parent node. This kind of \textit{local} information cannot be used to decide which one of the leaves is located in a position further to the right in the original input expression.

Instead, we would like to compare the \textit{last} elements of both stacks. The common suffix of the stacks shows that both positions are in the left subtree of the root. Once these paths – read from right to left – diverge, we have found the exact node \texttt{Add} where one of the positions is in the left subtree and the other in the right.

When comparing two \textbf{Stacks}, we therefore want to consider them as paths \textit{from the root}. Fortunately, this observation does not require us to change our definition of the \textbf{Stack} type; instead, we can define a variant of the \texttt{plug} function that interprets our contexts top-down rather than bottom-up:

\begin{align*}
\text{plug}_\text{j} : \text{Expr} \rightarrow \text{Stack}^* \rightarrow \text{Expr} \\
\text{plug}_\text{j} e \text{Top} &= e \\
\text{plug}_\text{j} (\text{Left} t \ sk) &= \text{Add} (\text{plug}_\text{j} e \ sk) t \\
\text{plug}_\text{j} (\text{Right} t \ sk) &= \text{plug}_\text{j} (\text{Add} t e) \ sk
\end{align*}

\begin{align*}
\text{plug}_{\text{i}} (n, sk) &= \text{plug}_\text{j} (\text{Val} n) \ sk
\end{align*}
We can convert freely between these two interpretations by reversing the stack. Furthermore, this conversion satisfies the plug↓-to-plug↑ property, relating the two variants of plug:

\[
\text{convert} : \text{Config} \rightarrow \text{Config}
\]

\[
\text{convert}(s, s) = (s, \text{reverse } s)
\]

\[
\text{plug}↑\text{-to-plug}↓ : \forall (e : \text{Config}) \rightarrow \text{plug}↓_C e \equiv \text{plug}↑_C (\text{convert } e)
\]

As before, we can create a wrapper around Config that enforces that our Config denotes a leaf in the input expression:

\[
\text{data Config}_↓ (e : \text{Expr}) : \text{Set where}
\]

\[
_\_\_ : (e : \text{Config}) \rightarrow \text{plug}↓_C e \equiv e \rightarrow \text{Config}_↓ e
\]

As a corollary of the plug↓-to-plug↑ property, we can define a pair of functions to switch between Config↓ and Config↑:

\[
\text{Config}_↓\text{-to-Config}_↑ : (e : \text{Expr}) \rightarrow \text{Config}_↓ e \rightarrow \text{Config}_↑ e
\]

\[
\text{Config}_↑\text{-to-Config}_↓ : (e : \text{Expr}) \rightarrow \text{Config}_↑ e \rightarrow \text{Config}_↓ e
\]

3.3 Ordering configurations

Finally, we can define the ordering relation over values of type Config↓. Even if the Config↑ is still used during execution of our tail-recursive evaluator, the Config↓ type will be used to prove its termination.

The _\_\_\_<_ type defined below relates two configurations of type Config↓ e, that is, two states of the abstract machine evaluating the input expression e:

\[
\text{data } _\_\_\_< e_1 e_2 : (e : \text{Expr}) \rightarrow \text{Config}_↓ e_1 \rightarrow \text{Config}_↓ e_2 \rightarrow \text{Set where}
\]

\[
\text{<StepR} : \text{=} \text{<}_1 ((t_1, s_1), \ldots, t_1) < ((t_2, s_2), \ldots)
\]

\[
\text{<StepL} : \text{=} \text{<}_1 ((t_1, s_1), \ldots, t_1) < ((t_2, s_2), \ldots)
\]

\[
\text{<Base} : (eqs) : \text{Add } e_1 e_2 \equiv \text{Add } e_1 (\text{plug}↓_C t_1 s_1)
\]

Despite the apparent complexity, the relation is straightforward. The constructors <StepR and <StepL cover the inductive cases, consuming the shared path from the root. When the paths diverge, the <Base constructor states that the positions in the right subtree are ‘smarter’ than those in the left subtree.

Now we turn into showing that the relation is well-founded. We sketch the proof below:

\[
\text{< WF} : \forall (e : \text{Expr}) \rightarrow \text{Well-founded } (_\_\_\_< e_1 e_2)
\]

\[
\text{< WF } x = \text{acc } (\text{aux } x)
\]

where

\[
\text{aux} : \forall (e : \text{Expr}) (x y : \text{Config}_↓ e) \rightarrow \_\_\_\_< (x y) x y
\]

The proof follows the standard schema of most proofs of well-foundedness. It uses an auxiliary function, aux, that proves every configuration smaller than x is accessible.

\[\text{aux} = \ldots\]

The proof proceeds initially by induction over our relation. The inductive cases, corresponding to the <StepR and <StepL constructors, recurse on the relation. In the base case, <Base, we cannot recurse further on the relation. We then proceed by recursing over the original expression e1 without the type index, the subexpressions to the left e1 and right e2 are not syntactically related thus a recursive call is not possible. This step in the proof relies on only comparing configurations arising from traversing the same initial expression e.

3.4 A terminating and correct tail-recursive evaluator

We now have almost all the definitions in place to revise our tail-recursive fold, tail-rec-eval. However, we are missing one essential ingredient: we still need to show that the configuration decreases after a call to the unload↑ function.

Unfortunately, the function unload↑ and the relation that we have defined work on ‘different’ versions of the Stack: the relation compares stacks top-down; the unload↑ function manipulates stacks bottom-up. Furthermore, the function unload↑ as defined previously manipulates elements of the Config type directly, with no further type-level constraints relating these to the original input expression.

In the remainder of this section, we will reconcile these differences, complete the definition of our tail-recursive evaluator and finally prove its correctness.

Decreasing recursive calls To define our tail-recursive evaluator, we will begin by defining an auxiliary step function that performs a single step of computation. We will define the desired evaluator by iterating the step function, proving that it decreases in each iteration.

The step function calls unload↑ to produce a new configuration, if it exists. If the unload↑ function returns a natural number, inj2 v, the entire input tree has been processed and the function terminates:

\[
\text{step} : (e : \text{Expr}) \rightarrow \text{Config}_↓ e \rightarrow \text{Config}_↓ e \equiv N
\]

\[
\text{step } e (n, stk, eq) \text{ with unload}↑ n (\text{Val } n) \text{ refl } stk
\]

\[
\text{... } \text{inj}_1 (n', stk) = \text{inj}_1 ((n', stk), \ldots)
\]

\[
\text{... } \text{inj}_2 v = \text{inj}_2 v
\]

We have omitted the second component of the result returned in the first branch, corresponding to a proof that plug↑_C (n', stk) \equiv e. The crucial lemma that we need to show to complete this proof, demonstrates that the unload↑ function respects our invariant:

\[
\text{unload}↑\text{-plug}↓ : \forall (v : N) (e : \text{Expr}) (eqsv \equiv \equiv s) (s : \text{Stack}^\uparrow) (e : \text{Config}) \rightarrow \text{unload}↑ n s eqsv \equiv \equiv e
\]

\[
\Rightarrow \forall (e' : \text{Expr}) \rightarrow \text{plug}↓_C e \equiv e' \rightarrow \text{plug}↑_C e \equiv e'
\]

Finally, we can define the theorem stating that the step function always returns a smaller configuration:
The proof is done by induction over the stack supported; the step induction over the accessibility predicate:

$$\text{rec-correct} : \forall (e : \text{Expr}) \to (c : \text{Config}_a e)$$
$$\to (ac : \text{Acc} (\langle e, \text{steps} \rangle \langle \text{Config}_a \text{to-Config}_b e \rangle))$$
$$\to \text{eval} e \equiv \text{rec} e \equiv ac$$

At this point, we still need to prove the step-correct lemma that it is repeatedly applied. As the step function is defined as a wrapper around the unload function, it suffices to prove the following property of unload*: 

$$\text{unload}^* : \forall (n : \mathbb{N}) \langle e : \text{Expr} \rangle (eq : \text{eval} e \equiv n) \langle s : \text{Stack}^* \rangle$$
$$\to (\text{eval} (\text{plug}_n e s) \equiv m)$$

This proof follows immediately by induction over s : Stack*. The main correctness theorem now states that eval and tail-rec-eval are equal for all inputs:

$$\text{correctness} : \forall (e : \text{Expr}) \to \text{eval} e \equiv \text{tail-rec-eval} e$$

4 A generic tail-recursive traversal

The previous section showed how to prove that our handwritten tail-recursive evaluation function was both terminating and equal to our original evaluator. In this section, we will show how we can generalize this construction to compute a tail-recursive equivalent of any function that can be written as a fold over a simple algebraic datatype. In particular, we generalize the following:

- The kind of datatypes, and their associated fold, that our tail-recursive evaluator supports, Section 4.1.
- The type of configurations of the abstract machine that computes the generic fold, Sections 4.2 and 4.3.
- The functions load and unload such that they work over our choice of generic representation, Section 4.4.
- The ’smaller than’ relation to handle generic configurations, and its well-foundedness proof, Section 4.5.
- The tail-recursive evaluator, Section 4.6.
- The proof that the generic tail-recursive function is correct, Section 4.7.

Before we can define any such datatype generic constructions, however, we need to fix our universe of discourse.

4.1 The regular universe

In a dependently typed programming language such as Agda, we can represent a collection of types closed under certain operations as a universe [Altenkirch and McBride 2003; Martin-Löf 1984], that is, a data type $$\text{U} : \text{Set}$$ describing the inhabitants of our universe together with its semantics.
el : U → Set, mapping each element of U to its corresponding type. We have chosen the following universe of regular types [Morris et al. 2006; Noort et al. 2008]:

\[
data Reg : Set_1 where
\]
\[
\begin{align*}
\emptyset & : Reg \\
\top & : Reg \\
\bot & : Reg \\
A & : (A : Set) → Reg \\
\odot, \ominus & : (R Q : Reg) → Reg \\
\odot, \ominus & : (R Q : Reg) → Reg \\
\end{align*}
\]

Types in this universe are formed from the empty type (∅), unit type (∪), and constant types (K A); the 1 constructor is used to refer to recursive subtrees. Finally, the universe is closed under both coproducts (\(\odot\)) and products (\(\ominus\)). We could represent the \(\text{pattern}\) functor corresponding to the \(\text{Expr}\) type in this universe as follows:

\[
\text{exprF} : \text{Reg} \\
\text{exprF} = K \odot (\bot \odot 1)
\]

Note that as the constant functor \(K\) takes an arbitrary type \(A\) as its argument, the entire datatype lives in \(\text{Set}_1\). This could easily be remedied by stratifying this universe explicitly and parametrising our development by a base universe.

We can interpret the inhabitants of \(\text{Reg}\) as a functor of type \(\text{Set} \to \text{Set}\):

\[
\begin{align*}
\lbrack \bot \rbrack &: \text{Reg} \to \text{Set} \\
\lbrack \top \rbrack &: \text{Set} \\
\lbrack \bot \rbrack &: \text{Set} \\
\lbrack \bot \rbrack &: \text{Set} \\
\text{inj} &: \text{Reg}_1 \to \text{Set} \\
\text{inj} &: \text{Reg}_1 \to \text{Set} \\
\text{inj} &: \text{Reg}_1 \to \text{Set} \\
\end{align*}
\]

To show that this interpretation is indeed functorial, we define the following fmap operation:

\[
\begin{align*}
fmap : (R : \text{Reg}) → (X → Y) → \lbrack R \rbrack X → \lbrack R \rbrack Y \\
fmap \emptyset f () &= \bot \\
fmap \top f tt &= tt \\
fmap f x &= f x \\
fmap (A f x) &= x \\
fmap (R \odot Q) f (\text{inj}_1 x) &= \text{inj}_1 (\text{fmap} \ R f x) \\
fmap (R \odot Q) f (\text{inj}_2 y) &= \text{inj}_2 (\text{fmap} \ Q f y) \\
fmap (R \odot Q) f (x, y) &= \text{fmap} \ R f x, \text{fmap} \ Q f y \\
\end{align*}
\]

Finally, we can tie the recursive knot, taking the least fixpoint of the functor associated with the elements of our universe:

\[
data \mu (R : \text{Reg}) : \text{Set} where
\]
\[
\text{In} : \lbrack R \rbrack (\mu R) → \mu R
\]

Next, we can define a generic fold, or catamorphism, to work on the inhabitants of the regular universe. For each code \(R : \text{Reg}\), the catamorph function takes an algebra of type \(\lbrack R \rbrack X → X\) as argument. This algebra assigns semantics to the ‘constructors’ of \(R\). Folding over a tree of type \(\mu R\) corresponds to recursively folding over each subtree and assembling the results using the argument algebra:

\[
\text{cata} : (R : \text{Reg}) → (\lbrack R \rbrack X → X) → \mu R → X
\]

Unfortunately, Agda’s termination checker does not accept this definition. The problem, once again, is that the recursive calls to \(\text{cata}\) are not made to structurally smaller trees, but rather \(\text{cata}\) is passed as an argument to the higher-order function \(\text{fmap}\).

To address this, we fuse the \(\text{fmap}\) and \(\text{cata}\) functions into a single map-fold function [Norell 2008]:

\[
\begin{align*}
\text{map-fold} : (R Q : \text{Reg}) → (\lbrack R \rbrack X → X) → \lbrack R \rbrack (\mu Q) → \lbrack R \rbrack X \\
\text{map-fold} ∅ &= \psi () \\
\text{map-fold} \bot &= ∃(\text{tt}) \\
\text{map-fold} \top &= \psi (\text{map-fold} \ Y Q x) \\
\text{map-fold} (K A) &= \psi x \\
\text{map-fold} (R \odot Q) &= \text{map-fold} \ R \text{R} \psi x \\
\text{map-fold} (R \odot Q) &= \text{map-fold} \ Q \psi y \\
\end{align*}
\]

We can now define \(\text{cata}\) in terms of \(\text{map-fold}\) as follows:

\[
\text{cata} : (R : \text{Reg}) → (\lbrack R \rbrack X → X) → \mu R → X
\]

This definition is indeed accepted by Agda’s termination checker.

Example We can now revisit our example evaluator from the introduction. To define the evaluator using the generic catamorph function, we instantiate the catamorphism to work on the expressions and pass the desired algebra:

\[
\begin{align*}
\text{eval} & : \mu \text{exprF} → \mathbb{N} \\
\text{eval} &= \text{cata} \ (\text{exprF}) \\
\phi &= \left\{ \begin{array}{ll}
\text{eval} &= \mathbb{N} \\
\phi (\text{inj}_1 n) &= n \\
\phi (\text{inj}_2 (n, n')) &= n + n'
\end{array} \right.
\end{align*}
\]

In the remainder of this paper, we will develop an alternate traversal that maps any algebra to a tail-recursive function that is guaranteed to terminate and produce the same result as the corresponding call to \(\text{cata}\).

4.2 Dissection As we mentioned in the previous section, the configurations of our abstract machine from the introduction are instances of McBride’s dissections [2008]. We briefly recap this construction, showing how to calculate the type of abstract machine configurations for any type in our universe. The key definition, \(\nabla\), computes a bifunctor for each element of our universe:

\[
\begin{align*}
\nabla : (R : \text{Reg}) → (\text{Set} → \text{Set} → \text{Set}) \\
\nabla ∅ &= X Y = \bot \\
\nabla \bot &= X Y = \bot \\
\nabla \top &= X Y = \top \\
\nabla (K A) &= X Y = ∃(\text{tt}) \\
\nabla (R \odot Q) &= \nabla (R \nabla X Y) ⊕ \nabla Q X Y \\
\nabla (R \odot Q) &= \nabla (\nabla R X Y) ⊕ \nabla Q X Y
\end{align*}
\]
This operation generalizes the zippers, by defining a bifunctor \( \nabla R X Y \). You may find it useful to think of the special case, \( \nabla R X Y (\mu R) \) as a configuration of an abstract machine traversing a tree of type \( \mu R \) to produce a result of type \( X \).

The last clause of the definition of \( \nabla \) is of particular interest: to \textit{dissect} a product, we either \textit{dissect} the left component pairing it with the second component interpreted over the second variable \( Y \); or we \textit{dissect} the second component and pair it with the first interpreted over \( X \).

A \textit{dissection} is formally defined as the pair of the one-hole context and the missing value that can fill the context.

\[
\mathcal{D} : (R : \text{Reg}) \rightarrow (X Y : \text{Set}) \rightarrow \text{Set}
\]

\[
\mathcal{D} R X Y = \nabla R X Y Y
\]

We can reconstruct Huët’s zipper for generic trees of type \( \mu R \) by instantiating both \( X \) and \( Y \) to \( \mu R \).

Given a \textit{dissection}, we can define a \textit{plug} operation that assembles the context and current value in focus to produce a value of type \( \mu R \):

\[
\text{plug} : (R : \text{Reg}) \rightarrow (X Y : \text{Set}) \rightarrow \mathcal{D} R X Y Y \rightarrow \mu R
\]

In the last clause of the definition, the \textit{dissection} is over the right component of the pair leaving a value \( r : \mu R \) to the left. In that case, it is only possible to reconstruct a value of type \( \mu R \) if we have a function \( \eta \) to recover \( Ys \) from \( Xs \).

In order to ease things later, we bundle a \textit{dissection} together with the functor to which it \textit{plugs} as a type-indexed type.

\[
\text{data} \mathcal{D}_X (R : \text{Reg}) (X Y : \text{Set}) \eta : X \rightarrow Y \mapsto \{ \eta \} Y : \text{Set} \text{ where } \\
\_\_ : (d : \mathcal{D} R X Y) \rightarrow \text{plug} R \eta d \equiv \mathcal{D}_X R X Y \eta
\]

### 4.3 Generic configurations

While the \textit{dissection} computes the bifunctor \textit{underlying} our configurations, we still need to take a fixpoint of this bifunctor. Each configuration consists of a list of \textit{dissections} and the current subtree in focus. To the left of the current subtree in focus, we store the partial results arising from the subtrees that we have already processed; on the right, we store the subtrees that still need to be visited.

As we did for the \textit{Stack}+ datatype from the introduction, we also choose to store the original subtrees that have been visited and their corresponding correctness proofs:

\[
\text{record} \text{Computed} (R : \text{Reg}) (X : \text{Set}) (\psi : [R] X \rightarrow X) : \text{Set} \text{ where } \\
\text{constructor } \_\_\_ : \\
\text{field} \text{Tree} : \mu R
\]

A \textit{stack} is a list of \textit{dissections}. To the left we have the \textit{Computed} results; to the right, we have the subtrees of type \( \mu R \). Note that the \textit{Stack}+ datatype is parametrised by the algebra \( \psi \), as the \textit{Proof} field of the \textit{Computed} record refers to it.

As we saw in Section 3.5, we can define two different \textit{plug} operations on these stacks:

\[
\text{plug} : (R : \text{Reg}) \rightarrow (X Y : \text{Set}) \rightarrow \text{List} \nabla R \psi \rightarrow \mu R
\]

\[
\text{plug} : (R : \text{Reg}) \rightarrow \text{List} \nabla R \psi \rightarrow \mu R
\]

Both functions use the projection, \textit{Computed.Tree}, as an argument to \textit{plug} to extract the subtrees that have already been processed.

To define the configurations of our abstract machine, we are interested in \textit{any} path through our initial input, but want to restrict ourselves to those paths that lead to a leaf. But what constitutes a leaf in this generic setting?

To describe leaves, we introduce the following predicate \textit{NonRec}, stating when a tree of type \( [R] X \) does not refer to the variable \( X \), that will be used to represent recursive subtrees:

\[
\text{data} \text{NonRec} : (R : \text{Reg}) \rightarrow [R] X \rightarrow \text{Set} \text{ where } \\
\text{NonRec} : \text{Reg} \rightarrow \text{NonRec} t \text{ if } t \text{ is a leaf } \text{ or } \text{NonRec} \psi \rightarrow \text{NonRec} \text{ Reg} t \psi
\]

As an example, in the pattern functor for the \textit{Expr} type, \( K \otimes (I \otimes I) \), terms built using the left injection are non-recursive:

\[
\text{Val-NonRec} : \psi (n : I) \rightarrow \text{NonRec} (K \otimes (I \otimes I) \text{ inj}_1 n)
\]

This corresponds to the idea that the constructor \textit{Val} is a leaf in a tree of type \textit{Expr}.

On the other hand, we cannot prove the predicate \textit{NonRec} for terms using the right injection. The occurrences of recursive positions disallow us from framing the proof (The type \textit{NonRec} does not have a constructor such as \textit{NonRec-1} : \( x : X \rightarrow \text{NonRec} 1 x \)).
This example also shows how ‘generic’ leaves can be recursive. As long as the recursion only happens in the functor layer (code \(\oplus\)) and not in the fixpoint level (code \(\iota\)).

Crucially, any non-recursive subtree is independent of \(X\) as is exhibited by the following coercion function:

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

Whose definition is not worth including as it follows directly by induction over the predicate.

We can now define the notion of leaf generically, as a substructure without recursive subtrees:

\[
\text{Leaf} : \text{Reg} \to \text{Set} \to \text{Set}
\]

\[
\text{Leaf} R X = \Sigma (\{ [R] X \}) (\text{NonRec} R)
\]

Just as we saw previously, a configuration is now given by the current leaf in focus and the stack, given by a dissection, storing partial results and unprocessed subtrees:

\[
\text{Config}^G : (R : \text{Reg}) \to (X : \text{Set}) \to (\psi : [R] X \to X) \to \text{Set}
\]

\[
\text{Config}^G R X \psi = \text{Leaf} R X \times \text{Stack}^G R X \psi
\]

Finally, we can recompute the original tree using a plug function as before:

\[
\text{plugC}-\mu_l : (R : \text{Reg}) \{ \psi : [R] X \to X \}
\to \text{Config}^G R X \psi \to \mu R \to \text{Set}
\]

\[
\text{plugC}-\mu_l R ((l, \text{isl}), s) t = \text{plug}-\mu_l R (\text{In} (\text{coerce} l \text{isl})) s t
\]

Note that the \(\text{coerce}\) function is used to embed a leaf into a larger tree. A similar function can be defined for the ‘bottom-up’ zippers, that work on a reversed stack.

4.4 One step of a catamorphism

In order to write a tail-recursive catamorphism, we start by defining the generic operations that correspond to the functions \(\text{load}\) and \(\text{unload}\) given in the introduction (Section 2).

\textbf{Load}  The function \(\text{load}^G\) traverses the input term to find its leftmost leaf. Any other subtrees the \(\text{load}^G\) function encounters are stored on the stack. Once the \(\text{load}^G\) function encounters a constructor without subtrees, it is has found the desired leaf.

We write \(\text{load}^G\) by appealing to an ancillary definition \(\text{first-cps}\), that uses continuation-passing style to keep the definition tail-recursive and obviously structurally recursive. If we were to try to define \(\text{load}^G\) by recursion directly, we would need to find the leftmost subtree and recurse on it – but this subtree may not be obviously syntactically smaller.

The type of our \(\text{first-cps}\) function is daunting at first:

\[
\text{first-cps} : ([R] : \text{Reg}) \{ \psi : [Q] X \to X \}
\to ([R] (\mu Q))
\to (\forall R (\text{Computed} Q X \psi) (\mu Q) \to (\forall Q (\text{Computed} Q X \psi) (\mu Q)))
\to (\text{Leaf} R X \to \text{Stack}^G Q X \psi \to \text{Config}^G Q X \psi \uplus X)
\to \text{Stack}^G Q X \psi
\to \text{Config}^G Q X \psi \uplus X
\]

The first two arguments are codes of type \(\text{Reg}\). The code \(Q\) represents the datatype for which we are defining a traversal; the code \(R\) is the code on which we pattern match. In the initial call to \(\text{first-cps}\) these two codes will be equal. As we define our function, we pattern match on \(R\), recursing over the codes in (nested) pairs or sums – yet we still want to remember the original code for our data type, \(Q\).

The next argument of type \([R] (\mu Q)\) is the data we aim to traverse. Note that the ‘outermost’ layer is of type \(R\), but the recursive subtrees are of type \(\mu Q\). The next two arguments are two continuations: the first is used to gradually build the dissection of \(R\); the second continues on another branch once one of the leaves have been reached. The last argument of type \(\text{Stack}^G Q X \psi\) is the current stack. The entire function will compute the initial configuration of our machine of type \(\text{Config}^G Q X \psi\):

\[
\text{load}^G : (R : \text{Reg}) \{ \psi : [R] X \to X \} \to \mu R
\to \text{Stack}^G R X \psi \to \text{Config}^G R X \psi \uplus X
\]

\[
\text{load}^G R (\text{In} l) s = \text{first-cps} R R s \text{id} (\lambda l \rightarrow \text{inj}_l \circ \_\_\_\_ \_ \_ s)
\]

We shall fill the definition of \(\text{first-cps}\) by cases. The clauses for the base cases are as expected. In \(\emptyset\) there is nothing to be done. The \(\text{I}\) and \(\text{K}\) codes consist of applying the second continuation to the tree and the stack.

\[
\text{first-cps} \emptyset \emptyset () \_ = \text{first-cps} \text{I} Q X s = \_ f (tt, \text{NonRec-I} s)
\]

\[
\text{first-cps} (\text{K} A) Q X s = \_ f (x, \text{NonRec-K A} s)
\]

The recursive case, constructor \(\text{I}\), corresponds to the occurrence of a subtree. The function \(\text{first-cps}\) is recursively called over that subtree with the stack incremented by a new element that corresponds to the dissection of the functor layer up to that point. The second continuation is replaced with the initial one.

\[
\text{first-cps} \text{I} (Q X s) = \text{first-cps} Q Q X s \text{id} (\lambda l \rightarrow \text{inj}_l \circ \_\_\_\_ \_ \_ s) (k tt \_ s)
\]

In the coproduct, both cases are similar, just having to account for the use of different constructors in the continuations.

\[
\text{first-cps} (R \oplus Q) P (\text{inj}_1 x) s = \text{first-cps} R P x (k \circ \text{inj}_1) \_ s
\]

\[
\text{first-cps} (R \oplus Q) P (\text{inj}_2 y) s = \text{first-cps} Q P y (k \circ \text{inj}_2) \_ s
\]

The interesting clause is the one that deals with the product.

First the function \(\text{first-cps}\) is recursively called on the left component of the pair trying to find a subtree to recurse over. However, it may be the case that there are no subtrees at all, thus it is passed as the first continuation a call to \(\text{first-cps}\) over the right component of the product. In case the continuation fails to find a subtree, it returns the leaf as it is.

\[
\text{first-cps} (R \oplus Q) P (r, q) s = \text{first-cps} R P r (k \circ \text{inj}_1 \circ (q)) \_ s
\]

\[
\text{first-cps} (R \oplus Q) P (r, q) s = \text{first-cps} Q P q (k \circ \text{inj}_2 \circ \_\_\_\_ (\text{coerce} l \text{isl})) \_ s
\]

\[
\text{cont} (l, \text{isl}) = f (l, 1) (\text{NonRec-} \oplus \text{R Q l} \text{isl} \text{isl})
\]

\[\text{As in the introduction, we use a sum type }\text{ by to align its type with that of }\text{unload}^G.\]
Unload Armed with load we turn our attention to unload. First of all, it is necessary to define an auxiliary function, right, that given a dissection and a value (of the type of the left variables), either finds a dissection $D R X Y$ or it shows that there are no occurrences of the variable left. In the latter case, it returns the functor interpreted over $Y$, $\llbracket R \rrbracket X$.

right $: (R : \text{Reg}) \to \nabla R X Y X \to \llbracket R \rrbracket X \otimes D R X Y$

Its definition is simply by induction over the code $R$, with the special case of the product that needs to use another ancillary definition to look for the leftmost occurrence of the variable position within $\llbracket R \rrbracket X$.

The function unload is defined by induction over the stack. If the stack is empty the job is done and a final value returned. In case the stack has at least one dissection in its head, the function right is called to check whether there are any more holes left. If there are none, a recursive call to unload is dispatched, otherwise, if there is still a subtree to be processed the function load is called.

unload $: (R : \text{Reg})$

$\to (\psi : \llbracket R \rrbracket X X X \to X)$

$\to ((t : \mu R) \to (s : X) \to \text{cata } R \psi t \equiv x)$

$\to \text{Stack} R X \psi X$

$\to \text{Config} R X \psi \forall X$

unload $\psi t x eq \equiv \text{inj}_1 x$

unload $\psi t x eq (h \cdot hs)$ with right $h(t, x, eq)$

unload $\psi t x eq (h \cdot hs) \mid \text{inj}_r \text{r with compute } RR r$

When the function right returns a injj \_r it means that there are not any subtrees left in the dissection. If we take a closer look, the type of the $r$ in injj \_r is $\llbracket R \rrbracket (\text{Computed } R X \psi)$. The functor $\llbracket R \rrbracket$ is storing at its variable positions both values, subtrees and proofs.

However, what is needed for the recursive call is first, the functor interpreted over values, $\llbracket R \rrbracket X$, in order to apply the algebra; second, the functor interpreted over subtrees, $\llbracket R \rrbracket (\mu R)$, to keep the original subtree where the value came from; third, the proof that the value equals to applying a cata over the subtree. The function compute masses $r$ to adapt the arguments for the recursive call to unload.

4.5 Relation over generic configurations

We can engineer a well-founded relation over elements of type Config $t$, for some concrete tree $t : \mu R$, by explicitly separating the functorial layer from the recursive layer induced by the fixed point. At the functor level, we impose the order over dissections of $R$, while at the fixed point level we define the order by induction over the stacks.

To reduce clutter in the definition, we give a non-type-indexed relation over terms of type Config. We can later use the same technique as in Section 3.4 to recover a fully type-indexed relation over elements of type Config $t$ by requiring that the zippers respect the invariant, plugC-$\mu_R c \equiv t$.

The relation is defined by induction over the Stack part of the zippers as follows.

data $\_ < c_\mu : \text{Config} R X \psi \rightarrow \text{Config} R X \psi \rightarrow \text{Set}$ where

Step $: (t_1, s_1) < (t_2, s_2)$ $\rightarrow (t_1, h) \sim s_1 < (t_2, h) \sim s_2$

Base $: \text{plugC}\_\mu_R (t_1, s_1) \equiv e_1 \rightarrow \text{plugC}\_\mu_R (t_2, s_2) \equiv e_1$

$\rightarrow (h_1, e_1) < \psi (h_2, e_2) \rightarrow (t_1, h_1) : s_1 < (t_2, h_2) : s_2$

This relation has two constructors:

• The Step constructor covers the inductive case. When the head of both stacks is the same, i.e., both Configs share the same prefix, it recurses directly on tail of both stacks.

• The constructor Base accounts for the case when the head of the stacks is different. This means that the paths given by the configuration denotes different subtrees of the same node. In that case, the relation we are defining relies on an auxiliary relation $\_ < c_\psi$, that orders dissections of type $D R (\text{Computed } R X \psi) (\mu R)$.

We can define this relation on dissections directly, without having to consider the recursive nature of our datatypes. We define the required relation over dissections interpreted on any sets $X$ and $Y$ as follows:

data $\_ < c_\psi : (R : \text{Reg}) \rightarrow D R X Y \rightarrow D R X Y \rightarrow \text{Set}$ where

step$_\psi$ $: \_ R \_ (r_1, t_1) \rightarrow \_ R \_ (r_1, t_2)$

$\rightarrow \_ R \_ (\text{inj} (r_1, t_1) \psi (\text{inj} (r_1, t_2)$

step$_\psi$ $: \_ \psi q (q_1, t_1) \rightarrow \_ \psi (q_1, t_2)$

$\rightarrow \_ R \_ (\text{inj} (r_1, q_1, t_1) \psi (\text{inj} (r_1, q_1, t_2)$

step$_\psi$ $: \_ R \_ (dr, t_1) \rightarrow \_ R \_ (dr', t_2)$

$\rightarrow \_ R \_ (\text{inj} (dr, q_1, t_1) \psi (\text{inj} (dr', q_1, t_2)$

The idea is that we order the elements of a dissection in a left-to-right fashion. All the constructors except for base$_\psi$ simply follow the structure of the dissection. To define the base case, base$_\psi$, recall that the dissection of the product of two functors, $R \otimes Q$, has two possible values. It is either a term of type $\nabla R X Y \times \psi Q Y$ like injj \_r or a term of type $\llbracket R \rrbracket X X \nabla Q X Y$ like injj \_r. The former denotes a position in the left component of the pair while the latter denotes a position in the right component. The base$_\psi$ constructor states that positions in right are smaller than those in the left.

This completes the order relation on configurations; we still need to prove our relation is well-founded. To prove this, we write a type-indexed version of each relation. The first relation, $\_ < c_\psi$, has to be type-indexed by the tree of type $\mu R$ to which both zipper recursively plug through plugC-$\mu_R$. The auxiliary relation, $\_ < c_\psi$, needs to be type-indexed by the functor of type $\llbracket R \rrbracket X$ to which both dissections plug.

data $\_ < c_\psi : \llbracket X \rightarrow Y \rrbracket : (R : \text{Reg}) \rightarrow (t : \llbracket R \rrbracket Y) \rightarrow D R X X \rightarrow \text{Set}$ where
Theorem 5.25: For any value $c$ of type $\text{Config}$, the traversal $\text{unload}^G$ delivers a smaller value than $\text{unload}$.

Proof: Consider two values $c_1$ and $c_2$ of type $\text{Config}$, with $c_1 \ll c_2$. We need to show that $\text{unload}^G(c_1) \ll \text{unload}^G(c_2)$. Without loss of generality, we can assume that $c_1$ and $c_2$ have the same structure, with $c_1$ being smaller than $c_2$.

Let $\text{unload}^G(c_1) = c_3$ and $\text{unload}^G(c_2) = c_4$. We need to show that $c_3 \ll c_4$. This can be proved by induction on the structure of $c_1$ and $c_2$. For the base case, where $c_1$ and $c_2$ are leaf values, we can directly observe that $c_3 \ll c_4$. For the recursive case, we use the induction hypothesis and the fact that $c_1 \ll c_2$ to show that $\text{unload}^G(c_1) \ll \text{unload}^G(c_2)$.

Finally, we can write the tail-recursive machine $\text{tail-rec-cata}$, as the combination of an auxiliary recursor over the accessibility predicate and a top-level function that initiates the computation with suitable arguments.

Theorem 5.26: The tail-recursive machine $\text{tail-rec-cata}$ correctly computes the generic construction $\text{G}$.

Proof: We prove this theorem by induction on the structure of $\text{G}$. The base case is trivial, as $\text{tail-rec-cata}$ simply returns the result of $\text{G}'. For the recursive case, we use the induction hypothesis and the fact that $\text{unload}^G$ is a correct reduction rule to show that $\text{tail-rec-cata} \text{G} = \text{G}'$. 

4.4 Example

To conclude, we show how to generically implement the example from the introduction (Section 1), and how the generic construction gives us a correct tail-recursive machine for free.

First, we recapture the pattern functor underlying the type $\text{Expr}$:

$\text{Expr} : \text{Reg}$ $\text{Expr}F = \text{K N } \otimes (l \otimes l)$

The $\text{Expr}$ type is then isomorphic to tying the knot over $\text{Expr}F$:

$\text{Expr} : \text{Set}$ $\text{Expr}G = \mu \text{Expr}$
The function \( \text{eval} \) is equivalent to instantiating the \textit{catamorphism} with an appropriate algebra:

\[
\psi : \text{expr}\ F \rightarrow N \\
\psi (\text{inj}_1 v) = n \\
\psi (\text{inj}_2 (e_1, e_2)) = e_1 + e_2 \\
\text{eval} : \text{Exp}E \rightarrow N \\
\text{eval} = \text{cata expf}\ \psi
\]

Finally, a tail-recursive machine equivalent to the one we derived in Section 3.4, \textit{tail-rec-eval}, is given by:

\[
\text{tail-rec-eval}^G : \text{Exp}E \rightarrow N \\
\text{tail-rec-eval}^G = \text{tail-rec-cata expf}\ \psi
\]

## 5 Discussion

There is a long tradition of calculating abstract machines from an evaluator, dating back as far as early work on the abstract machines for the evaluation of lambda calculus terms [Landin 1964]. In particular, Danvy [Ager et al. 2003; Danvy 2009] has published many examples showing how abstract machines arise from defunctionalizing an interpreter written in continuation-passing style. This work in turn, inspired McBride’s work on dissections [2008], that defines the key constructions on which this paper builds. McBride’s work, however, does not give a proof of termination or correctness.

The universe of regular types used in this paper is somewhat restricted: we cannot represent mutually recursive types [Yakushev et al. 2009], nested data types [Bird and Meertens 1998], indexed families [Dybjær 1994], or inductive-recursive types [Dybjær and Setzer 1999]. Fortunately, there is a long tradition of generic programming with universes in Agda, arguably dating back to Martin-Löf [1984]. It would be worthwhile exploring how to extend our construction to more general universes, such as the context-free types [Altenkirch et al. 2007], containers [Abbott et al. 2005; Altenkirch et al. 2015], or the ‘sigma-of-sigma’ universe [Chapman et al. 2010; Oury and Swierstra 2008].

Doing so would allow us to exploit dependent types further in the definition of our evaluators. For example, we might then define an interpreter for the well-typed lambda terms and derive a tail recursive evaluator automatically, rather than verifying the construction by hand [Swierstra 2012].

The termination proof we have given defines a well-founded relation and shows that this decreases during execution. There are other techniques for writing functions that are not obviously structurally recursive, such as the Bove-Capretta method [Bove and Capretta 2005], partiality monad [Danielsen 2012], or coinductive traces [Nakata and Uustalu 2009].

In contrast to the well-founded recursion used in this paper, however, these methods do not yield an evaluator that is directly executable, but instead defer the termination proof.

Given that we can – and indeed have – shown termination of our tail-recursive abstract machines, the abstract machines are executable directly in Agda.

One drawback of our construction is that the stacks now not only store the value of evaluating previously visited subtrees, but also records the subtrees themselves. Clearly this is undesirable for an efficient implementation. It would be worth exploring if these subtrees may be made computationally irrelevant – as they are not needed during execution, but only used to show termination and correctness. One viable approach might be porting the development to Coq, where it is possible to make a clearer distinction between values used during execution and the propositions that may be erased.

## References


From algebra to abstract machine: a verified generic construction

TyDe'18, September 23–29, 2018, St. Louis, MO, USA

10.1145/1328438.1328474

Peter Morris, Thorsten Altenkirch, and Conor McBride. 2006. Exploring the Regular Tree Types. In Types for Proofs and Programs (TYPES 2004) (Lecture Notes in Computer Science), Christine Paulin-Mohring

Jean-Christophe Filliatre and Benjamin Werner (Eds.).


